Programación Gráfica 2D (VII)
Más sobre Tilesets.

Autor: Sergio Hidalgo
serhid@wired-weasel.com

Introducción

En un tutorial anterior expliqué qué eran los tilesets (agrupaciones de varios tiles en una única superficie), y desde entonces he venido asumiendo que teniamos algún tipo de clase "Tileset" a la que le podiamos decir simplemente "dibuja el tile 5" y se encargaba de todo.

En este tutorial voy a comentar como podría funcionar esa clase, y la voy a expandir un poco más añadiendo el concepto de offsets (desplazamientos o anclas). Pero de momento, empecemos por el principio:

Tilesets básicos

Como decía, un tileset básico no es mas que una superficie con varios tiles en ella. En el código fuente del tutorial 5 podeis ver un ejemplo (tiles.bmp). La estructura de esa superficie sería la siguiente:

Como veis, los tiles se ordenan por filas y columnas. En el ejemplo del tutorial 5 el tileset tiene una única fila, pero lo normal es tener varias.

Una restricción importante es que todos los tiles dentro de un mismo tileset tendrán el mismo ancho y alto, lo que simplifica bastante los cálculos. Estas dimensiones pueden ser distintas para cada uno de los distintos tilesets, por lo que mas adelante podremos tener objetos que ocupen varias casillas, por ejemplo.

Otra restricción es el orden en que "colocaremos" los tiles. Iremos rellenando el tileset desde el primero a la izquierda, y por filas. En la imagen se indican en gris oscuro los tiles "ocupados", y en blanco los que no tienen nada. En este ejemplo el tileset tendría 15 tiles ocupados de un máximo de 18. De esa forma, la numeración empezará por cero, y aumentará en el mismo orden. El último tile ocupado sería el nº 14.

La tercera restricción es que el ancho de la imagen será múltiplo del ancho de sus tiles, y lo mismo para el alto. Esto nos asegura que no tendremos ningún espacio muerto. La rejilla ocupará la

totalidad de la imagen.

Con esto, ya podemos esbozar nuestra clase Tileset:

Atributos:

- **superficie:** La propia superficie SDL
- **anchoTile, altoTile:** Las dimensiones del tile
- **nFilas, nColumnas:** El número de filas y columnas en el tileset

Métodos:

- **cargar (archivo, anchoTile, altoTile):** Carga la SDL_Surface, e inicializa nFilas y nColumnas. El ancho y alto del tile se deben conocer de antemano.

- **blit (nTile, destX, destY, superficieDestino):** Blitea el tile de número "nTile" sobre la superficie "superficieDestino", en las coordenadas (destX, destY)

No voy a entrar a implementar toda la clase, ni en detalles de qué tipos debe devolver cada método, o incluso si el método cargar debería ser o no el constructor en realidad. Eso lo dejo a la elección de cada uno. También habría un destructor encargado de liberar la superficie, pero de nuevo, eso son detalles. Voy a poner únicamente lo más importante que hace cada método:

```
cargar (archivo, anchoTile, altoTile) {

   Carga la superficie SDL (ver  tutoriales anteriores)

   this->anchoTile = anchoTile;
   this->altoTile = altoTile;
   nFilas = superficie->h / altoTile;
   nColumnas = superficie->w /  anchoTile;
}
```

```
blit (nTile, destX, destY,  superficieDestino) {
   SDL_Rect rectOrigen;
   SDL_Rect rectDestino;

   //Comprobamos si el tile es valido
   if (nTile < nFilas * nColumnas) {

      //Calculamos el rectangulo de origen
      rectOrigen.x = (nTile % nColumnas) *  anchoTile;
      rectOrigen.y = (nTile / nColumnas) *  altoTile;
      rectOrigen.w = anchoTile;
      rectOrigen.h = altoTile;

      rectDestino.x = destX;
      rectDestino.y = destY;

      SDL_BlitSurface (superficie,  &rectOrigen,
                       superficieDestino, &rectDestino);

   } else
      return ERROR;
}
```

Y ya está, con esto tenemos una clase como la que hemos estado suponiendo que existía en los otros tutoriales. Puede que para muchos juegos mas o menos simples fuera suficiente, pero tiene serias restricciones, y estamos asumiendo un par de cosas que puede que no siempre sean así:

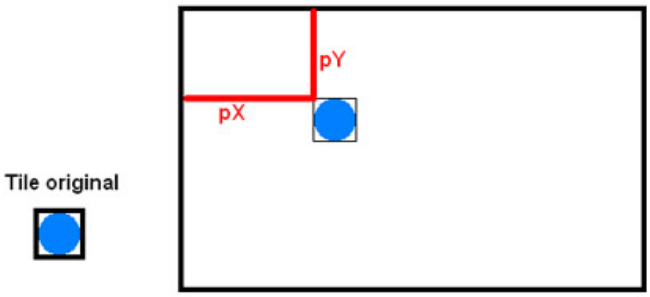
Asumimos que el color transparente es conocido de antemano (pasado como otro parámetro en cargar, o definido como constante, etc...)

Asumimos que las dimensiones de los tiles son conocidas de antemano.
Asumimos que no usaremos desplazamientos (offsets).

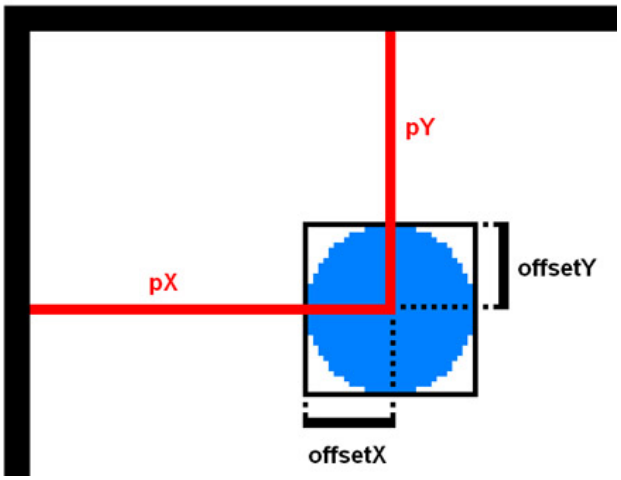
Estas tres limitaciones son las que vamos a intentar eliminar a lo largo del resto del tutorial. Pero antes que nada, vamos a ver que es eso de los desplazamientos (anclas, offsets, o como se les quiera llamar):

Offsets

Imaginad que estamos haciendo un juego estilo “Pong”, y tenemos una pelota. La posicion de la pelota en la pantalla la guardamos en dos variables (px, py), y por otra parte tenemos un tile con el gráfico de la pelota. Cuando dibujamos el tile en la posición de la pelota, ocurre esto:



El tile de la pelota no está centrado en (px, py), sino que ese punto se corresponde con la esquina superior izquierda. Esto es así porque siempre que bliteamos, copiamos un rectángulo, y las coordenadas destino que indicamos (px, py), siempre se van a corresponder a la esquina superior izquierda del rectángulo.

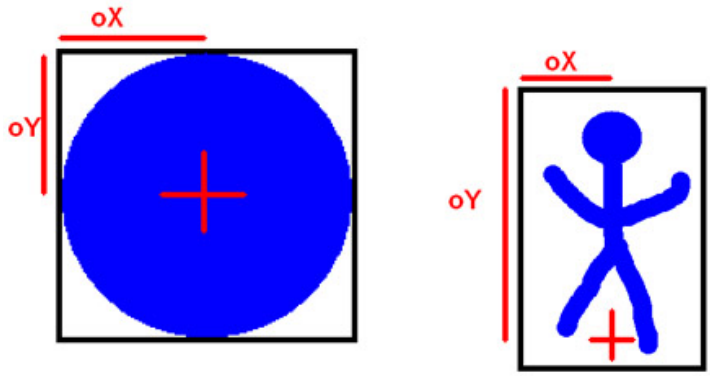


Así que para centrar la pelota en el punto que indicamos, tenemos que desplazar las coordenadas destino del blit hacia la izquierda y hacia arriba. Esas cantidades será a lo que llamaremos “offsets”, o desplazamiento:

En el caso de la pelota, los offsets serán de la mitad del ancho y alto del tile original, para que la imagen quede centrada. Así que en lugar de blitear en (px, py), bliteamos en (px – offsetx, py – offsety).

Pero lo que vale para la pelota, puede que no valga para un tile con otro tipo de objeto. Si ahora queremos dibujar un personaje, es probable que nos sea mas util si el punto (px, py) se corresponde con los pies del personaje, y no con el centro del mismo. Así que estos desplazamientos serán únicos para cada uno de los tiles que tengamos.

El llamarlo “ancla” no es más que otro nombre para lo mismo, aunque puede ser algo mas intuitivo. El concepto es que definimos un punto del tile como “punto de anclaje”, de forma que al blitear, ese punto será el que caiga en la posición (px, py). En el caso de la pelota el ancla estaría en el centro, mientras que en el caso del personaje estaría sobre sus pies.



Tilesets extendidos

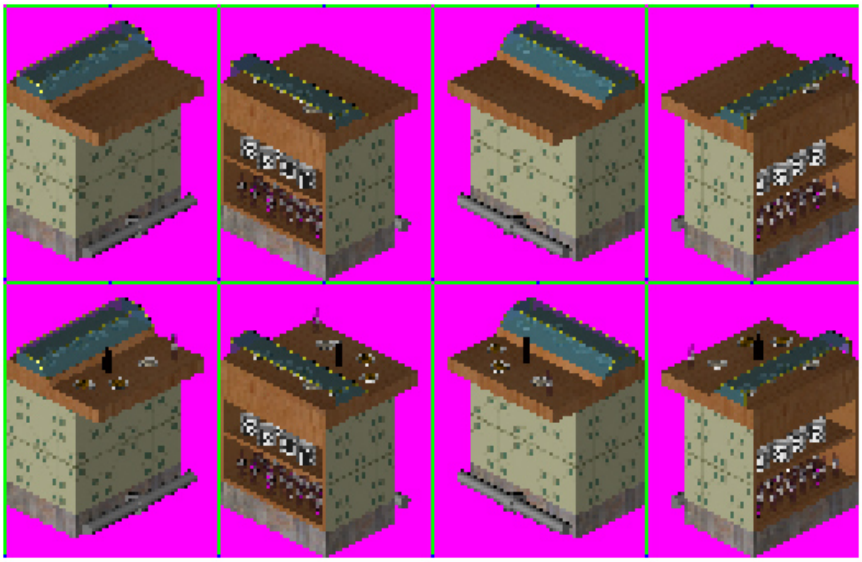
Bueno, ahora que está mas o menos explicado el tema de las anclas, vamos a extender la clase Tileset para añadirlo. Además, aprovecharemos para quitar las dos otras limitaciones. Ahora el tamaño de los tiles, y su color transparente, lo averiguará la propia clase Tileset sin tener que indicárselo como parámetros.

Tenemos dos opciones para hacer esto:

1. Dejamos la imagen como estaba, y creamos un archivo extra con la información de tamaño de los tiles, color transparente, y desplazamientos para cada tile. La clase Tileset leerá la información directamente de ese archivo.

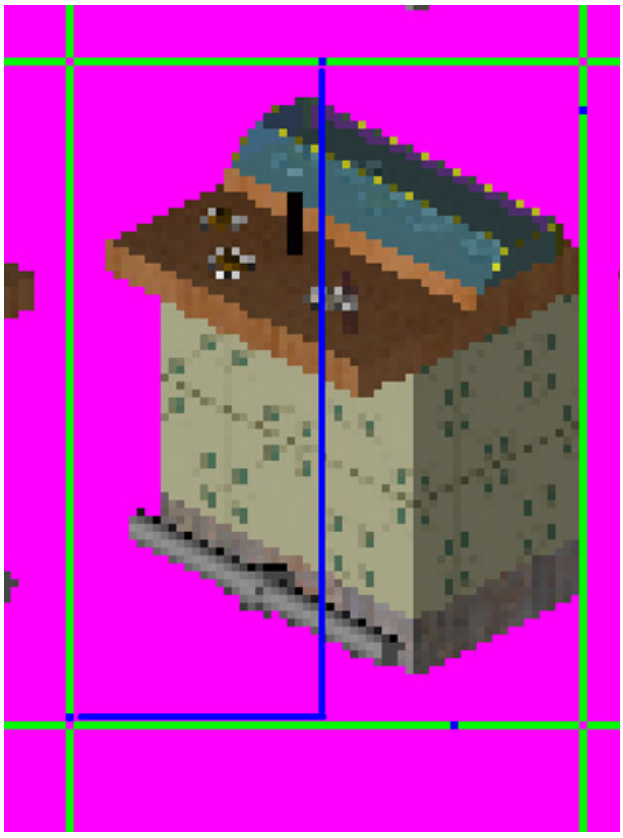
2. Incluimos la información directamente en la propia imagen. La clase Tileset tendrá que obtenerla analizando ciertos pixels de la imagen una vez cargada.

Nosotros optamos por la segunda, que consideramos más cómoda. Para incluir la información, creamos una rejilla de 1 pixel que separará los tiles entre sí. Se ve mejor con una imagen:



La imagen está tomada de uno de los tilesets que hice para el proyecto, un trozo de una barra de bar. La rejilla verde sirve para separar los distintos tiles (que siguen teniendo todos el mismo tamaño, igual que antes). En los puntos donde se cruzan las lineas horizontales y verticales, el pixel rosa indica el color transparente.

Los desplazamientos o anclas se indican mediante unos pixeles azules en las lineas verdes. Aquí se ve algo mejor:



La distancia desde la esquina superior izquierda (punto rosa) hasta el punto de la linea horizontal indica el offsetX. La distancia hasta el punto de la linea vertical indica el offsetY (pensad en el juego de los barquitos). En este caso, el punto de anclaje del tile está abajo y en el centro.

Este no es mas que uno de los posibles sistemas para meter la información de forma visual en la propia imagen. No es ni mejor ni peor que otros, simplemente es el que utilizamos nosotros en nuestro proyecto. Cada uno tiene que ver qué sistema prefiere, o cumple los requisitos para lo que necesita hacer.

No voy a poner el código de la clase Tileset extendida completa, porque no la programé yo, pero si que voy a explicar como funciona:

Nuevo atributo:

Punto *desplazamientos; //Un array de desplazamientos para cada tile. Se inicializa en cargar( )

El método **cargar ( )** ahora solo recibe como parámetro el nombre del archivo. Una vez cargada la imagen SDL, la bloquea, y leyendo los pixels y mirando sus colores rellena los demás atributos que necesita. En concreto:

Color transparente:

El color transparente será el indicado en el pixel (0, 0)

nColumnas:

Recorremos los pixels de y=0, desde (0,0), hasta (ancho, 0). Contamos el número de veces que aparece el color transparente. Ese número será el número de columnas.

nFilas:

Recorremos los pixels de x=0, desde (0,0) hasta (0, alto). Contamos el número de veces que aparece el color transparente. Ese número será el número de filas.

Ancho y alto del tile

Dividimos las dimensiones de la superficie por el número de columnas o el número de filas, respectivamente.

Offsets del tile nº nTile

Nos situamos en el pixel (origenX, origenY):

```
origenX = (nTile % nColumnas) * anchoTile + (nTile % nColumnas);
origenY = (nTile / nColumnas) * altoTile + (nTile / nColumnas);
```

Ese será el pixel rosa justo encima y a la izquierda del tile nTile.
Entonces hacemos:

```
origenX++;
contador = 0;

while (pixel (origenX, origenY) != AZUL) {
   origenX++;
   contador++;
}

desplazamientos[nTile].x = contador;
```

Calcular el offetY es exactamente igual, sólo que sustiyendo origenX++ por origenY++, y al final guardándolo en desplazamientos[nTile].y.

El array de desplazamientos tiene que ser instanciado antes con el tamaño para el número total de tiles que tengamos, claro.

Blit

El método blit también tiene que cambiar. En este caso, además de tener en cuenta los offsets, habrá que sumar los pixeles de la rejilla verde, que no queremos que aparezca al blitear.

```
blit (nTile, destX, destY,  superficieDestino) {
   SDL_Rect rectOrigen;
   SDL_Rect rectDestino;

   if (nTile < nFilas * nColumnas) {

      //Calculamos el rectangulo de origen
      rectOrigen.x = (nTile % nColumnas) *  anchoTile;
      rectOrigen.y = (nTile / nColumnas) *  altoTile;

      //Sumamos los pixels de la rejilla
      rectOrigen.x += (nTile % nColumnas) +  1;
      rectOrigen.y += (nTile / nColumnas) +  1;

      rectOrigen.w = anchoTile;
      rectOrigen.h = altoTile;

      //Restamos los desplazamientos a las  coordenadas del destino
      rectDestino.x = destX -  desplazamientos[nTile].x;
      rectDestino.y = destY -  desplazamientos[nTile].y;

      SDL_BlitSurface (superficie,  &rectOrigen,
                       superficieDestino, &rectDestino);

   } else
      return ERROR;
}
```

NOTA: Es probable que haya errores en el código de este tutorial, porque no he podido compilarlo ni probarlo. Espero que con las explicaciones haya bastante para que podais sacarlo vosotros mismos, pero si encontrais algún error, o algo que no quede claro, decídmelo y lo corregiré cuanto antes.

Rectángulo del tile

Este método que voy a comentar no lo vamos a usar de momento, pero más adelante es probable que nos resulte útil. Consiste simplemente en devolver el rectángulo en la pantalla que ocupa el tile si lo bliteamos en el punto (px, py). Es tan simple que creo que no necesita ninguna explicación.

```
devolverRectangulo  (int nTile, int px, int py) {

   Rectangulo r;

   r.x = px -  desplazamientos[nTile].x;
   r.y = py -  desplazamientos[nTile].y;
   r.w = anchoTile;
   r.h = altoTile;

   return r;
}
```

NOTA: Las clases Punto y Rectangulo que aparecen de vez en cuando en el código no son más que una manera de simplificar un poco el tema. Estoy suponiendo que incluyen atributos x, y (y también w, h para el rectángulo), y que tienen sobrecargados los operadores de suma, asignación, igualdad, etc...

Máscara de opacidad

El último método que voy a comentar de la clase Tileset también es uno que será util en el futuro. Consiste en que, dado un tile número nTile, y un par de coordenadas relativas al rectángulo del mismo (cX, cY), queremos saber si ese pixel es transparente o no lo es.

El método es bastante sencillo. Basta con posicionarse en el pixel indicado:

```
x = (nTile % nColumnas) * anchoTile + (nTile % nColumnas) + cX + 1;
y = (nTile / nColumnas) * altoTile + (nTile / nColumnas) + cY + 1;
```

Bloqueamos la superficie, miramos el color, y si es transparente devolvemos true, si no, false.

Eso está bien, pero es demasiado costoso, porque nos obliga a estar continuamente bloqueando superficies de memoria de video. Otra manera más efectiva de hacerlo es crear una máscara de opacidad en la memoria RAM, y consultar en ella en lugar de hacerlo en la superficie.

Para simplificar, la máscara no será mas que un array de booleanos con el mismo tamaño que la superficie SDL:

bool mascara[ancho][alto];

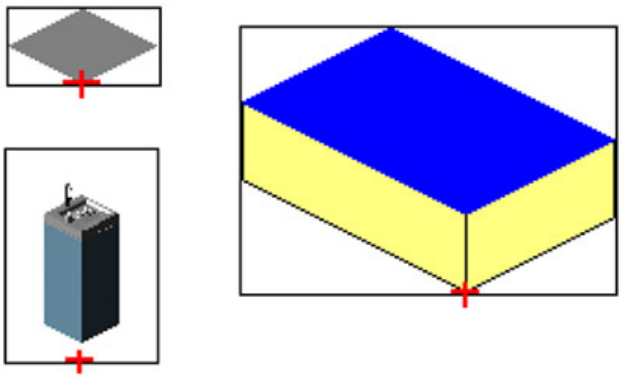
Al cargar la superficie, la recorremos entera, y para cada pixel (x, y), si es transparente hacemos mascara[x][y] = true, en caso contrario false.

Esto incrementa el tiempo de carga, pero asi cuando llamemos al método para saber si el pixel es o no transparente, bastará con devolver el valor de mascara[x][y], sin tener que bloquear nada.

Usando las anclas

Hasta ahora no nos han hecho falta, porque siempre dibujábamos o bien tiles del suelo (todos exactamente iguales), o bien objetos de tamaño 1x1 y de altura fija, como los del final del tutorial 5. Pero en cuanto queramos meter objetos de distintas alturas, que ocupen más de una casilla, o bien personajes y otros elementos, las vamos a necesitar.

Para seguir un convenio, lo que haremos es poner el ancla siempre en el punto más bajo del tile (el pixel del objeto con mayor Y). En el caso de los objetos que ocupen varias casillas, será siempre en la parte más baja de la casilla más baja (la que se dibuja al final).



En esta imagen se ven algunos ejemplos de como colocar el ancla para un tile de suelo típico, un objeto de 1x1, y un objeto de 3x2.

En próximos tutoriales explicaré como dibujar estos objetos grandes, pero de momento basta con saber que podemos tenerlos, y recordar cual es la posición de su punto de anclaje.

Y con esto ya he terminado por hoy. Un saludo!