UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación para videojuegos II

Unidad 3
Tilemaps y Scrolling

Docente
Pablo Abratte

CONTENIDO

1. Tile maps

En las primeras épocas de desarrollo de juegos para PC, la economía de recursos era un factor mucho más problemático de lo que es hoy, ya que no se disponía de demasiada memoria RAM para almacenar imágenes y la velocidad de los microprocesadores resultaba insuficiente para redibujar grandes regiones de la pantalla de forma rápida y suave.

Una solución fue representar las imágenes grandes, utilizando un conjunto limitado de imágenes más pequeñas. De esta manera, fue posible crear niveles de mayor tamaño y complejidad, ocupando menor cantidad de memoria, e incrementar la velocidad de renderizado actualizando sólo algunos fragmentos de la pantalla. Esta técnica, conocida como *tiled rendering* o también como *tile map,* se volvió ampliamente usada en géneros específicos como juegos de plataformas y RPG (Role-Playing Game), teniendo un auge durante la era de consolas de 8 y 16 bits.

A pesar de que las necesidades de hardware, en la actualidad, son muy distintas a las de aquel entonces, los tile maps siguen teniendo vigencia y gozan de mucha popularidad, ya que tornan más sencilla la creación y edición de niveles de gran tamaño y complejidad, utilizando relativamente pocas piezas gráficas. Además de las ventajas de rendimiento mencionadas anteriormente, también facilitan la implementación de estructuras de ordenamiento espacial y detección de colisiones.

Un *tile map* consiste en un arreglo bidimensional de tiles (o mosaicos). Cada uno de ellos consiste en un objeto representado por una imagen y posee atributos referentes, generalmente, a la lógica del juego. Estos atributos indican, por ejemplo, si un personaje puede caminar sobre el tile o si recibe daño al tocarlo. El conjunto de todas las imágenes para representar un tile se conoce como *tileset.*

Tile map
Un *tile map* consiste en un arreglo bidimensional de tiles (o mosaicos). Cada uno de ellos consiste en un objeto representado por una imagen y posee atributos referentes, generalmente, a la lógica del juego.

Los *tiles* tienen usualmente formas simples como cuadrados o rectángulos. Éstos son dispuestos en forma de grilla, espaciados según el alto/ancho de las imágenes utilizadas para representarlos, las cuales tienen el mismo tamaño.

En la siguiente figura es posible observar el conjunto de imágenes que conforman un tileset y un escenario generado a partir de dichas imágenes:

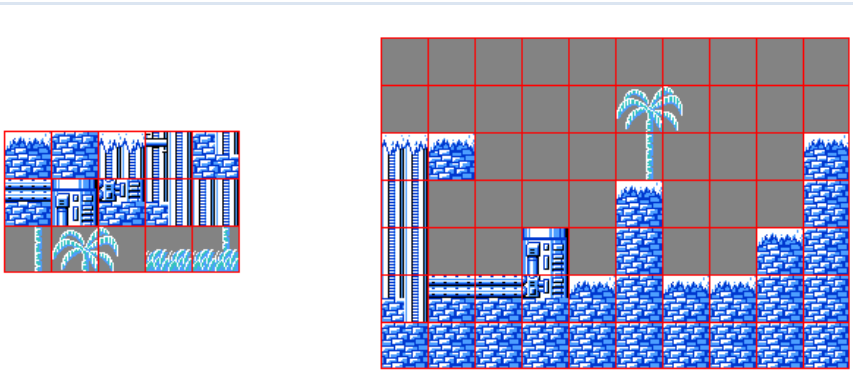

Figura 1. Un tileset y un nivel formado con el mismo.

Algunas variaciones del modelo incluyen: tiles con formas más complejas o con vista isométrica; la utilización de capas separadas para describir el escenario y los objetos con los que se puede interactuar, así como agrupamientos de tiles, conocidos como *supertiles* o *chunks.*

Propondremos, a continuación, estructuras y funciones para representar un tile map. Nuevamente, aclaramos que la implementación propuesta no es la única forma de hacerlo, ni necesariamente la más adecuada para cualquier caso.

Para representar un tile, utilizaremos la clase mostrada a continuación:

```
class Tile: public sf::Sprite{
public:
    int iImage;            // el numero de imagen
    bool solid;            // si se puede chocar con el tile
    sf::FloatRect rect;    // para facilitar la
                           // deteccion de colisiones
};
```

La clase *Tile* propuesta hereda los atributos de *sf::Sprite* para poder asignarle una imagen y una posición. Utilizaremos, además, un entero para indicar el número de imagen del tileset con la que el tile deberá ser dibujado. La variable *solid* servirá para especificar si se puede chocar con el tile o pararse sobre él. El atributo *rect* guardará la región rectangular que el tile ocupa dentro del nivel, es decir, su *bounding box*, y será usado para el cálculo de colisiones entre los personajes y el escenario.

Será posible ampliar la clase si se agregan atributos que permitan cubrir otras situaciones del juego. Por ejemplo, si el tile representa un objeto que puede romperse o si el personaje recibe daño al tocarlo, etc.

Presentaremos, ahora, una clase para representar un nivel como un tile map y explicaremos algunas de sus funciones. La declaración de la clase se presenta a continuación:

```
class Nivel{
private:
    // la estructura para representar un tile
    class Tile: public sf::Sprite{
         public:
         int iImage;            // el numero de imagen
         bool solid;            // si se puede chocar con el tile
         sf::FloatRect rect;    // para facilitar la
                                // deteccion de colisiones
    };

    // nombre del archivo tileset
    string tileset_filename;
    // tamano del tileset en tiles (ancho x alto)
    sf::Vector2i tileSetSize;
    // tamano de los tiles (ancho x alto)
    sf::Vector2i tileSize;
    // manejador de las imagenes del tileset
    SpriteSheetManager sm;

    // la matriz (o vector de vectores) de tiles
    vector< vector<Tile> > tiles;
    // tamano del nivel en tiles (ancho x alto)
    sf::Vector2i levelSize;

    // inicializa la matriz de tiles
    void Init();

    public:
    // constructor
    Nivel(string level_file);

    // salvar y guardar un nivel
    void Load(string file);
    void Save(string file);

    // dibujar el nivel en pantalla
    void Draw(sf::RenderWindow &w);
};
```

Como vemos, la clase *Tile* definida anteriormente es declarada esta vez dentro de la clase *Nivel*. Es así porque la primera será utilizada internamente por la segunda y no será necesario que esté disponible para el usuario, por lo que su encapsulamiento nos permitirá alcanzar un diseño más robusto.

Utilizaremos un objeto del tipo *SpriteSheetManager* ya desarrollado para cargar y acceder al tileset con el que generaremos el nivel. Las variables *tileset_filename* y *tileSetSize* guardarán el nombre del archivo de imagen que contiene el tileset y la cantidad de tiles que el mismo posee a lo largo y a lo ancho. Será útil almacenar estos valores para luego volcar los datos de nuestro nivel en un archivo.

Como dijimos, un tile map es básicamente una matriz de tiles. La variable *tiles* será el arreglo bidimensional que utilizaremos para representar nuestro nivel y en *levelSize* se guardará su ancho y su alto. Dado que el tamaño del nivel se desconoce de antemano, será necesario encontrar una forma de reservar dinámicamente la memoria de la matriz, para lo que utilizaremos un vector que contenga a otros vectores (de tiles). Debemos notar que, de esta manera, las filas de la matriz (vectores de tiles) no estarán contiguas en memoria como sí ocurriría con un arreglo bidimensional reservado de forma estática. Tanto el dimensionamiento de la matriz como la inicialización de su contenido serán realizados por la función *Init().*

Analizaremos, ahora, la implementación de algunas funciones de la clase propuesta para comprender su funcionamiento. El constructor, mostrado a continuación, simplemente recibe el nombre de un archivo de texto que contiene la información del nivel y llama al método *Load()* para cargarlo:

```
// Constructor: cargar el nivel desde el
// archivo level_file
Nivel::Nivel(string level_file){
        Load(level_file);
}
```

El código de la función *Load(),* encargado de cargar el nivel de un archivo de texto, es el siguiente:

```
// carga un nivel desde un archivo de nivel
void Nivel::Load(string filename){
    // abrimos el archivo
    ifstream entrada(filename.c_str());
    // leemos el nombre del archivo de tilesets
    getline(entrada,tileset_filename);
    // cargamos el tamano del tileset y del nivel
    entrada>>tileSetSize.x>>tileSetSize.y;
    entrada>>levelSize.x>>levelSize.y;

    // cargamos el tileset
    sm.Load(tileset_filename, tileSetSize.x, tileSetSize.y);
    tileSize.x=sm[0].GetWidth();
    tileSize.y=sm[0].GetHeight();

    // inicializamos la matriz de tiles
    Init();

    // leemos la matriz de numeros de imagenes
    for(unsigned i=0; i<levelSize.y; i++){
        for(unsigned j=0; j<levelSize.x; j++){
            entrada>>tiles[i][j].iImage;
        }
    }

    // leemos la matriz que nos indica cuales
    // tiles son solidos
    for(unsigned i=0; i<levelSize.y; i++){
        for(unsigned j=0; j<levelSize.x; j++){
            entrada>>tiles[i][j].solid;
        }
    }
    // cerramos el archivo
    entrada.close();

    // finalmente asignamos las imagenes a los tiles
    // (si su numero de imagen es distinto de -1)
    int iImg;
```

```
    for(unsigned i=0; i<levelSize.y; i++){
        for(unsigned j=0; j<levelSize.x; j++){
            iImg=tiles[i][j].iImage;
            if(iImg!=-1){
                tiles[i][j].SetImage(sm[iImg]);
            }
        }
    }
}
```

Este método abre el archivo y lee el nombre del archivo de imagen que contiene el tileset y luego su tamaño, así como también el tamaño del nivel. Una vez que se dispone de estos datos, puede cargarse el tileset y posteriormente llamar a la función Init(), que analizaremos más adelante, para inicializar la matriz de tiles utilizada para representar el nivel. Después de inicializar el arreglo, la función lo rellena con los datos de dos matrices que lee desde el archivo: la primera indicará los números de imagen con la que cada tile debe dibujarse y la segunda,cuáles son los tiles con los que los personajes pueden colisionar. Por último, a partir de los números leídos en la primera matriz, se asignará a cada tile la imagen correspondiente.

La función *Save()*, cuyo código no mostraremos, opera de manera similar a *Load()*, guardando los datos del nivel en un archivo de texto con el mismo formato con el que esos datos fueron leídos. Esta función es útil para desarrollar herramientas que nos permitan crear y editar niveles.

A continuación, mostramos la implementación de la función *Init(),* encargada de inicializar la matriz de tiles:

```
// inicializa la matriz de tiles
void Nivel::Init(){
    // vaciamos la matriz de tiles (por las dudas si se llama
    // a Load() mas de una vez)
    tiles.clear();
    tiles.resize(0);
    // variables temporales para ir llenando la matriz
    // una fila y un tile
    vector<Tile> filaTemp;
    Tile tileTemp;

    // para calcular la posicion en pantalla de cada tile
    int posx, posy;

    // inicializamos la matriz
    for(unsigned i=0; i<levelSize.y; i++){
        // vaciamos la fila y le insertamos todos los tiles
        filaTemp.clear();
        for(unsigned j=0; j<levelSize.x; j++){
            // calculamos la posision del tile
            posx=j*tileSize.x;
            posy=i*tileSize.y;
            tileTemp.SetPosition(posx, posy);
            // calculamos el rectangulo que va a ocupar el tile
            tileTemp.rect.Left=posx;
            tileTemp.rect.Right=posx+tileSize.x;
            tileTemp.rect.Top=posy;
            tileTemp.rect.Bottom=posy+tileSize.y;

            // inicializamos el numero de imagen
            tileTemp.iImage=-1;

            // insertamos al tile en la fila
            filaTemp.push_back(tileTemp);
        }
        // insertamos la fila en la matriz
        tiles.push_back(filaTemp);
    }
}
```

El método llama primero a *clear()* y *resize()* para borrar todos los elementos del vector y liberar su memoria. Esto es necesario en caso de que la matriz ya se encuentre poblada con un nivel cargado anteriormente.

Como hemos dicho, para representar la matriz de tiles usaremos un vector que guardará otros vectores. Para llenar esta matriz, utilizaremos un procedimiento similar al realizado hasta ahora con vectores, pero lo que insertaremos serán otros vectores. Por cada fila de la matriz, colocaremos los tiles en la variable *filaTemp,* y una vez que la fila esté completa, la insertaremos en la matriz. Luego, vaciaremos el vector *filaTemp* para repetir el procedimiento hasta haber insertado todas las filas. Antes de insertar cada tile, inicializaremos su posición y su bounding box. Guardar la posición de cada tile acelerará el proceso de dibujado del tile map, mientras que su bounding box será necesario para el cálculo de colisiones. También se inicializará el número de imagen del tile a -1. Este valor será utilizado para indicar que no le corresponde ninguna imagen y no deberá, por lo tanto, ser dibujado. Como puede observarse en la parte del código que posiciona los tiles, la esquina superior izquierda del nivel se encontrará en las coordenadas (0,0). Este detalle deberá ser tenido en cuenta más adelante, cuando coloquemos a nuestros personajes en el escenario, ya que tomaremos al nivel como sistema de referencia para las coordenadas de los mismos.

En la *Figura 2* se ilustra este sistema de referencia:

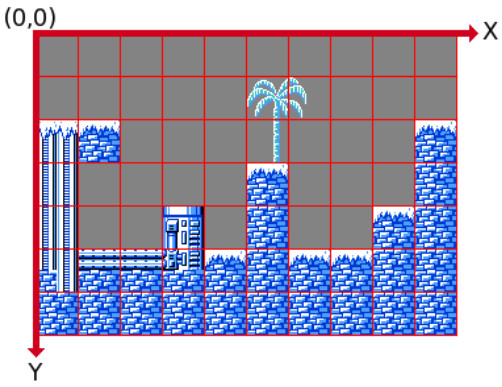

Figura 2. Sistema de referencia del nivel.

Al haber asignado la posición e imagen del tile en las funciones *Init()* y *Load(),* respectivamente, dibujar el tilemap sólo implica recorrer la matriz de tiles y dibujar los que tengan una imagen asignada (recordemos que cada tile es un sprite).

El siguiente código realiza esta tarea:

```
// dibuja el nivel en pantalla
void Nivel::Draw(sf::RenderWindow &w){
    for(unsigned i=0; i<levelSize.y; i++){
        for(unsigned j=0; j<levelSize.x; j++){
            if(tiles[i][j].iImage!=-1)
                w.Draw(tiles[i][j]);
        }
    }
}
```

A continuación, mostramos el contenido de un archivo de texto con la información de un nivel *(Figura 3)* y la representación gráfica de dicha información *(Figura 4):*

```
megaman_iceman_level.png
5 3
10 7
```

```
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  11  12  -1  -1  -1
 2   0  -1  -1  -1  10  -1  -1  -1   0
 9  -1  -1  -1  -1   0  -1  -1  -1   1
 9  -1  -1   6  -1   1  -1  -1   0   1
 8   5   5   7   0   1   0   0   1   1
 1   1   1   1   1   1   1   1   1   1

 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 1   1   0   0   0   0   0   0   0   1
 1   0   0   0   0   1   0   0   0   1
 1   0   0   1   0   1   0   0   1   1
 1   1   1   1   1   1   1   1   1   1
 1   1   1   1   1   1   1   1   1   1
```

Figura 3. Archivo que contiene la representación del nivel en la *Figura 2*.

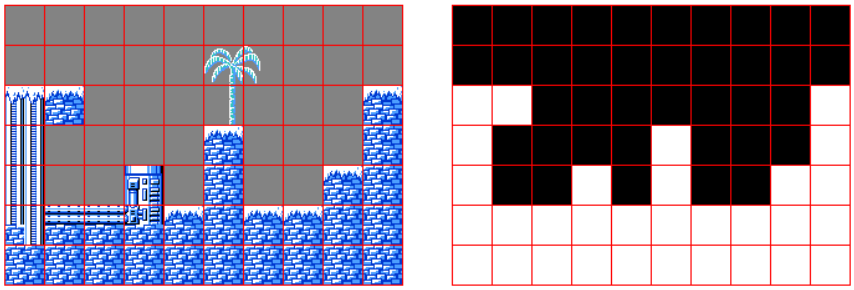

Figura 4. Representación del nivel descrito en la *Figura 3*. A la izquierda, el dibujo del nivel; a la derecha, los tiles sólidos con los que puede colisionarse son representados en blanco.

2. Detección de colisiones en un tile map

En el parágrafo anterior aprendimos a crear un nivel utilizando un tile map, al cual agregamos características para saber cuándo nuestro personaje podía colisionar con un tile. Ahora, abordaremos la problemática de la detección de colisiones entre un personaje y el escenario.

Como dijimos anteriormente, ubicaremos a nuestro personaje dentro del nivel según un sistema de referencia en el cual el origen de coordenadas coincide con el tile superior izquierdo del nivel *(Figura 2)*.

Lo primero que necesitamos es definir una figura de colisión para nuestro personaje. La misma será un *Axis-Aligned Bounding Box* (AABB). Esta elección no es casual, sino que se debe a que las figuras de colisión de los tiles del escenario también son AABBs. Además, la detección de colisiones de este tipo de figuras es sencilla, computacionalmente eficiente y, a pesar de no ser demasiado precisa, resulta adecuada en una gran cantidad de casos.

Hasta aquí hemos dibujado los sprites de nuestro personaje centrados en las coordenadas del mismo. De igual manera, definiremos su bounding box como un rectángulo, cuyo centro serán las coordenadas de nuestro personaje. Así, a medida que el mismo se desplace, el bounding box seguirá sus movimientos.

No es necesario que el bounding box contenga a todo el personaje. En muchos casos se logran mejores resultados cuando envuelve su figura de manera aproximada o sólo las partes más importantes. Tampoco es buena idea cambiar el tamaño del AABB según el cuadro de la animación, ya que los cambios en el AABB dificultan considerablemente la detección y el manejo de colisiones.

En la *Figura 5* es posible observar un AABB definido para *Megaman,* dibujado sobre diversos cuadros de sus animaciones. El AABB conserva siempre el mismo tamaño y, según el cuadro, envuelve la totalidad del personaje o sólo una parte significativa del mismo:

Figura 5. AABB para *Megaman* en distintos frames.

Para representar el AABB de nuestro personaje, utilizaremos la estructura sf::FloatRect, facilitada por SFML.

A continuación, mostramos el código de una función que, agregada la clase *Megaman,* permite obtener su AABB:

```
// devuelve el Axis-Aligned Bounding Box
// para Megaman
sf::FloatRect Megaman::GetAABB(){
        sf::Vector2f p=GetPosition();
        return sf::FloatRect(  round(p.x-10),
                               round(p.y-10),
                               round(p.x+8),
                               round(p.y+16));
}
```

La función devuelve un rectángulo, cuyas dimensiones fueron elegidas para envolver de manera ajustada al personaje, centrado en las coordenadas de dicho personaje.

Una vez definido el AABB de nuestro personaje, el problema de la detección de colisiones con el escenario se reduce a calcular si existe intersección entre dicho AABB y los AABBs pertenecientes a los tiles del escenario.

Comprobar si existe colisión con cada tile del escenario sería una tarea ineficiente e inútil, ya que el personaje sólo podrá colisionar con un grupo de tiles cercano al mismo.

La *Figura 6* ilustra esta situación. Los tiles resaltados en amarillo son los que se solapan con el bounding box de nuestro personaje y, por lo tanto, los posibles candidatos a colisionar con el mismo:

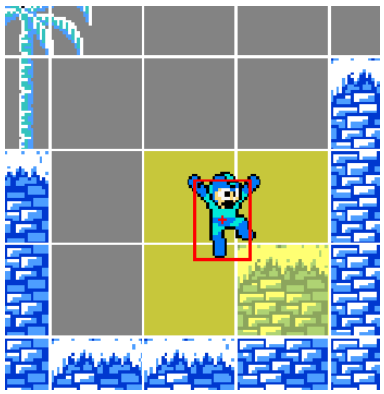

Figura 6. AABB de *Megaman* y tiles candidatos a colisionar con el mismo (amarillo).

La cantidad de tiles con los que el AABB del personaje puede solaparse depende de los tamaños de este último y de los tiles. En el caso mostrado en la *Figura 6,* esa cantidad nunca será mayor a cuatro.

Obtener una lista de los tiles solapados con una región rectangular es muy sencillo gracias a la estructura del tile map. La función que mostramos a continuación realiza esta tarea y será utilizada de forma interna por la clase *Nivel,* mostrada anteriormente, para el cálculo de colisiones:

```
// llena el vector ovTiles con las coordenadas de los tiles que se
// superponen con el rectangulo r, nos es util para detectar colisiones
void Nivel::GetOverlappingTiles( sf::FloatRect r,
                                 vector<sf::Vector2i> &ovTiles){
    // tanto i como j comienzan con las coordenadas
    // (en tiles) del rectangulo r
    for(int i=r.Top/tileSize.y; i<r.Bottom/tileSize.y; i++){
         for(int j=r.Left/tileSize.x; j<r.Right/tileSize.x; j++){
            ovTiles.push_back(sf::Vector2i(i, j));
        }
    }
}
```

La función recibe un rectángulo que representa el AABB del personaje y coloca en el vector *ovTiles* los tiles solapados con dicha región. El procedimiento es sencillo ya que, conociendo el tamaño de los tiles, puede calcularse dentro de qué tile se encuentra cada vértice del rectángulo *r*. A partir de esto, se recorren los tiles solapados con la región *r* y, dentro del vector *ovTiles,* se almacenan los números de fila y columna de los mismos.

Una vez determinada la lista de tiles que se solapan con el bounding box del personaje, habrá que confirmar si existe efectivamente una colisión lo cual implicará recorrerla e indagar si hay algún tile con la bandera de sólido en verdadero. En este sentido, será necesario definir un criterio para resolver qué pasará en caso de que exista colisión con más de un tile sólido a la vez. Finalmente, se deberá obtener el área en que los rectángulos se interpenetran para poder definir, en base a ella, la respuesta del personaje.

El siguiente código corresponde a la clase *Nivel* y permite saber si el AABB pasado por parámetro colisiona con el escenario:

```
// revisa si hay o no colision del rectangulo r con algun tile solido
// devuelve verdadero o falso segun haya colision o no
// devuelve en areaColision el area de interpenetracion con el tile
// en caso de haber colision con mas de un tile, devuelve
// el area de colision con el tile que tenga mayor area de colision
bool Nivel::HayColision(sf::FloatRect r, sf::FloatRect &areaColision){
    vector<sf::Vector2i> ovTiles;
    GetOverlappingTiles(r, ovTiles);
    sf::FloatRect intersec; float areaIntersec, maxArea=0;
     for(unsigned i=0; i<ovTiles.size(); i++){
        if(tiles[ovTiles[i].x][ovTiles[i].y].solid){
            r.Intersects(tiles[ovTiles[i].x][ovTiles[i].y].rect,
                         &intersec);
            areaIntersec=intersec.GetWidth()*intersec.GetHeight();
            if(areaIntersec>maxArea){
                maxArea=areaIntersec;
                areaColision=intersec;
            }
        }
    }
    return maxArea>0;
}
```

La función llama a *GetOverlappingTiles(),* pasándole el rectángulo *r* para conseguir los tiles que se solapan con dicha región. Luego, recorre la lista y, si alguno de ellos es sólido, se calcula la región de la intersección. En caso de existir más de un tile sólido, el criterio elegido consiste en conservar el área de colisión más grande. La función

retorna un booleano indicando si existe o no colisión y el área de la misma en la variable *areaColisión*.

Como vemos, se utiliza el método *Intersects()* de *sf::Rect* para conocer el área de colisión o solapamiento entre dos rectángulos. El prototipo de esta función es el siguiente:

```
bool Intersects(const Rect<T> &Rectangle, Rect<T> *OverlappingRect=NULL);
```

La función devuelve verdadero o falso según el rectángulo que realizó la llamada y *Rectangle* se intersecan o no. Opcionalmente, la función puede recibir un puntero a otro rectángulo en el que devolverá el área de colisión.

Podemos utilizar la función *HayColision()* para modificar el comportamiento de nuestro personaje según las colisiones con el nivel. La lógica del manejo de colisiones consistirá en predecir la posición futura del personaje y evitar el movimiento, si éste lo llevase a un estado de colisión. De esta manera, el bounding box de nuestro personaje nunca llegará a penetrar un tile porque lo evitaremos de antemano.

El algoritmo, a grandes rasgos, consistirá en lo siguiente:

1. Nuestro personaje parte desde una posición en la que no colisiona con ningún tile. Esto puede observarse en la *Figura 7a,* en la que, si bien *Megaman* está pegado al suelo, no está colisionando con él ni con otro tile sólido.
2. En el caso de tener que mover al personaje, se calcula la posición en la que el mismo se encontrará en el próximo período de tiempo. Si en dicha posición no existirá colisión con el escenario, entonces podrá moverse al personaje hasta allí. Esta situación es ilustrada en la *Figura 7b.*
3. Si la posición futura implica colisionar con un tile sólido, entonces no podremos mover al personaje, ya que quedará dentro del tile *(Figura 7c).* Sin embargo, tampoco es correcto dejarlo donde está, ya que puede estar aún lejos del tile con el que colisiona. Lo que debemos hacer es mover al personaje en la dirección deseada, pero no en la distancia que se movería normalmente, sino aquella que lo deje pegado al tile pero sin colisionar, como se ve en la *Figura 7d.* Esta distancia que permite lograr el ajuste al tile es obtenida a partir del área de colisión de la posición futura. Dicho ajuste es acompañado generalmente por un cambio de estado para evitar que el personaje continúe moviéndose.

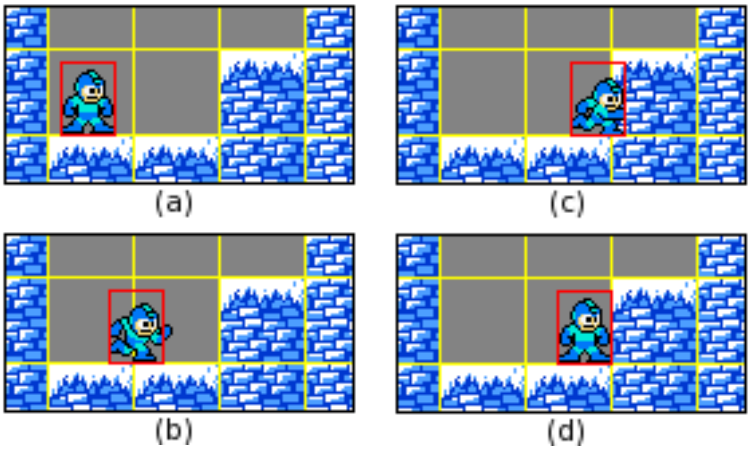

Figura 7. a) Posición inicial del personaje en la que no ocurren colisiones. b) Posición futura en la que no ocurrirán colisiones. c) Posición futura en la que ocurrirá una colisión. d) Posición ajustada al borde del tile para evitar la colisión.

Abordaremos, ahora, los detalles de la implementación para que nuestro personaje *Megaman* pueda responder a la topología del nivel, necesitará conocerlo. Para ello, modificaremos la clase para que guarde un puntero al nivel en el que se encuentra.

Agregaremos, además, nuevas funciones y atributos que serán necesarios para el manejo de colisiones.

El siguiente código muestra las modificaciones introducidas en la clase:

```
class Megaman: public Sprite{
        private:
        Estado estado;  // el estado actual

        ...

        Nivel *n;       // el nivel, para detectar colisiones

        // funciones para saber si hubo colision en las distintas
        // direcciones
        bool ChocaraPared(float dt, float &distAjuste);
        bool ChocaraTecho(float dt, float &distAjuste);
        bool ChocaraSuelo(float dt, float &distAjuste);


        public:
        // constructor
        Megaman(ManejadorDisparos *d, Nivel *n);

        ...

        // realiza el movimiento, detecta colisiones y anima
        void Mover_y_Animar(Joystick j, float dt);

        // devuelve el bounding box de Megaman
        sf::FloatRect GetAABB();
};
```

Como mencionamos, es necesario que *Megaman* pueda interactuar con el nivel para saber si es capaz de moverse a una determinada posición. Para ello, guardará un puntero (*n*) al nivel donde se encuentra, el cual deberá ser pasado al constructor en el momento de la creación del personaje.

Las funciones *ChocaraPared()*, *ChocaraTecho()* y *ChocaraSuelo()* recibirán el tiempo transcurrido y calcularán la posición futura del personaje en una determinada dirección. Estas funciones devolverán falso, en caso de no existir colisión al moverse en dicha dirección, y retornarán verdadero si ocurre lo contrario, actualizando la variable *distAjuste,* pasada por referencia, con la máxima distancia que es posible moverse en esa dirección sin colisionar.

Estas funciones serán llamadas desde *Mover_y_Animar()*.

A continuación, mostramos el código de dichas funciones:

```
bool Megaman::ChocaraPared(float dt, float &distAjuste){
        // creamos rectangulos para el bounding box actual
        // y el area de colision
        FloatRect aabb, areaColision;

        // conseguimos el bounding box en la posicion actual
        aabb=GetAABB();
        bool chocaPared;

        // la distancia que nos moveriamos
        float despl=dt*vx*direccion;

        // buscamos el bounding box que tendriamos
        // si nos moviesemos, preguntamos si
        // colisionamos y la distancia
        // que nos podriamos mover sin colisionar (ajuste)
        aabb.Left+=despl;
        aabb.Right+=despl;
        // calculamos si habria colision
```

```
        chocaPared=n->HayColision(aabb, areaColision);
        distAjuste=direccion*(dt*vx-areaColision.GetWidth());
        return chocaPared;
}
```

```
bool Megaman::ChocaraTecho(float dt, float &distAjuste){
        bool chocaConTecho;
        // calculamos la velocidad que tendriamos
        float newvy=vy+gravity*dt;
        // si estamos cayendo, no podemos chocar con
        // el suelo
        if(newvy>0) return false;
        else{
                FloatRect aabb, areaColision;

                // la distancia que nos vamos a mover
                float despl=dt*newvy;

                // conseguimos el AABB actual y calculamos
                // el que tendriamos en un instante de tiempo
                aabb=GetAABB();
                aabb=aabb;
                aabb.Top+=despl;
                aabb.Bottom+=despl;
                // calculamos si habria colision
                chocaConTecho=n->HayColision(aabb, areaColision);
                distAjuste=-(-despl-areaColision.GetHeight());
        }
        return chocaConTecho;
}
```

```
bool Megaman::ChocaraSuelo(float dt, float &distAjuste){
        bool chocaConSuelo;
        // calculamos la velocidad que tendriamos
        float newvy=vy+gravity*dt;
        // si estamos subiendo, no podemos chocar con
        // el suelo
        if(newvy<0) return false;
        else{
                FloatRect aabb, areaColision;
                aabb=GetAABB();
                // la distancia que nos vamos a mover
                float despl=dt*newvy;

                // conseguimos el AABB actual y calculamos
                // el que tendriamos en un instante de tiempo
                aabb.Top+=despl;
                aabb.Bottom+=despl;
                // calculamos si habria colision
                chocaConSuelo=n->HayColision(aabb, areaColision);
                distAjuste=despl-areaColision.GetHeight();
        }
        return chocaConSuelo;
}
```

La función *ChocaraPared()* permite saber si el personaje colisionará al moverse horizontalmente, ya sea hacia la derecha o izquierda, según la variable *direccion*. Para esto, calcula la posición en la que el bounding box se encontrará en el próximo instante de tiempo.

Las funciones *ChocaraSuelo()* y *ChocaraTecho()* funcionan de manera similar, pero necesitan calcular la velocidad futura antes de establecer cuál será la próxima posición. Además, tienen en cuenta que, si nuestro personaje está cayendo, no puede colisionar con el techo; mientras que, si está subiendo, no puede hacerlo con el suelo.

A continuación, mostramos un fragmento de la función *Mover_y_Animar()* con las modificaciones necesarias para reaccionar ante las colisiones con el nivel:

```
// Realiza los cambios de estado, movimientos y animacion del personaje
void Megaman::Mover_y_Animar(Joystick j, float dt){
     // la distancia para ajustar en caso de colision
     float distAjuste;

     switch(estado){
         case PARADO:
             if(j.left){
                 direccion=-1;
                 // si podemos correr, lo hacemos
                 if(!ChocaraPared(dt, distAjuste))
                    CambiarEstado(CORRIENDO);
                 else
                  // si no hay lugar para correr,
                  // nos ajustamos a la pared,
                  // pero no cambiamos el estado
                  Move(distAjuste, 0);
             }else if(j.right){
                 // lo mismo si es hacia la derecha
                 direccion=1;
                 if(!ChocaraPared(dt, distAjuste))
                     CambiarEstado(CORRIENDO);
                 else
                     Move(distAjuste, 0);
             }

             ...

             // si no hay suelo debajo, empezamos a caer
             if(!ChocaraSuelo(dt, distAjuste)){
                 CambiarEstado(SALTANDO);
                 vy=0;
             }
             break;

         case SALTANDO:
             if(j.left){
                 direccion=-1;
                 // si no chocamos la pared, empezamos a movernos
                 if(!ChocaraPared(dt, distAjuste)){
                     CambiarEstado(SALTANDO_Y_MOVIENDOSE);
                 }else{
                     // si no, ajustamos a la pared sin cambiar el estado
                     Move(distAjuste, 0);
                 }
                 // lo mismo para la derecha
            }else if(j.right){
                 direccion=1;
                 if(!ChocaraPared(dt, distAjuste)){
                     CambiarEstado(SALTANDO_Y_MOVIENDOSE);
                 }else{
                     Move(distAjuste, 0);
                 }
            }

            // si tocamos el suelo, cambiamos el estado y ademas
            // debemos ajustar nuestra posicion al piso para que nuestro
            // personaje no quede penetrando el tile
            if(ChocaraSuelo(dt, distAjuste)){
                 CambiarEstado(PARADO);
                 Move(0, distAjuste);
                 vy=0;
            }else if(ChocaraTecho(dt, distAjuste)){
                 // lo mismo si chocamos con el techo
                 Move(0, distAjuste);
                 vy=0;
            }
         break;
```

```
    ...
    }

    ...
}
```

Se utilizan llamadas a las funciones definidas anteriormente para decidir cómo mover al personaje y a qué estado pasar. Por ejemplo, en caso de que el mismo se encuentre en estado PARADO y se presione la tecla izquierda, se llamará a la función *ChocaPared().* En caso de que no exista colisión, el personaje podrá pasar al estado CORRIENDO y moverse a la izquierda, mientras que en caso contrario se ajustará su posición para estar al lado del tile, sin cambiar de estado. Si fuera posible que el personaje se moviera hacia abajo sin colisionar (*ChocaraSuelo()* devuelve falso), entonces no hay nada debajo de él y debe comenzar a caer. En esta ocasión se utiliza el estado SALTANDO para representar tanto un salto como una caída. Por otro lado, si el personaje está cayendo y en su próxima posición colisionará con el suelo, deberá ajustar su posición al nivel del suelo y cambiar su estado a PARADO.

El código anterior no muestra la totalidad de la función, por lo que se recomienda el análisis del ejemplo completo para observar y comprender el resto del código.

La detección y manejo de colisiones con un tile map no siempre es una tarea trivial, ya que pueden intervenir muchas variables, dependiendo de la complejidad del personaje. Muchas veces es necesario realizar varios ajustes al código hasta lograr el funcionamiento correcto del mismo.

Hay muchas formas de plantear la detección y manejo de colisiones, por lo que el método propuesto es sólo una aproximación que puede resultar adecuada en mayor o menor medida, dependiendo de las circunstancias.

3. Scrolling

Esta técnica es generalmente utilizada para simular una cámara que sigue al personaje a medida que éste se desplaza por el nivel. Los primeros videojuegos en usarla datan aproximadamente de la segunda mitad de 1970.

Para implementar scrolling haremos uso de la clase *sf::View,* provista por SFML. Ésta nos permite definir la región del mundo virtual de nuestro juego que será visualizada en la ventana. Toda ventana de renderizado *(sf::RenderWindow)* tiene siempre una vista asociada a ella.

Una vista puede definirse a través de un rectángulo o especificando su centro y las mitades de su ancho y alto, como se muestra en el siguiente código:

```
// vista definida a partir de un rectangulo
sf::FloatRect r(0,0,300,300);
sf::View v1(r);

// vista definida a partir de su centro y su medio tamanio
sf::Vector2f centro_vista2(150, 150);
sf::Vector2f medioTamanio_vista2(150, 150);
sf::View v2(centro_vista2, medioTamanio_vista2);
```

Las dos vistas definidas en el código refieren a la misma región de nuestro mundo virtual: la zona rectangular delimitada por los vértices (0, 0) y (300, 300).

Una vez definida la vista, ésta debe ser asignada a la ventana mediante la función *SetView()* de la clase *sf::RenderWindow,* como se muestra en el siguiente ejemplo:

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

int main(int argc, char *argv[]) {
    // creamos un sprite
    sf::Image i;
    i.LoadFromFile("mario.png");
    sf::Sprite s;
    s.SetImage(i);

    // creamos tres ventanas diferentes
    sf::RenderWindow ventana1(sf::VideoMode(300,300),"Vista 1");
    sf::RenderWindow ventana2(sf::VideoMode(300,300),"Vista 2");
    sf::RenderWindow ventana3(sf::VideoMode(300,300),"Vista 3");

    // creamos una vista diferente para cada ventana
    sf::View vista1(sf::FloatRect(0,0,224,224));
    sf::View vista2(sf::Vector2f(128,128), sf::Vector2f(64,64));
    sf::View vista3(sf::FloatRect(0,128,224,224));

    // asignamos las vistas a las ventanas
    ventana1.SetView(vista1);
    ventana2.SetView(vista2);
    ventana3.SetView(vista3);


    sf::Event e;
    bool salir=false;
    while(!salir){
        while(  ventana1.GetEvent(e) ||
                ventana2.GetEvent(e) ||
                ventana3.GetEvent(e)){
            // si alguna ventana se cerro, salimos
            if(e.Type == e.Closed) salir=true;
        }

        // limpiamos las 3 ventanas
        ventana1.Clear(sf::Color(0,0,0));
```

Scrolling
El término *scrolling* se refiere al desplazamiento de la vista del usuario a lo largo de una imagen u objeto, que es más grande que la pantalla y del cual sólo puede verse una porción.

Clase *sf::View*
La clase *sf::View* nos permite definir la región del mundo virtual de nuestro juego que será visualizada en la ventana.

```
39.         ventana2.Clear(sf::Color(0,0,0));
40.         ventana3.Clear(sf::Color(0,0,0));
41.
42.         // dibujamos en las 3 ventanas
43.         ventana1.Draw(s);
44.         ventana2.Draw(s);
45.         ventana3.Draw(s);
46.
47.         // acualizamos las 3 ventanas
48.         ventana1.Display();
49.         ventana2.Display();
50.         ventana3.Display();
51.     }
52.     return 0;
53. }
54.
```

En este ejemplo se crea, a partir de una imagen en disco, un sprite ubicado por defecto en el origen de coordenadas. Se generan tres ventanas y se asigna una vista distinta a cada una: la vista de la primera abarca toda la extensión del sprite, mientras que la vista de la segunda muestra sólo la región rectangular, cuyos vértices son (64,64) y (192,192). En el caso de la tercera ventana, la vista muestra solamente la mitad inferior de la imagen y no conserva la misma relación de aspecto cuadrado de la ventana, por lo que la imagen se verá deformada, como puede apreciarse a continuación:

Figura 8. Resultado del programa anterior: la primera ventana muestra la imagen completa; la segunda, un fragmento de la misma utilizando la relación de aspecto de la ventana, y la tercera, otro fragmento pero no respetando la relación de aspecto de la ventana.

Una vez que se asigna una vista a la ventana, esta última no guarda una copia de la primera, sino una referencia a la misma. Por lo tanto, los cambios hechos en la vista afectarán los redibujados futuros de la ventana, sin necesidad de volver a llamar al método *SetView()*.

El siguiente ejemplo ilustra este comportamiento, creando una vista y cambiando, luego de asignarla a la ventana, su posición (pero no su tamaño):

```
1. #include <SFML/Window.hpp>
2. #include <SFML/Graphics.hpp>
3.
4. int main(int argc, char *argv[]) {
5.     // creamos la ventana
6.     sf::RenderWindow w(sf::VideoMode(600,600),"Ejemplo Vista 2");
7.
8.     // definimos la velocidad del scrolling
9.     const float viewVelx=15, viewVely=12;
10.
11.    // creamos un sprite con una imagen
12.    sf::Image i;
13.    i.LoadFromFile("level1.png");
14.    sf::Sprite s;
```

```
15.     s.SetImage(i);
16.
17.     // definimos el tamano de la vista y su ubicacion
18.     unsigned const tamanioVista=128;
19.     sf::View v;
20.     v.SetHalfSize(tamanioVista, tamanioVista);
21.     v.SetCenter(tamanioVista, tamanioVista);
22.     w.SetView(v);
23.
24.     // algunas variables que vamos a utilizar
25.     float elapsedTime;
26.     sf::Vector2f viewCenter;
27.
28.     while(w.IsOpened()) {
29.         sf::Event e;
30.         while(w.GetEvent(e)) {
31.             if(e.Type == e.Closed)
32.                 w.Close();
33.         }
34.
35.         // calculamos el tiempo que paso desde el ultimo frame
36.         elapsedTime=w.GetFrameTime();
37.
38.         // cambiamos la posicion de la vista
39.         v.Move(viewVelx*elapsedTime, viewVely*elapsedTime);
40.
41.         // si la vista se sale de la imagen, hacemos que "rebote"
42.         viewCenter=v.GetCenter();
43.         if(    viewCenter.x>i.GetWidth()-tamanioVista ||
44.             viewCenter.x<tamanioVista) viewVelx*=-1;
45.         if(    viewCenter.y>i.GetHeight()-tamanioVista ||
46.             viewCenter.y<tamanioVista) viewVely*=-1;
47.
48.         // limpiamos la pantalla y dibujamos
49.         w.Clear(sf::Color(0,0,0));
50.         w.Draw(s);
51.         w.Display();
52.     }
53.     return 0;
54. }
55.
```

En el ejemplo, se carga una imagen de forma similar al código mostrado anteriormente. Luego, se inicializa una vista y se la asigna a la ventana. En el bucle principal del juego, la vista es desplazada utilizando la función *Move(),* que modifica la posición del centro de la vista sin alterar su tamaño. Cabe destacar que el método *SetView()* es llamado solamente una vez al principio del programa. El código de las líneas 42 a 46 evita que la vista siga desplazándose más allá de los límites de la imagen, invirtiendo su velocidad de desplazamiento.

A continuación, mostraremos cómo aplicar lo que hemos visto hasta ahora para implementar scrolling en nuestros juegos. Para una referencia detallada de la clase *sf::View,* les sugerimos consultar la documentación de SFML en este enlace.

En primer lugar, introduciremos algunas modificaciones en la clase *Nivel,* definida anteriormente, para obtener funcionalidad de scrolling. Los agregados de la clase pueden apreciarse en el siguiente código:

```
class Nivel{

    ...

    // la vista del nivel
    sf::View levelView;

    ...

    public:
```

```
    ...

    // devuelve la vista del nivel
    sf::View &GetView();
    // inicializa la vista del nivel
    sf::View &InitLevelView(int res_x, int res_y,
                            int tiles_x=-1, int tiles_y=-1);
    // mueve la vista (realiza scrolling)
    void SetViewCenter(sf::Vector2f newCenter);
    // mueve la vista de forma suave
    void SetViewCenterSmooth(sf::Vector2f newCenter, float factor,
                             float dt);
};
```

El nivel guardará ahora, en la variable *levelView,* la porción del mismo que será visualizada en la ventana de juego. La vista podrá ser asignada a dicha ventana mediante la referencia devuelta por el método *GetView(),* cuya implementación puede observarse en este fragmento de código:

```
sf::View &Nivel::GetView(){
    return levelView;
}
```

La función *SetViewCenter()* nos permitirá desplazar el centro de la vista del nivel. Llamando a esta función con la posición del personaje, podremos desplazar la vista manteniéndola centrada en el mismo a medida que éste se desplaza por el nivel.

El código de la función es el siguiente:

```
void Nivel::SetViewCenter(sf::Vector2f newCenter){
    sf::Vector2f halfSize=levelView.GetHalfSize();
    sf::Vector2f levelSizeCoords;

    // las coordenadas maximas del nivel
    // para saber si nos salimos afuera
    levelSizeCoords.x=levelSize.x*tileSize.x;
    levelSizeCoords.y=levelSize.y*tileSize.y;

    // si el nuevo centro se sale del nivel,
    // lo corregimos
    if(newCenter.x-halfSize.x<0)
        newCenter.x=halfSize.x;
    else if(newCenter.x+halfSize.x>levelSizeCoords.x)
        newCenter.x=levelSizeCoords.x-halfSize.x;
    if(newCenter.y-halfSize.y<0)
        newCenter.y=halfSize.y;
    else if(newCenter.y+halfSize.y>levelSizeCoords.y)
        newCenter.y=levelSizeCoords.y-halfSize.y;

    // finalmente, ajustamos el centro de la vista
    levelView.SetCenter(newCenter);
}
```

Antes de desplazar el centro de la vista, la función revisa que la misma no se mueva fuera de la extensión del nivel. De esta forma, la cámara cesa su movimiento si el personaje ha llegado a un extremo de dicho nivel.

El método *SetViewCenterSmooth()* realiza una función similar al método anterior, pero en lugar de acomodar el centro de la vista en la posición del personaje, la desplaza gradualmente en dirección de la nueva posición, logrando un movimiento suave de la cámara.

La función se implementó de la siguiente manera:

```
void Nivel::SetViewCenterSmooth(sf::Vector2f newCenter, float factor,
                                float dt){
    sf::Vector2f oldCenter=levelView.GetCenter();
    float scrollDeltax, scrollDeltay;

    // se calcula el desplazamiento de la camara
    // en base al tiempo transcurrido y a un factor
    scrollDeltax=(newCenter.x-oldCenter.x)*(factor*dt);
    scrollDeltay=(newCenter.y-oldCenter.y)*(factor*dt);

    newCenter.x=oldCenter.x+scrollDeltax;
    newCenter.y=oldCenter.y+scrollDeltay;

    // se actualiza la posicion de la camara
    // llamando a SetViewCenter() para que realice
    // los chequeos de limites
    SetViewCenter(newCenter);
}
```

La función calcula el desplazamiento de la cámara en base al tiempo transcurrido y a la variable *factor,* que indicará la suavidad del desplazamiento. Cuanto más pequeño sea el valor de factor, más lento será el desplazamiento de la cámara. Finalmente, la función ajusta la nueva posición de la cámara llamando a *SetViewCenter()* para que realice los ajustes necesarios en caso de que el personaje llegue a un extremo del nivel.

Hasta ahora hemos presentado funciones para desplazar el centro de la vista, pero no dijimos nada sobre la forma de ajustar el tamaño de la misma. Como vimos anteriormente, al momento de inicializar la vista es necesario respetar la relación de aspecto de la ventana. Para ello debe tenerse en cuenta tanto las dimensiones de la ventana, como de la porción de nivel que se desea visualizar.

En el caso de que ambos (la ventana y el nivel) sean más anchos que altos *(Figura 9. a),* para respetar la relación de aspecto de la ventana se busca que la vista abarque toda la altura del nivel y se calcula el ancho adecuado para la misma, de forma que el scrolling ocurra solamente a lo ancho del nivel. Algo similar sucede si la ventana y el nivel son más altos que anchos *(Figura 9. d).*

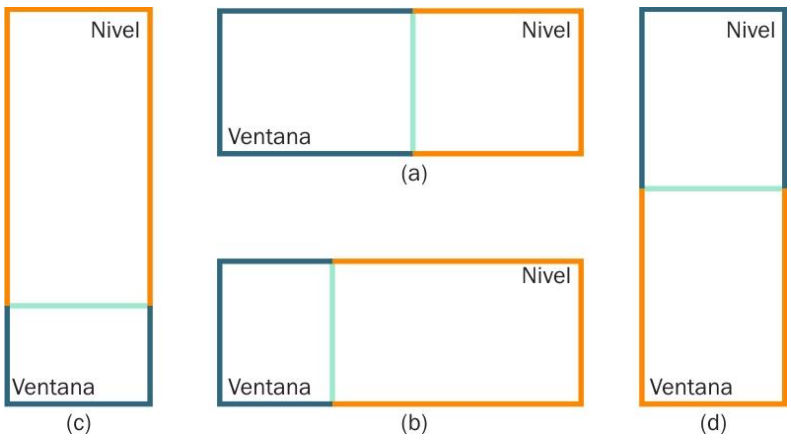

Figura 9. El tamaño de la ventana (celeste) y de un nivel o porción del mismo que desea visualizarse (anaranjado) en distintos casos: a) El nivel y la ventana son más anchos que altos. b) El nivel es más ancho que alto y la ventana más alta que ancha. c) El nivel es más alto que ancho y la ventana más ancha que alta. d) La ventana y el nivel son más altos que anchos.

Ahora bien, cuando la ventana es más ancha que alta y el nivel más alto que ancho *(Figura 9. c),* o viceversa *(Figura 9. b),* debemos lograr que el lado mayor de la vista abarque todo el lado menor del nivel, en tanto que el lado menor de la misma se calcula en base a la relación de aspecto de la ventana.

Una forma sencilla de entender este procedimiento es la siguiente: imaginemos que un rectángulo pequeño se coloca con la relación de aspecto de la ventana dentro de otro rectángulo más grande que representa el nivel. Luego, se hace crecer el rectángulo pequeño conservando su relación alto / ancho hasta que alguno de sus lados iguala al lado homónimo del rectángulo mayor.

La función *InitLevelView(),* mostrada a continuación, realiza esta tarea, inicializando el tamaño de la vista a partir de la resolución de la ventana o pantalla y la cantidad aproximada de tiles que desean visualizarse, tanto a lo ancho como a lo alto. Estos valores son pasados a la función como parámetros. Para establecer relación con la *Figura 9* debemos tener en cuenta que los parámetros *tiles_x* y *tiles_*y indican la porción del nivel que desea visualizarse, es decir, el rectángulo dibujado con color anaranjado.

```
// inicializa la vista del nivel, recibe:
// la resolucion de la ventana o pantalla
// el tamanio aproximado de la porcion del nivel que desea
// verse (si se omite se toma todo el nivel)
sf::View &Nivel::InitLevelView(   int res_x, int res_y,
                                  int tiles_x=-1, int tiles_y=-1){
    // si el tamanio deseado de la vista es -1
    // entonces deseamos visualizar todo el nivel
    if(tiles_x<0) tiles_x=levelSize.x;
    if(tiles_y<0) tiles_y=levelSize.y;

    // la cantidad real de tiles que abarcara la vista
    float realtiles_x, realtiles_y;

    // calculamos la cantidad real de tiles de la vista
    if(res_x>res_y){
        if(tiles_x>tiles_y){ // caso Fig 8 a
            realtiles_y=tiles_y;
            realtiles_x=(res_x*tiles_y)/float(res_y);
        }else{ // caso Fig 8 b
            realtiles_x=tiles_x;
            realtiles_y=(res_y*tiles_x)/float(res_x);
        }
    }else{
        if(tiles_x>tiles_y){ // caso Fig 8 c
            realtiles_y=tiles_y;
            realtiles_x=(res_y*tiles_x)/float(res_x);
        }else{ // caso Fig 8 d
            realtiles_x=tiles_x;
            realtiles_y=(res_y*tiles_x)/float(res_x);
        }
    }

    // asignamos el tamaño a la vista
    levelView.SetHalfSize(  sf::Vector2f(realtiles_x*tileSize.x/2.0,
                            realtiles_y*tileSize.y/2.0));
    // llamamos a SetViewCenter() para colocarla en una
    // posicion valida
    SetViewCenter(sf::Vector2f(0,0));

    // devolvemos una referencia a la vista
    return levelView;
}
```

En caso de que no se especifique el tamaño de la vista deseada, se ajustará a todo el nivel. Las variables *realtiles_x* y *realtiles_y* guardarán el tamaño real de la vista en tiles y éstos serán calculados según el tamaño deseado de la vista y la relación de aspecto de la ventana. Después, se asignará el tamaño a la vista, se posicionará llamando a *SetViewCenter()* y se devolverá una referencia a la misma para proveer de mayor flexibilidad a la función.

A continuación, mostramos fragmentos de código de la función principal del ejemplo que hace uso de las funciones vistas anteriormente:

```
int main(int argc, char *argv[]) {
    // la resolucion de la ventana
    int const resx=800, resy=600;
    // creamos la ventana
    sf::RenderWindow w(VideoMode(resx,resy),"Megaman");
    // creamos el nivel
    Nivel n("Mylevel.lev");
    // inicializamos la vista del nivel
    n.InitLevelView(resx, resy);
    // asignamos la vista del nivel a la ventana
    w.SetView(n.GetView());

    ...

    while(w.IsOpened()) {
        // manejamos los eventos
        while(w.GetEvent(e)) {
            if(e.Type == e.Closed)
                w.Close();

        ...

        // actualizamos el estado de Megaman
        megaman.Mover_y_Animar(j, clk.GetElapsedTime());
        // movemos la vista segun la posicion del personaje
        n.SetViewCenter(megaman.GetPosition());

        ...

        // limpiamos la ventana
        w.Clear(Color(10,10,20));

        // dibujamos el nivel y Megaman
        n.Draw(w);
        n.Draw(megaman);

        // actualizamos la ventana
        w.Display();
    }
    return 0;
}
```

Finalmente, resta prestar atención a un aspecto no menor: ahora que visualizamos sólo una región de nuestro nivel, resulta ineficiente e innecesario realizar el dibujado de todos los tiles.

Por ello, en el siguiente fragmento de código podrán observar el método *Draw()* de la clase *Nivel,* modificado para redibujar solamente los tiles visibles en pantalla, haciendo uso de la función *GetOverlappingTiles(),* desarrollada anteriormente:

```
void Nivel::Draw(sf::RenderWindow &w){
    vector<sf::Vector2i> ovTiles;
    GetOverlappingTiles(levelView.GetRect(), ovTiles);
    Tile temp;
    for(unsigned i=0; i<ovTiles.size(); i++){
        temp=tiles[ovTiles[i].x][ovTiles[i].y];
        if(temp.iImage!=-1){
            w.Draw(temp);
        }
    }
}
```

Una vez más, recomendamos la observación del código completo del ejemplo para contemplar otros detalles que no fueron expuestos aquí.

4. Parallax scrolling

Parallax scrolling consiste en una técnica en la que varias imágenes de fondo, parcialmente transparentes y apiladas una sobre la otra, se mueven a distintas velocidades, dando la ilusión de profundidad en un videojuego 2D.

Parallax scrolling
Consiste en una técnica en la que varias imágenes de fondo parcialmente transparentes y apiladas una sobre la otra, se mueven a distintas velocidades dando la ilusión de profundidad en un videojuego 2D.

La técnica se hizo popular en el juego *Moon Patrol,* de 1982, y se volvió ampliamente utilizada debido a su sencillez y sus vistosos resultados.

En la *Figura 10* mostramos un conjunto de layers o capas utilizadas para implementar parallax scrolling.

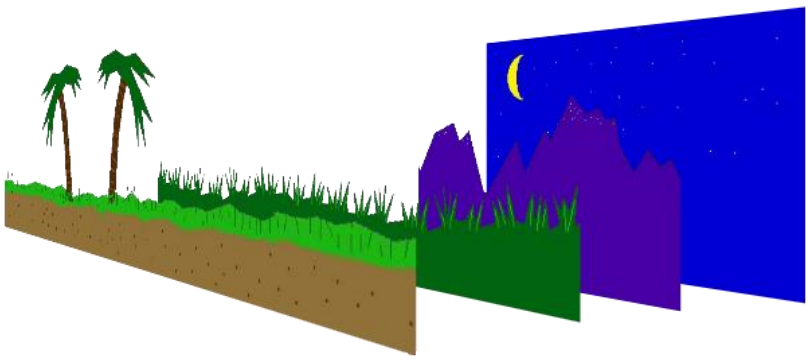

Figura 10. Imágenes parcialmente transparentes utilizadas como capas para un parallax scrolling.

El efecto de profundidad mencionado anteriormente se logra moviendo las capas que se encuentran más cerca de la cámara a mayor velocidad que las capas más distantes.

Si bien existen diversas maneras de implementar parallax scrolling, la técnica por la que optaremos consiste en utilizar sprites para representar las distintas capas del fondo.

La mayoría de las veces se utilizan imágenes cíclicas para simular que la imagen es infinita o de gran tamaño, a medida que ésta es desplazada. Esto puede introducir cierta complejidad a la hora de redibujar el punto de unión entre el fin y el principio de la imagen, ya que se hace necesario dibujar una copia de la imagen a continuación de sí misma cuando ésta no abarca toda la pantalla, como lo indica la *Figura 11.*

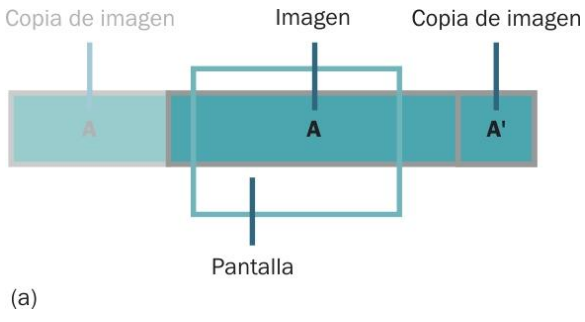

Figura 11. Desplazamiento de una imagen cíclica: (a) cuando la imagen abarca el ancho de la pantalla, puede ser dibujada normalmente; (b) cuando se dibuja el punto de unión de la imagen con sí misma, deben dibujarse distintas partes de la imagen.

El módulo gráfico de SFML está implementado sobre OpenGL, por lo que podemos utilizar comandos de bajo nivel para realizar el desplazamiento cíclico de una imagen de manera sencilla y eficiente, empleando el hardware gráfico. Para ello, basta con aplicar una transformación a las coordenadas de textura de la imagen del sprite.

El siguiente ejemplo ilustra la forma de realizar esto:

```
1. #include <SFML/Window.hpp>
2. #include <SFML/Graphics.hpp>
3.
4. int main(int argc, char *argv[]) {
5.     // cargamos la imagen desde disco
6.     sf::Image i;
7.     i.LoadFromFile("sfml.png");
8.
9.     // creamos el sprite
10.    sf::Sprite s;
11.    s.SetImage(i);
12.
13.    // creamos la ventana y le asignamos una vista
14.    sf::RenderWindow w(sf::VideoMode(800,500),"Despl textura");
15.    sf::View v(sf::FloatRect(0,0,i.GetWidth(), i.GetHeight()));
16.    w.SetView(v);
17.
18.    // variables con la velocidad del desplazamiento
19.    // y el desplazamiento de la capa
20.    float const velx=0.2;
21.    float desplx=0;
22.
23.    // PASO IMPORTANTE 1: habilitamos la repeticion de textura
24.    // o textura ciclica en la coordenada s para la imagen
25.    // del sprite
26.    s.GetImage()->Bind();
27.    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
28.
29.    sf::Event e;
30.    unsigned anchoImagen=s.GetImage()->GetWidth();
31.    while(w.IsOpened()) {
32.        // manejamos los eventos
33.        while(w.GetEvent(e)) {
34.            if(e.Type == e.Closed)
35.                w.Close();
36.        }
37.        // aumentamos la posicion de la capa, y la
38.        // mantenemos dentro de un rango para evitar
39.        // desbordes de la variable
40.        desplx+=velx*w.GetFrameTime();
41.        if(desplx>anchoImagen) desplx-=anchoImagen;
42.
43.        w.Clear(sf::Color(0,0,0));
44.
45.        // PASO IMPORTANTE 2: antes de dibujar, aplicamos
46.        // el desplazamiento a la matriz de textura de la
47.        // imagen del sprite, luego debemos restaurar el
48.        // estado anterior de la matriz
49.        glMatrixMode(GL_TEXTURE);
50.        glPushMatrix();
51.
52.        glLoadIdentity();
53.        glTranslatef(desplx, 0, 0);
54.        w.Draw(s);
55.
56.        glMatrixMode(GL_TEXTURE);
57.        glPopMatrix();
58.        w.Display();
59.    }
60.    return 0;
61. }
```

El código del ejemplo produce el desplazamiento de la imagen hacia la izquierda de manera cíclica. Sin embargo, la posición del sprite nunca cambia. A continuación, explicaremos dos pasos importantes que hay que tener en cuenta para lograr esto.

Primero, se debe habilitar la repetición de textura para la imagen, lo cual es realizado en las líneas 26 y 27. La función *Bind()* de la clase *sf::Image* permite seleccionar como activa la textura para que los comandos siguientes tengan efecto sobre la misma. En

este caso, la función *glTexParameter()* se utiliza para indicar que queremos que la textura se repita horizontalmente.

En segundo lugar, se debe aplicar una transformación a las coordenadas de textura con la que es dibujado el sprite. OpenGL dispone de diversas matrices: de modelo *(GL_MODELVIEW),* de proyección (*GL_PROJECTION*) y de textura, entre otras. Esta última es la encargada de transformar las coordenadas de textura, por lo que, aplicando transformaciones a dicha matriz, podremos alterar las coordenadas de textura con las que se dibuja el sprite. En las líneas 48 y 49 se selecciona esa matriz y se la empuja en la pila para poder restaurarla más tarde. Luego, se limpia, se le aplica el desplazamiento guardado en la variable *desplx* y se dibuja el sprite utilizando dicha transformación. Finalmente, en las líneas 56 y 57, vuelve a seleccionarse la matriz (ya que SFML cambiará la matriz activa para realizar el dibujado) y se recupera su estado anterior.

Como dijimos anteriormente, el sprite nunca es desplazado de su posición original. Cabe destacar, además, que las coordenadas de textura del sprite tampoco cambian, sino que sólo son transformadas de manera previa al dibujado del mismo.

Al utilizar comandos de OpenGL, debemos tener el cuidado de conservar el estado original de las cosas para no alterar el funcionamiento de la librería, en este caso, retornar la matriz de texturas a su estado anterior.

Tal como lo venimos haciendo, propondremos una clase general para facilitar la implementación de parallax scrolling en distintos juegos. La clase permite representar una capa del paralaje y moverla, tanto horizontal como verticalmente. Su prototipo es el siguiente:

```
1. #ifndef PARALLAXLAYER_H
2. #define PARALLAXLAYER_H
3.
4. #include <SFML/Window.hpp>
5. #include <SFML/Graphics.hpp>
6.
7. class ParallaxLayer: private sf::Sprite{
8. private:
9.     // dimensiones de la imagenes
10.    int ancho_img, alto_img;
11.    // velocidad del movimiento de la capa
12.    float factor_x, factor_y;
13.    // posicion actual de la capa
14.    float despl_x, despl_y;
15.    // si es ciclica en x o en y
16.    bool repeat_x, repeat_y;
17.    // desplazamiento respecto a la vista
18.    float offset_x, offset_y;
19.
20. public:
21.    // constructor
22.    ParallaxLayer(    sf::Image &img,
23.                      float factor_x, bool repeat_x=true,
24.                      float offset_x=0, float factor_y=0,
25.                      bool repeat_y=false, float offset_y=0);
26.    // mover la capa segun el tiempo
27.    void Move(float dt);
28.    // mover la capa segun el desplazamiento del personaje
29.    void Move(float dx, float dy);
30.    // dibujar la capa
31.    void Draw(sf::RenderWindow &w);
32.    // para mover las capas junto con la vista
33.    // en caso de usar scrolling
34.    void SetPosition(sf::View &v);
35. };
36.
37. #endif
```

La clase recibe, en el constructor, la mayoría de sus parámetros: la variable *img* es una referencia a la imagen a utilizar para la capa; *repeat_x* y *repeat_y* expresan si debe o no usarse repetición horizontal y vertical en la textura; *offset_x* y *offset_y* indican el desplazamiento de la capa respecto a la vista, ya que no todas las capas tienen

necesariamente el mismo tamaño, y finalmente, *factor_x* y *factor_y* son las velocidades del movimiento de la capa en las dos direcciones.

A continuación, exponemos el código del constructor:

```
ParallaxLayer::ParallaxLayer(    sf::Image &img,
                      float factor_x, bool repeat_x=true,
                      float offset_x=0, float factor_y=0,
                      bool repeat_y=false, float offset_y=0){
    SetImage(img);

    // calculamos el ancho y alto de la imagen, esto
    // nos va a servir para evitar desbordes en el desplazamiento
    ancho_img=img.GetWidth();
    alto_img=img.GetHeight();

    // inicializamos las variables internas
    despl_x=despl_y=0;
    this->factor_x=factor_x;
    this->factor_y=factor_y;
    this->repeat_x=repeat_x;
    this->repeat_y=repeat_y;
    this->offset_x=offset_x;
    this->offset_y=offset_y;

    // seleccionamos la textura y habilitamos la repeticion
    // segun corresponda
    img.Bind();
    if(repeat_x)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    if(repeat_y)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // ajustamos la posicion de la capa
    Sprite::SetPosition(offset_x, offset_y);
}
```

El constructor asigna la imagen y posición al sprite, inicializa otras variables y activa la repetición de textura para la imagen, con un código similar al visto anteriormente. Las variables *despl_x* y *despl_y* contienen el desplazamiento actual de la capa, mientras que el ancho y el alto nos servirán para limitar dicho desplazamiento, como veremos más adelante.

La clase posee funciones que permiten desplazar la capa en función de la posición del personaje o del tiempo transcurrido. Las variables *factor_x* y *factor_y* indicarán la velocidad de las capas respecto a la velocidad del personaje, en el primer caso, o el desplazamiento en función del tiempo, en el segundo, y deberán ser ajustadas de acuerdo a estas situaciones.

A continuación, mostramos el código de dichas funciones:

```
// Permite mover la capa segun el desplazamiento del personaje
void ParallaxLayer::Move(float dx, float dy){
    // para x
    if(repeat_x){
        // si debemos desplazar la textura simplemente
        // incrementamos la variable despl_x y la mantenemos
        // limitada para evitar desbordes
        despl_x+=factor_x*dx;
        if(factor_x>0 && despl_x>ancho_img)
            despl_x-=ancho_img;
        else if(factor_x<0 && despl_x<-ancho_img)
            despl_x+=ancho_img;
    }else{
        // de lo contrario movemos el sprite
        Sprite::Move(factor_x*dx, 0);
    }

    // para y
    if(repeat_y){
```

```
        // si debemos desplazar la textura simplemente
        // incrementamos la variable despl_y y la mantenemos
        // limitada para evitar desbordes
        despl_y+=factor_y*dy;
        if(factor_y>0 && despl_y>alto_img)
            despl_y-=alto_img;
        else if(factor_y<0 && despl_y<-alto_img)
            despl_y+=alto_img;
    }else{
        // de lo contrario movemos el sprite
        Sprite::Move(0, factor_y*dy);
    }
}
```

La función actualiza el desplazamiento de la capa, tanto horizontal como verticalmente. Para cada dirección, en caso de que deba repetirse la textura, se actualiza la variable de desplazamiento para luego utilizarla al dibujar. Si no tiene que repetirse la textura, simplemente se desplaza el sprite en dicha dirección. Esto resulta útil para realizar movimientos pequeños y no cíclicos de la capa.

La función que realiza el movimiento de la capa en función del tiempo utiliza la función anterior de la siguiente forma:

```
// Permite mover la capa dado un lapso de tiempo dt
void ParallaxLayer::Move(float dt){
    Move(dt, dt);
}
```

Finalmente, la función para el dibujado de la capa realiza el mismo procedimiento ilustrado anteriormente, mientras que la función *SetPosition()* ajusta la posición del sprite a la vista actual, utilizando el offset pasado al constructor.

El código de ambas funciones se expone a continuación:

```
// Dibuja la capa
void ParallaxLayer::Draw(sf::RenderWindow &w){
    // selecciona y guarda la matriz de textura
    glMatrixMode(GL_TEXTURE);
    glPushMatrix();
    // aplicamos la transformacion y dibujamos
    glLoadIdentity();
    glTranslatef(despl_x,despl_y,0);
    w.Draw(*this);
    // restauramos la matriz de textura
    glMatrixMode(GL_TEXTURE);
    glPopMatrix();
}
```

```
// Permite ajustar la capa a la vista para usar scrolling
void ParallaxLayer::SetPosition(sf::View &v){
    Sprite::SetPosition( offset_x+v.GetRect().Left,
                         offset_y+v.GetRect().Top);
}
```

Abajo es posible observar el código de un ejemplo de utilización de la clase que ha sido desarrollada para realizar parallax scrolling con las imágenes de la *Figura 12*.

```
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include "ParallaxLayer.h"
int main(int argc, char *argv[]) {
    // creamos la ventana
    sf::RenderWindow w(sf::VideoMode(800,400),"Parallax Scrolling");
```

```
    // nombres de los archivos de capas
    char *archivosCapas[]={ "parallax-1-800x200.png",
                            "parallax-2-800x120.png",
                            "parallax-3-800x200.png",
                            "parallax-4-800x200.png",
                            "parallax-5-1600x400.png"};

    // cargamos las imagenes de las capas
    sf::Image imgCapas[5];
    for(unsigned i=0; i<5; i++)
        imgCapas[i].LoadFromFile(archivosCapas[i]);

    // los offsets y velocidades de las capas
    float offsetYCapas[]={0,120,80,90,0};
    float velCapas[]={0.05, 0.09, 0.15, 0.23, 0.3};

    // inicializamos las capas del parallax
    ParallaxLayer *capas[5];
    for(unsigned i=0; i<5; i++)
        capas[i]=new ParallaxLayer(   imgCapas[i], velCapas[i], true, 0,
                                      0, false, offsetYCapas[i]);

    float elapsedTime; // el tiempo transcurrido
    while(w.IsOpened()) {
        sf::Event e;
        elapsedTime=w.GetFrameTime();
        // atendemos eventos
        while(w.GetEvent(e)) {
            if(e.Type == e.Closed)
                w.Close();
        }
        // limpiamos la pantalla
        w.Clear(sf::Color(0,0,0));
        // movemos las capas y dibujamos
        for(unsigned i=0; i<5; i++) {
            capas[i]->Move(elapsedTime);
            capas[i]->Draw(w);
        }
        // actualizamos la panatalla
        w.Display();
    }
    return 0;
}
```

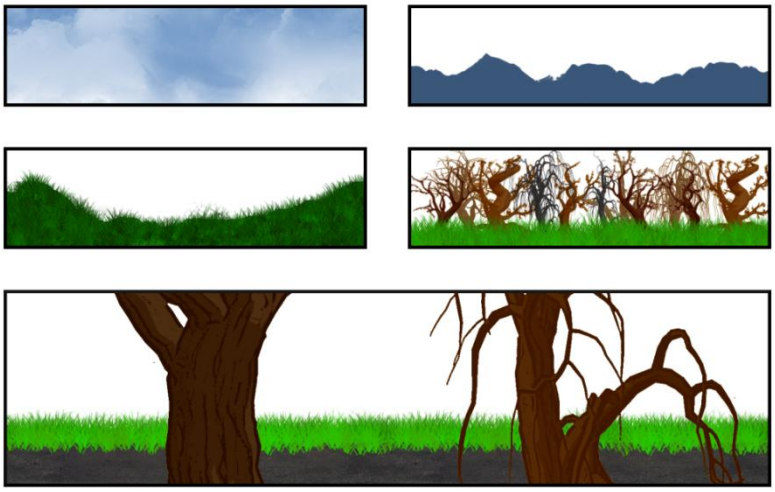

Figura 12. Capas utilizadas en el ejemplo de parallax scrolling.

Para aplicar parallax scrolling a nuestro ejemplo de *Megaman* a fin de que las capas se muevan de acuerdo al desplazamiento de nuestro personaje, agregaremos a la clase *Nivel* un vector de objetos *ParalaxLayer,* al que identificaremos con el nombre *capasParallax*. Además, modificaremos la función *SetViewCenter()* de dicha clase, agregando el código que se muestra a continuación:

```
void Nivel::SetViewCenter(sf::Vector2f newCenter){
    ...
    for(unsigned i=0; i<capasParallax.size(); i++){
        capasParallax[i]->SetPosition(levelView);
         capasParallax[i]->Move(-(oldCenter.x-newCenter.x),
                                oldCenter.y-newCenter.y);
    }
    ...
}
```

De igual forma, modificamos la función *Draw()* para que ésta realice también el dibujado de las capas de fondo:

```
void Nivel::Draw(sf::RenderWindow &w){
    for(unsigned i=0; i<capasParallax.size(); i++){
        capasParallax[i]->Draw(w);
    }
    ...
}
```

Nuevamente, aconsejamos recurrir al ejemplo para observar el código completo.

Bibliografía

Wikipedia [en línea]

http://en.wikipedia.org/wiki/Tile_engine

http://en.wikipedia.org/wiki/Tile-based_video_game

http://en.wikipedia.org/wiki/Parallax_scrolling

Tile Based Games [en línea] http://www.tonypa.pri.ee/tbw/start.html

Shiny Blog [en línea] http://shinylittlething.com/2009/08/08/pygame-parallax-scrolling-in-2d-games/