Programación Gráfica 2D (VIII)
Bichos y tochos.

Autor: Sergio Hidalgo
serhid@wired-weasel.com

Introducción

En el último tutorial hablaba sobre los Tilesets y explicaba conceptos como el del ancla, en parte porque es algo que prácticamente cualquier motor 2D debería implementar, y en parte porque tener esa base nos va a facilitar bastante la vida cuando hablemos de meter personajes y objetos más grandes que una casilla, que es de lo que va a tratar este tutorial.

Y con esto entramos de nuevo en un terreno más o menos nebuloso, porque al contrario de cosas como los Tilesets, el sistema de personajes, o el usar este tipo de objetos grandes es algo que depende mucho del juego en concreto que estemos haciendo, y no es algo que pueda (o deba) ser exactamente igual en todos los motores isométricos.

Así que todo lo que voy a contar aquí es lo que hicimos para nuestro proyecto en concreto, con sus características, limitaciones, etc... concretas. Si estais haciendo un motor para un juego parecido (un Tycoon, por ejemplo) probablemente os sea de utilidad. Si lo que estais haciendo es un arcade al estilo de Alien Shooter, pues no tanto, pero al menos espero que os sirva para coger algunas ideas.

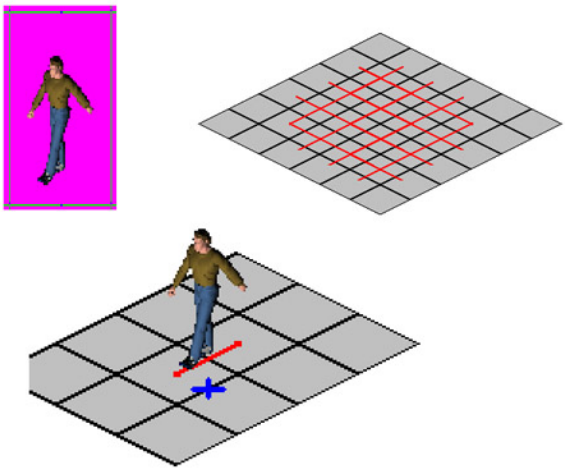
Bichos

Un bicho (personaje) no es más que un ente que se mueve por el mapa. En nuestro caso, vamos a tener dos grandes restricciones:

- Un bicho ocupa una (y solo una) casilla. En el fondo es como los objetos que teníamos hasta ahora, y se puede dibujar igual que los dibujábamos (con un solo tile). Es decir, un objeto de 1x1 que se mueve por el mapa.

- Un bicho no puede estar en cualquier posición. Sólo puede estar, o bien centrado en una casilla, o bien moviéndose del centro de una casilla al de otra adyacente. Esto es una gran diferencia con el movimiento típico de un arcade o muchos RTS, donde las unidades van a donde pulse el ratón, sea o no el centro de una casilla.

En la primera imagen podeis ver el tile de un personaje, en la segunda están marcadas en rojo las posiciones válidas donde podría estar, y en la última un ejemplo del resultado (en ancla está marcado en azul, recordad que los ponemos siempre en el pixel más bajo).



NOTA: En este caso, y por simplificar, sólo salen los movimientos en las diagonales (NO, NE, SE, SO), que son los más importantes. Permitir o no que el personaje se mueva al (N, S, E, O) es a gusto de cada uno.

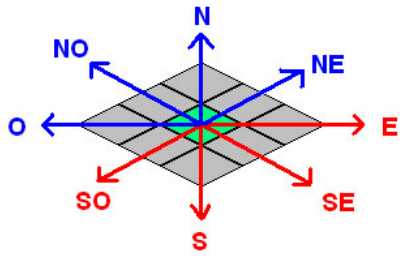
La pregunta ahora es, ¿cómo hacemos para relacionar los personajes con el mapa?, ¿cómo los integramos?. Lo haremos de forma parecida a los objetos: Igual que cada casilla tiene un puntero al objeto que hay en ella, tendrá también un puntero (o una lista de punteros) al personaje/s que está/n en esa posición.
De esa forma un personaje siempre está "enlazado" a una (y sólo una) casilla, y lo dibujaremos cuando toque dibujar esa casilla, igual que dibujamos los objetos que haya.

Dibujando los bichos

Cuando un personaje esté en el centro de la casilla a la que está enlazado, no hay problema, es el caso más fácil. Simplemente dibujaremos el tile del personaje en las mismas coordenadas que el tile del suelo de la casilla. Como las anclas están bien alineadas (ver tutorial anterior), no habrá que tocar nada y habremos terminado.

```
void dibujarCasilla (Punto coordenadasPantalla) {

      //Dibujamos el suelo
      tilesetSuelo->blit (tipoSuelo, coordenadasPantalla);

      //Dibujamos el objeto si lo hay
      if (objeto != NULL)
            objeto->dibujar (coordenadasPantalla);

      //Dibujamos el personaje si lo hay
      if (personaje != NULL)
            personaje->dibujar (coordenadasPantalla);

}
```

El otro caso es que el personaje se esté moviendo de una casilla a otra adyacente. En ese caso, enlazaremos siempre al personaje con la casilla que se dibuje más tarde de las dos. Es decir, si nos movemos al NO o al NE, dejamos en enlace en la casilla original. Si nos movemos hacia el SO o SE, ponemos el enlace en la casilla de destino:



Verde: Casilla Original
Azul: Enlazamos en la casilla original
Rojo: Enlazamos en la casilla destino

De esta forma nos aseguramos de respetar siempre el orden de profundidad (el algoritmo del pintor), el personaje tapará a la casilla con menor valor de coordenada Y absoluta (y a lo que haya en ella).

Lo único que nos queda es el desplazamiento propiamente dicho. Este desplazamiento será una cantidad de pixels dX y dY que sumaremos a las coordenadas de pantalla de la casilla a la que estemos enlazado.
En cierta forma, tenemos dos desplazamientos, el del ancla, que es fijo, y que lo calcula el Tileset automáticamente, y este nuevo (dX, dY) que va variando cada fotograma, y lo calculamos manualmente.

```
void Personaje::dibujar (Punto coordPantalla) {
      tilesetPJ->blit (nTile, coordPantalla.x + dX,
                       coordPantalla.y + dY);
}
```

Dado que el desplazamiento siempre será relativo a la casilla a la que esté enlazado el personaje, el valor será positivo o negativo dependiendo de la dirección del movimiento. En concreto, si queremos movernos una unidad (1 pixel, por ejemplo), tendremos que cambiar (dX, dY) tal y como indica esta tabla:

DIRECCION	dX	dY
ISOD_NINGUNA	0	0
ISOD_N	0	-1
ISOD_NE	+2	-1
ISOD_E	+2	0
ISOD_SE	+2	+1
ISOD_S	0	+1
ISOD_SO	-2	+1
ISOD_O	-2	0
ISOD_NO	-2	-1

NOTA: Suponiendo una relación de aspecto 2:1

Hay que tener en cuenta que al empezar el movimiento, y de nuevo dependiendo de la dirección, deberemos inicializar (dX, dY) a los valores correctos. Cuando lleguemos a los valores finales (que también dependen de la dirección) habremos llegado a la casilla destino, y el movimiento habrá terminado. En ese caso podremos enlazar el personaje a la nueva casilla (si no lo estaba ya), y hacer (dX, dY) = (0, 0).

Los valores iniciales y finales de (dX, dY) para cada movimiento están en la siguiente tabla:

DIRECCION	dXini	dYini	dXfin	dYfin
ISOD_NINGUNA	-	-	-	-
ISOD_N	0	0	0	-TAL
ISOD_NE	0	0	TAN/2	-TAL/2
ISOD_E	-TAN	0	0	0
ISOD_SE	-TAN/2	-TAL/2	0	0
ISOD_S	0	-TAL	0	0
ISOD_SO	TAN/2	-TAL/2	0	0
ISOD_O	0	0	-TAN	0
ISOD_NO	0	0	-TAN/2	-TAL/2

NOTA: TAN = TILE_ANCHO, TAL = TILE_ALTO

Implementar el movimiento por tanto es bastante mecánico. Al empezar, enlazamos con la casilla correspondiente, y hacemos (dX, dY) = (dXini, dYini). Después, cada fotograma actualizamos (dX, dY) según la primera tabla, hasta que sea igual a (dXfin, dYfin), que habremos terminado.

De hecho, el incrementar (o decrementar) (dX, dY) puede hacerse en más (o menos) que un pixel. Bastaría multiplicar las cantidades de la primera tabla por la “velocidad” que más nos guste, el truco está en mantener siempre la misma proporción que aparece en la tabla.

Como digo, la implementación es algo mecánico, así que solo voy a poner un poco de pseudocódigo:

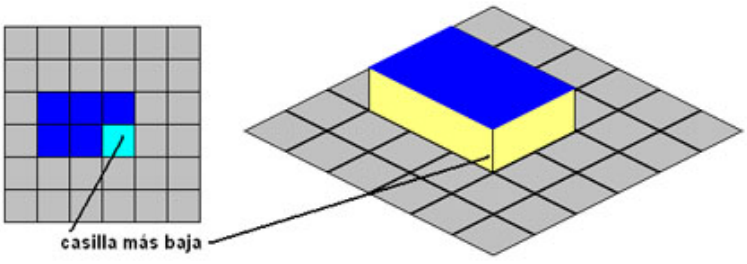
```
iniciarMovimiento (direccion) {

      //Estamos ya enlazados a la casilla “origen”, dependiendo del
      //movimiento, cambiaremos ese enlace o no.

      if (direccion es SO, S, SE, E) {
            casillaDestino = tileWalk (casillaOrigen, direccion);
            enlazar (casillaDestino);
      }

      //Hacemos (dX, dY) = (dXini, dYini)
      switch (direccion) {
      case N:
            (dX, dY) = (0, 0);
            break;
      case S:
            (dX, dY) = (0, -TAL);
            break;
      [...]
```

```
	}
}

actualizarMovimiento (direccion) {

	switch (direccion) {
	case N:
		//Incrementamos
		(dX, dY) += (0, -1);

		//Comprobamos si estamos en (dXfin, dYfin)
		//y ya hemos terminado
		if ((dX, dY) == (0, -TAL)){
			enlazar (casillaDestino);
			(dX, dY) = (0,0);
			return;
		}
		break;

	case S:
		(dX, dY) += (0, 1);

		if ((dX, dY) == (0,0)) {
			enlazar (casillaDestino);
			(dX, dY) = (0,0);
			return;
		}
		break;
	[...]
	}
}
```

Desde luego se puede optimizar mucho más, pero creo que así sirve para ver la idea.

Tochos

Con un “tocho”, me refiero a un objeto que ocupa más de una casilla, como el de esta imagen:



La razón de añadirlos es porque normalmente en nuestro juego querremos tener objetos que sean más grandes que el bloque básico que comentábamos en otros tutoriales. Por ejemplo, queremos tener mesas de reuniones, o mostradores de recepción, o altares satánicos, etc...

Todo eso se podría hacer también diviendo el objeto grande en varios pequeños, cada uno del tamaño de una casilla. De esta forma tendríamos un grupo de objetos como los de hasta ahora, que al colocarlos en las posiciones correctas, como un puzzle, formarían el objeto grande. Ahora se me viene a la cabeza el X-COM como un juego donde se sacaba partido a esto, haciendo que se pudiera destruir cada una de las “partes” de un objeto grande del escenario.

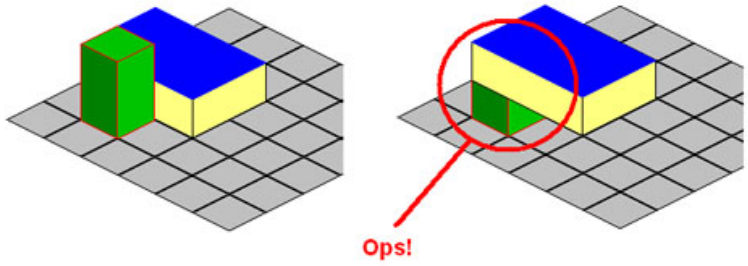
Pero por lo general estaremos más interesados en tratarlo como un único objeto, y por ejemplo si permitimos al jugador que pueda manipularlo moviéndolo, rotándolo, etc... el tratarlo como un montón de ellos sólo nos va a traer complicaciones de cabeza.

Pero si tratamos los tochos como un único objeto, aunque simplificamos su gestión como elementos del juego, los problemas nos los vamos a encontrar a la hora de dibujarlos, porque nuestro motor se basa en casillas, y en dibujar cada una de las casillas de forma independiente. Entonces, ¿en qué momento es cuando tenemos que dibujar el tocho?

La primera idea será el de dibujarlo al llegar a la casilla más baja:

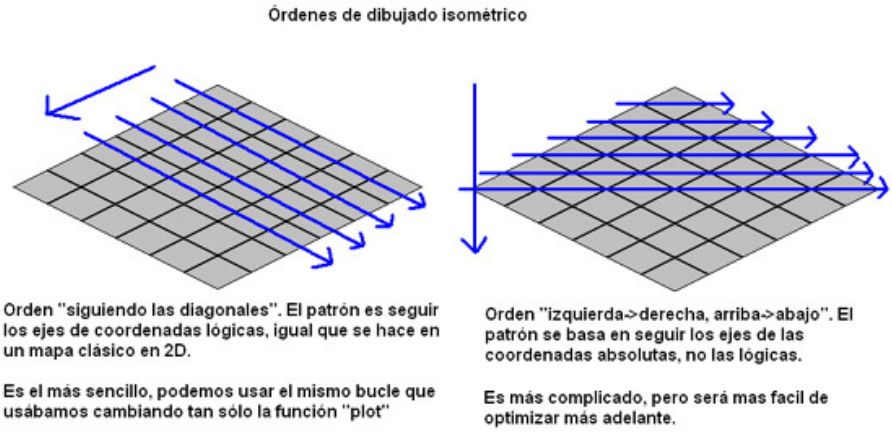
Empezamos por arriba, siguiendo el orden de siempre, y cuando estamos en las casillas del objeto, no lo dibujamos hasta que no sea la última, la más baja. Al llegar a ella, dibujamos todo el tocho de golpe.

El problema de este método se ve en la siguiente imagen:



A la izquierda se ve el resultado correcto. A la derecha el que obtendremos siguiendo el método, ya que al dibujar el tocho tapamos el otro objeto que habíamos dibujado antes, aunque este debería aparecer por encima.

Aquí es donde vamos a pagar el precio de haber elegido este orden para dibujar las casillas. Si recordais el tutorial 5:



Había dos modos posibles de ordenarlas. Nosotros elegimos el segundo, que nos facilitaba el optimizar para no tener que dibujar todo el mapa. Pero si hubiésemos dibujado según las diagonales, ahora podríamos alterar ligeramente el algoritmo para añadir estos objetos sin tener el

problema del solapamiento. No voy a entrar en los detalles, que podreís encontrar si rebuscáis por aquí: http://www.gamedev.net/community/forums/forum.asp?forum_id=13

Trozeando tochos

Así que si no podemos dibujar los tochos de una vez al llegar a la casilla más baja, tendremos que trozearlos, y dibujar un trocito en cada casilla correspondiente.

OJO: No estoy hablando de dividirlos en varios objetos, como hablaba arriba, sino en seguir tratándolo como un único objeto, sólo que lo dibujamos casilla a casilla.

De esta forma, ya no vamos a tener como antes:

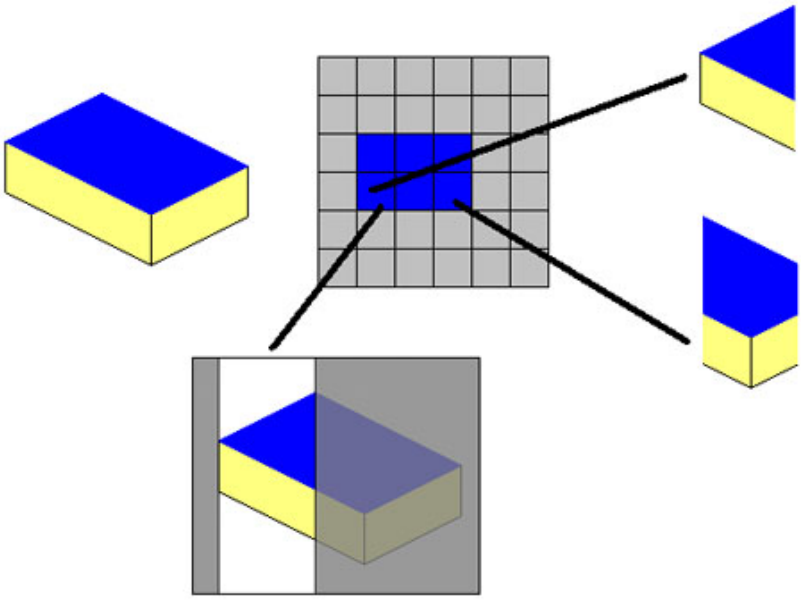
```
objeto->dibujar (coordenadasPantalla);
```

Sino que ahora añadiremos como parámetro las coordenadas lógicas de la casilla. Dependiendo de esas coordenadas, dibujamos la parte del objeto que sea:

```
objeto->dibujar (coordenadasPantalla, coordenadasCasilla);
```

Ahora volvemos a tener dos opciones: O bien guardamos cada trocito como un tile independiente, y a la hora de dibujar simplemente bliteamos el tile que corresponde a la casilla; o bien usamos un único tile para todo el objeto, y lo que hacemos es dibujar sólo el cachito que sea el de la casilla.

Nosotros elegimos la segunda opción, porque la primera nos complicaba demasiado la creación de los gráficos.



Como veis en la imagen, para cada casilla del objeto corresponde un cachito del tile. En realidad, una “columna” centrada en la casilla, que será tan ancha como TILE_ANCHO.

Antes que nada, vamos a poner algunos nombres y coordenadas, que si no esto empieza a ser lioso muy rápido:

- Los objetos ahora se definen por un rectángulo (x, y, w, h), que guarda las coordenadas de la casilla del mapa de arriba a la izquierda, su ancho, y su alto.

- En un alarde de imaginación, voy a llamar “casilla del ancla” a la casilla de abajo a la derecha, porque es la que tiene el ancla original. Sería la (x + w - 1, y + h - 1).

- La casilla concreta del objeto que estemos dibujando en cada momento la vamos a llamar “casilla actual”.

- Todas las casillas que ocupa el objeto tienen un puntero al mismo. Así que en todas ellas vamos a llamar a objeto->dibujar () pasándole las coordenadas lógicas de la casilla actual.

- Y por último, vamos a llamar “región” al trozo del tile que corresponde a una casilla, como en la imagen de arriba.

¿Bueno, empezamos?

La primera manera que se nos ocurre para dibujar esto, es usar un clipper (recordais, hace eones, que os comenté el tema de los clippers?). La idea aquí sería en dibujar siempre el objeto completo, pero usando un clipper que nos limite a la región que corresponda a la casilla actual.
Sería exactamente lo que se ve en el último dibujo de la imagen de arriba, una “columna” que abarcaría toda la pantalla de alto, pero tan solo TILE_ANCHO de ancho.

Así que para una casilla actual (Xi, Yi) con coordenadas de pantalla (pXi, pYi), el clipper sería el rectángulo:

```
clipper.x = pXi - (TILE_ANCHO / 2);
clipper.y = 0;
clipper.w = TILE_ANCHO;
clipper.h = PANTALLA_ALTO;
```

NOTA: Fijaos como le restamos la mitad del ancho del tile en la coordenada X, esto es porque nuestras anclas estan en el medio del tile, en lugar de a la izquierda.

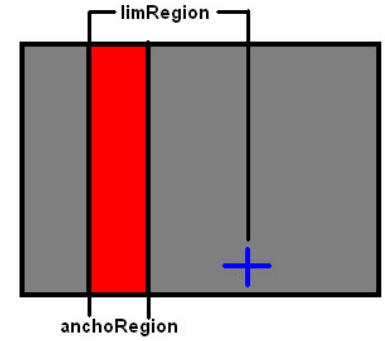
NOTA: Y por el amor de Dios, acordaos de restaurar el clipper original cuando termineis de blitear los objetos!

El clipper está bien, funciona, pero no es elegante. A fin de cuentas estamos bliteando todo el objeto una y otra vez, y cambiando el clipper continuamente. Lo elegante sería blitear únicamente la región actual, y no tener que tocar nada mas.

Para eso, primero toca modificar nuestra vieja clase Tileset, añadiendo un nuevo método que blitee un tile, pero no entero, solo la región que indiquemos:

```
blitRegion (nTile, destX, destY, limRegion, anchoRegion, 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;

        //Sumamos los pixels de la region
        rectOrigen.x += limRegion + desplazamientos[nTile].x;

        rectOrigen.w = anchoRegion;
        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;
}
```



ROJO = Region AZUL = Ancla

Como veis en la imagen, la región siempre va a ser una columna tan alta como el tile, y los únicos parámetros que indicamos son su ancho, y su límite izquierdo (en pixels, y relativo al punto del ancla).

La razón de hacerlo relativo al punto del ancla es porque usaremos la casilla del ancla como punto de referencia, y el ancla del tileset siempre estará en esa casilla. Así, para calcular el limRegion de la casilla actual tomamos la diferencia con la casilla del ancla, y hacemos lo siguiente:

```
difX = casillaAncla.x - casillaActual.x;
difY = casillaAncla.y - casillaActual.y;
limRegion = (difY - difX - 1) * TILE_ANCHO / 2;
```

Vale, vale, tiempo... ¿y esto de dónde sale?

Si os fijais, es algo parecido a lo que hacíamos para plotear una casilla, sólo que tomando la casillaAncla como origen de coordenadas (por lo que el Y y el X se invierten, ya que vamos hacia arriba).

Si no lo veis, haced memoria (tutorial 5):

```
plotX = (x - y) * (TILE_ANCHO / 2);
```

Esto era lo que hacíamos para plotear una casilla normalmente. En este caso, el origen de coordenadas estaba arriba en el centro. En nuestro caso, al tener el origen abajo, tenemos que cambiar el orden de x e y. El -1 que aparece es el mismo que salía al hacerlo con el clipper, tenemos que desplazarnos TILE_ANCHO / 2 pixels a la izquierda adicionales para compensar por la posición del ancla.

Asi que el código final queda:

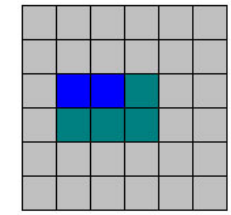
```
Objeto::dibujar (Punto casillaActual) {

      Punto cPantalla = plot (casillaAncla);

      difX = casillaAncla.x - casillaActual.x;
      difY = casillaAncla.y - casillaActual.y;
      limRegion = (difY - difX - 1) * TILE_ANCHO / 2;

      tileset->blitRegion (nTile, cPantalla.x, cPantalla.y, limRegion,
                           TILE_ANCHO, superficieDestino);

}
```

NOTA: Fijaos como las coordenadas de pantalla en las que bliteamos son siempre las de la casilla del ancla. Estas coordenadas podemos calcularlas una y otra vez haciendo un plot (como aquí), o calcularlas una única vez y reutilizarlas para cada una de las casillas del mismo objeto.

Optimizaciones

Ahora que tenemos un sistema que funciona, podemos empezar a optimizarlo y hacerlo más eficiente. La primera optimización sería la de evitar tener que dibujar varias regiones iguales.

¿A qué me refiero? Bueno, si os fijais, como la región es tan alta como el propio tile del objeto, no hace falta dibujar las casillas “de arriba”, porque tendrán la misma región que la casilla que tengan abajo. De esta forma, solo habría que dibujar las casillas de los bordes exteriores.



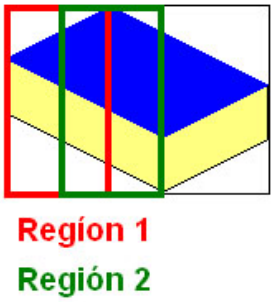
Dibujar No Dibujar

Con esta optimización, de un objeto de NxM casillas nos estamos ahorrando de dibujar (N-1)(M-1). Para los fanáticos de los costes, hemos reducido el coste de nuestro algoritmo de $O(N^2)$ a $O(N)$, lo que no está mal.

Para ello, podemos, o bien no poner puntero en las casillas que no dibujamos, o bien meter esta comprobación en Objeto::dibujar:

```
if ((casillaActual.x == casillaAncla.x) ||
      (casillaActual.y == casillaAncla.y)) {

      //Código de antes

}
```

La segunda optimización es no dibujar regiones que se tapen unas a otras. Como el ancho de la región es TILE_ANCHO, pero dibujamos una cada TILE_ANCHO / 2, dos casillas consecutivas tendrán regiones que se tapan parcialmente la una a la otra.



De esta forma, podemos dibujar una casilla sí, y otra no, y aún así cubrir todo el objeto con el menor número posible de blits.

Lo que está claro es que las regiones más a la izquierda y más a la derecha hay que dibujarlas. Estas corresponden a las casillas de coordenadas: (casillaAncla.x - ancho + 1, casillaAncla.y), y (casillaAncla.x, casillaAncla.y - alto + 1), respectivamente.

Luego todo lo que hay que hacer es medir la distancia de casillaActual con respecto a estas casillas límite, y dibujarla únicamente si es un número par:

```
if (((casillaActual.x == casillaAncla.x) &&
      (((casillaActual.x - (casillaAncla.x - ancho + 1)) % 2) == 0)) ||
      ((casillaActual.y == casillaAncla.y) &&
      (((casillaActual.y - (casillaAncla.y - alto + 1)) % 2) == 0))) {

      Punto cPantalla = plot (casillaAncla);

      difX = casillaAncla.x - casillaActual.x;
      difY = casillaAncla.y - casillaActual.y;
      limRegion = (difY - difX - 1) * TILE_ANCHO / 2;

      tileset->blitRegion (nTile, cPantalla.x, cPantalla.y, limRegion,
                           TILE_ANCHO, superficieDestino);

}
```

NOTA: Como de costumbre, este código no está probado ni compilado, por lo que probablemente contenga algún bug. En este caso, yo estaría atento a contar el número de paréntesis xD

Y con esto termino por hoy. Como ya os he dicho al principio, estos métodos para dibujar personajes y tochos no son para nada los únicos, simplemente los que nos inventamos nosotros y metimos en nuestro juego. Si estais haciendo otro juego distinto, tendreis otras necesidades, y posiblemente os tengais que buscar otras soluciones.

En cualquier caso, para estas alturas ya contais con bastante para hacer un motor isométrico bastante decentillo. En un futuro tutorial comentaré más sobre cuál podría ser su arquitectura general, y que tipo de clases usaríamos. Aunque de nuevo, esto es algo que depende mucho de lo que vaya a hacer con él cada uno.

Un saludo!