Programación Gráfica 2D (IX)
Picking y Dirty Rectangles.

Autor: Sergio Hidalgo
serhid@wired-weasel.com

Introducción

Al igual que contaba en el tutorial anterior, en éste voy a explicar un par de técnicas que no son "obligatorias" para todo juego isométrico, sino que cada uno tendrá que decidir en función de lo que necesita si le merecen o no la pena de implementar.

Pero en este caso, si que creo que se trata de algo un poco más general que el tema de los personajes que comentaba en el último. En este tutorial vamos a hablar de cómo hacer para detectar sobre qué objeto o personaje ha hecho click el jugador (con una precisión a nivel de pixel), y de cómo podemos optimizar aún más el dibujado.

Estas dos técnicas van a usar un par de métodos de la clase TileSet que expliqué en el tutorial 7:

- **devolverRectangulo**, que dado un punto que se corresponde al ancla, nos devuelve el rectángulo que ocupará el tile si lo bliteamos en ese lugar

- **checkTransparente**, que dado un punto nos dice si el pixel correspondiente del tile es o no transparente (en este caso el punto no es relativo al ancla, sino a la esquina del rectángulo del tile).

Por eso, y porque estas técnicas también se van a aplicar sobre los personajes que aparecían en el último tutorial, he decidido esperar un poco antes de explicarlas las dos juntas. Si no recordáis como funcionaban estos métodos de TileSet, os recomiendo que lo repaseis un momento antes de seguir.

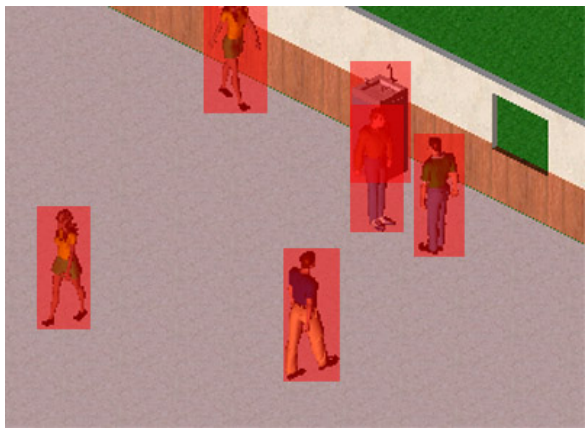
Picking

"Picking" es una palabra que se suele usar en programación 3D, cuando tenemos las coordenadas de un pixel de pantalla, y tenemos que averiguar a qué polígono (o elemento) corresponden. Esto se usa mucho en el caso de que el jugador pueda hacer click en pantalla con el ratón y seleccionar estos objetos. Así que a falta de una palabra mejor, me quedo con "picking" para lo que vamos a hacer nosotros en 2d.

En nuestro caso, vamos a tener un mapa isométrico con un montón de objetos y personajes sobre los que el jugador puede pulsar. Cuando el jugador hace click, queremos averiguar qué elemento (a partir de ahora los llamaremos a todos "elementos") ha pulsado, para realizar sobre él la acción que sea. Además, también queremos detectar si no ha pulsado sobre ningún elemento.

En el caso de un juego 2D clásico es bastante sencillo. Sabemos la posición de cada elemento y el rectángulo que ocupa en pantalla, así que solo tenemos que recorrerlos todos y ver si el pixel está dentro de ese rectángulo.

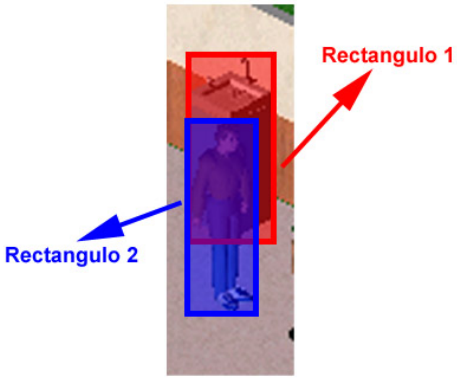
En juegos isométricos la cosa se complica, porque tenemos elementos que pueden taparse unos a otros, y estar a distintas profundidades. Así que hay un poco más de trabajo para averiguar cuál de ellos es el que debe ser seleccionado.



Antes de empezar, una advertencia. No nos importa para nada lo que haya en la SDL_Surface de la pantalla, pero sí que nos importa lo que ve el jugador. Me explico: tenemos que asegurarnos de que la representación interna que manejamos nosotros (posición de elementos) se corresponde exactamente con lo que el jugador estaba viendo en pantalla en el instante que hizo el "click".

Esto viene a decir que el proceso de Picking habrá que hacerlo antes de cualquier movimiento de objetos, personajes, scroll, etc... para trabajar con los datos tal y como se los mostramos al jugador al terminar el último fotograma. Si no lo hacemos así y primero movemos los elementos, el jugador tendrá más dificultad a la hora de seleccionarlos, porque habrá una diferencia de varios pixels entre lo que él ve en pantalla y los datos con los que trabajamos nosotros.

Y bueno, dicho esto, vamos a ir viendo como solucionar el tema de los rectángulos que se tapan entre sí:



El rectangulo 1 esta por detras del 2, por lo que esta a mayor profundidad (fue dibujado antes)

En este caso, tenemos que el objeto que se dibuja más tarde (por encima) está "menos profundo", así que tendría prioridad a la hora de ser seleccionado. El proceso sería algo así:

- Si el punto pulsado está fuera del rectángulo 2, lo descartamos, y comprobamos el siguiente elemento (rectángulo 1).
- Si el punto pulsado está dentro del rectángulo 2:
  - Si es un pixel transparente, lo descartamos, y comprobamos el siguiente elemento (rect. 1)
  - Si es un pixel no transparente, el elemento pulsado ha sido el 2, y terminamos.

Extendiéndolo a más de dos elementos:

- Tomamos el elemento que fue el último en dibujarse
- if (punto dentro del rectangulo)
  - if (punto no transparente)
    - return elemento (fin, hemos encontrado elemento)
- Si no, descartamos el elemento, tomamos el siguiente en la lista, y volvemos a empezar
- Si nos quedamos sin elementos, entonces ninguno de ellos fue pulsado

Para saber en qué orden se dibujaron los elementos, usaremos una estructura como una lista o una pila, en la que iremos introduciendo los datos según los dibujamos (en el fotograma anterior), y luego los extraemos en el orden inverso.

En esa pila, podríamos guardar una estructura con estos datos:

```
struct datosPicking{
      Elemento *elem;   //Puntero para identificar al elemento
      Rectangulo rect;  //Rectangulo en pixels que ocupa en pantalla
};
```

Para rellenar la pila, cada vez que dibujemos un objeto (o personaje, o cualquier elemento que queramos que sea "pulsable" hacemos lo siguiente:

```
objeto->dibujar (casillaActual);
datosPicking.elem = objeto;
datosPicking.rect = objeto->devolverRectangulo (casillaActual);
pila.push (datosPicking);
```

NOTA: Estoy suponiendo que "devolverRectangulo" usa el método de tileset para calcular el rectángulo que ocupa en pantalla, como he comentado arriba. La Pila se puede implementar también como una lista enlazada, o usar una de las estructuras ya preparadas de las STL.

El pseudo-código para detectar el elemento pulsado se quedaría entonces así:

```
Elemento* picking (Punto posCursor) {

      //Mientras queden elementos por comprobar
      while (!pilaVacia) {
            //Extraemos el último en dibujarse
            pila.pop (datos);

            //Si se pulsó en su rectangulo
            if (puntoEnRectangulo (posCursor, datos.rect)) {
                  //Si el pixel no es transparente
                  if (!datos.elem->checkTransparente (Punto(posCursor.x -
                        datos.rect.left, posCursor.y - datos.rect.top))) {

                        //Hemos encontrado el elemento, lo devolvemos
                        return datos.elem;

                  }
            }

      }

      //Hemos comprobado todos y no ha sido pulsado ninguno
      return NULL;
}
```

No creo que haga falta decirlo, pero por supuesto, una vez hemos terminado de usar la pila para el picking, hay que vaciarla antes de dibujar el siguiente fotograma y volverla a usar. Siempre tenemos que tenerla limpia cuando empezamos a dibujar.

Y ya está. Al final es bastante sencillo, y si hemos implementado el método "checkTransparente" usando una máscara de opacidad, funcionará también bastante rápido.

Y un último detalle: este método está pensado para detectar pulsaciones sobre elementos como objetos o personajes, y no sobre las propias casillas del mapa. Si queremos saber sobre qué casilla está el cursor, seguiremos usando el MouseMapping de toda la vida.

Dirty Rectangles

La otra técnica de la que voy a hablar es un método de optimización bastante clásico, que se ha venido utilizando en juegos 2D desde el principio de los tiempos. Viene simplemente de seguir la regla de oro de la programación gráfica: *"No dibujarás nada que no sea necesario"*.

Si nos fijamos en la mayoría de juegos de este estilo, veremos que por regla general la mayor parte de la pantalla muestra un mapa (prácticamente estático), y sólo unos pocos elementos pequeños se mueven por él y están animados. La idea de esta técnica consiste en dibujar únicamente esos elementos que cambian, y evitar tener que tocar el resto de la pantalla que no ha variado desde el fotograma anterior.

Actualmente en los PCs modernos no es tan necesario implementar algo así, porque las CPUs y las tarjetas gráficas tienen la potencia suficiente para ir rápido aunque estemos dibujando toda la pantalla desde cero en cada fotograma. Pero si alguno está pensando en hacer un juego isométrico para otras plataformas menos potentes (como un teléfono móvil, por ejemplo), puede que le sea muy útil.

NOTA: Como en este tipo de plataformas es muy posible que no haya SDL, voy a explicar esta técnica de la forma más general que pueda. En cualquier caso, es independiente del lenguaje que usemos y de la librería gráfica.

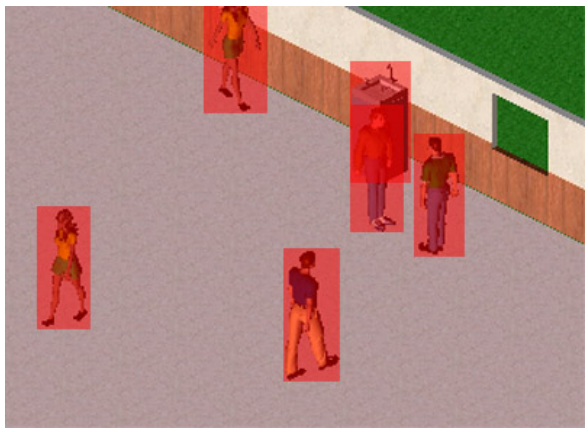
Un caso en el que esta técnica puede ser poco útil es en los juegos con muchos elementos animados y/o móviles (por ejemplo, juegos de acción). En estos casos habrá cambios en prácticamente toda la pantalla, así que la ganancia de usar Dirty Rectangles será muy limitada. Por el contrario, en juegos de estrategia, donde suele haber poco movimiento, resulta bastante más útil. Y si el juego es por turnos, no digamos.

Otro caso típico donde los Dirty Rectangles solían fallar es en el tema del scroll. Cuando la pantalla se desplaza, nos vemos forzados a redibujarla queramos o no, por lo que en juegos clásicos 2D la penalización no se evitaba.
En nuestro caso es diferente. Aunque nos veamos obligados a dibujar la pantalla cuando se desplaza, podemos hacerlo simplemente "copiándola", no tenemos que hacer el plotting de todas las casillas una a una, así que aun en el caso del scroll, seguimos teniendo un beneficio respecto a no usar esta técnica.

Barriéndole el polvo a esos rectángulos

La técnica consiste en marcar una serie de rectángulos como sucios (dirty). Esos rectángulos se corresponden con el área de la pantalla que ha cambiado desde el último fotograma y necesita ser redibujada de nuevo. En la fase de dibujado, recorremos la lista de rectángulos y los dibujamos. Después, los mandamos a la pantalla.

Si recordais la imagen del principio de tutorial, vereis que los rectángulos sucios se corresponden casi con los que considerábamos para el picking. Digo casi, porque en este caso la fuente no habría que redibujarla, y los rectángulos de los personajes no serían exactamente esos, pero sirve para hacerse una idea:



1. Así que en primer lugar, en la fase de actualizado de objetos y personajes rellenamos la lista de rectángulos con los que sea necesario redibujar (si un elemento no está animado ni se mueve, no lo añadimos). La lista no es ninguna estructura compleja, símplemente una lista de rectángulos normal y corriente.

2. Después, a la hora de dibujar, recorremos la lista, y dibujamos sólo esas áreas. Aquí nos viene bien el haber sido previsores: si recordais el código del tutorial 6, ya lo tenemos preparado para dibujar regiones pequeñas en lugar de la pantalla completa. Simplemente llamamos al método “dibujar” para cada uno de estos rectángulos.

NOTA: También sería buena idea utilizar el clipper para asegurarnos de que no nos salimos en ningún momento del rectángulo que estamos dibujando (con el riesgo de machacar la parte “estática” de la imagen).

3. Finalmente, indicamos a la tarjeta que actualice en pantalla los rectángulos que hemos cambiado. En SDL, esto lo podemos hacer llamando a SDL_UpdateRects ( ).

Las fases dos y tres no tienen mucha complicación. Lo más complejo de entrada puede ser el marcar los rectángulos “dirty”. Hay que distinguir entre los posibles casos:

- Elemento estático y no animado (ej: una estatua):
  No se añade.

- Elemento estático y animado (ej: una hoguera):
  Añadimos el rectángulo que ocupa su tile.

- Elemento móvil (ej: un personaje):
  Calculamos el rectángulo que ocupaba su tile en el fotograma anterior.
  Movemos el personaje.
  Calculamos el nuevo rectángulo que ocupa su tile.
  Calculamos la unión de ambos rectángulos (el rectángulo más pequeño que incluye a ambos), y lo añadimos a la lista.

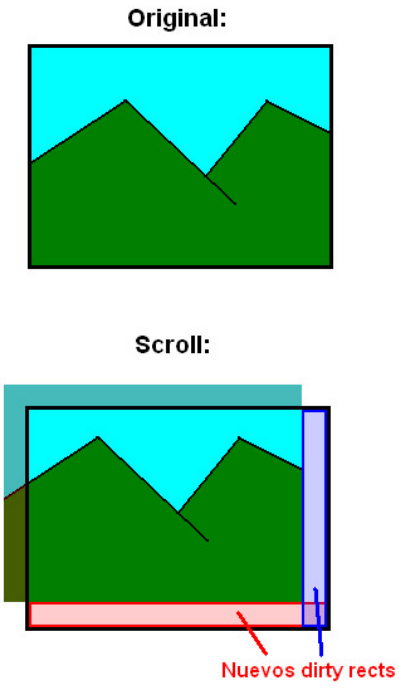
El pseudo-códido sería algo así:

```
void doFrame () {

      //Actualizamos el mundo del juego y generamos la lista de rectángulos
      dirtyRects = update ();

      while (dirtyRects no es vacia) {
            r = extraer primer elemento de dirty rects
            dibujar (r);      //Dibujamos cada dirty rectangle
      }

      UpdateRects (dirtyRects);     //Actualizamos la pantalla
}
```

Scrolling (otra vez)

Aquí tenemos de nuevo al caso raro: ¿qué pasa si el jugador también está haciendo scroll?

En principio, como la pantalla se mueve, habría que actualizarla entera. Pero sabemos que aunque la imagen se desplace, eso no significa que haya cambiado su contenido. Podemos salir del apuro "desplazando" la imagen del último fotograma (mediante un blit), y marcando como dirty el rectángulo del nuevo borde que aparece al hacer el desplazamiento.



En el caso de tener un scroll en dos direcciones (como la imagen) habría que añadir dos rectángulos. En este caso no hace falta calcular su unión, porque nos daría de resultado toda la pantalla :P

Esto sería lo primero que calcularíamos (incluso antes de hacer el "Update" de los personajes y demás, para que sus rectángulos ya vengan ajustados a la nueva posición de la cámara).

El único problema que tenemos que tener en cuenta es cuando calculamos los rectángulos de los elementos móviles. Como tenemos que tener su posición en el último fotograma, hay que tener en cuenta que la cámara se ha movido. Así que si lo que hacíamos era guardar en una variable cual fue su última posición, habrá que desplazarla también para ajustarla de nuevo.

Aparte de esto, nos estamos ahorrando un montón de plots y de casillas que dibujar una a una, asi que merece la pena el complicarnos un poco más.

NOTA: Si hacemos scroll, a la hora de llamar a SDL_UpdateRects ( ) tendremos que actualizar la pantalla completa, y no sólo los rectángulos que hemos redibujado, porque en realidad estamos modificando toda la superficie.

¿Dibujar dos veces lo mismo?, ¿para qué?

Bueno, con esto bastaría para un sistema de dirty rectangles bastante decente. Y si además sabemos que vamos a tener pocos elementos activos en la pantalla, podemos darnos por satisfechos y dejarlo así.

Pero el problema de este sistema es que, aunque en el caso ideal de tener pocos elementos funciona perfecto y reduce enormemente el tiempo de dibujado, si tenemos muchos elementos que se tapan unos a otros, vamos a recibir una penalización.

¿Por qué es eso?. Bueno, si tenemos dos rectángulos de forma que uno tape una zona del otro (como en la fuente y el personaje de la imagen, o el caso del scroll en dos direcciones), la zona que cubren ambos tendrá que ser dibujada dos veces. Esto significa plotear dos veces las mismas casillas, con todos los cálculos que requiere eso.

Y además, este caso es bastante más común en juegos isométricos por el tema de la profundidad, así que es bastante probable que si tenemos muchos elementos, tengamos también muchos rectángulos tapándose entre sí. En el peor de los casos podríamos acabar dibujando tantas o más casillas como cuando no usábamos dirty rectangles.

Por suerte la solución es sencilla: Creamos un array de booleanos tan grande como el mapa, y al principio de cada fotograma lo inicializamos a "false". Después, cuando dibujemos una casilla:

- Si está a "false" -> La dibujamos, y ponemos la marca a "true".
- Si está a "true" -> Pasamos a la siguiente casilla.

En realidad, no sería necesario un array tan grande como el mapa (además, que a lo largo de todos los tutoriales hemos estado intentando hacer que nuestro motor funcione de forma eficiente independientemente del tamaño del mapa, y no vamos a cambiar ahora).
Sería bastante con hacer un array tan grande como el máximo de casillas que caben en la pantalla, y usar algún tipo de función Hash para asignar cada coordenada a una posición en el array.

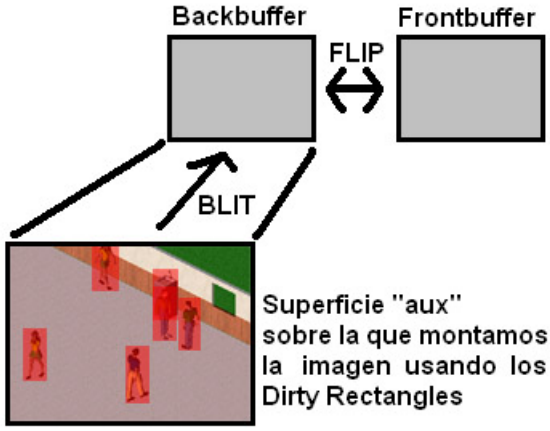
Implementar eso sí que puede ser algo más farragoso, así que no voy a entrar en más detalle, lo dejo como ejercicio al lector :D

Dirty Rectangles con Double Buffering

Si no estamos usando dos buffers para reducir el parpadeo, el método que he comentado antes de usar SDL_UpdateRects ( ) para actualizar sólo las regiones sucias funciona perfectamente. Pero si el double buffering está activo, entonces tenemos un problema:

La imagen que tenemos en el backbuffer (la superficie en la que montamos la nueva imagen para mostrarla) no guarda la pantalla del último fotograma, sino la del anterior a ese, es decir, la de hace dos fotogramas. Para nosotros, lo mismo da que estuviese llena de basura, no nos sirve de nada.

Como la técnica de los dirty rects se basa en disponer de la última imagen, necesitaremos crear una superficie auxiliar del mismo tamaño que la pantalla y montar en ella los fotogramas. Cuando los tengamos preparados, la copiaremos íntegramente al backbuffer (screen) para mostrarla. En este caso, seguiremos haciendo SDL_Flip ( ) como hacíamos antes.



De esta forma, la superficie “aux” guardará siempre el estado de la pantalla en el último fotograma, al que de otra manera no tendríamos acceso. Los dirty rectangles se implementan entonces sobre esta superficie, en lugar de hacerlo directamente sobre la pantalla.

Dirty Rectangles con GUIs y otros elementos

Hasta ahora no habíamos comentado esto, pero ya tenemos un motor bastante completo y es probable que queramos añadirle un GUI (ventanas, menús, etc...) y otros elementos gráficos que no son parte directamente del motor isométrico, sino que se dibujan encima de la imagen que este genera.

El problema de esto es que si usamos dirty rects, estos elementos pueden afectar a la pantalla y tendríamos que tenerlos en cuenta a la hora de calcular las zonas “sucias”, redibujarlas, etc...

En el caso de usar double buffering, como antes, el problema se resuelve por sí mismo. Lo único que tenemos hacer es seguir usando “aux” exclusívamente para el motor isométrico, y montar el GUI y los demás elementos directamente sobre la pantalla una vez hemos copiado los contenidos de “aux”. De esta forma, no interfieren para nada con los dirty rectangles.

Pero si no tenemos dos buffers, y no hay superficie “aux” que valga, entonces al dibujar estos elementos estamos efectivamente machacando la imagen del motor isométrico que necesitábamos para montar el siguiente fotograma.

En realidad, el problema no es tan grave, porque siempre dibujamos las ventanas y estos elementos “por encima”, por lo que las zonas perdidas nunca son visibles, y no nos importan. Lo único que tenemos que recordar es que si el jugador cierra una ventana tenemos que marcar todo el área que ocupaba como sucia, para redibujar el mapa que había por debajo. Lo mismo si permitimos que las ventanas se puedan mover arrastrándolas.

El puntero del ratón (si lo estamos dibujando nosotros como un sprite más) suele ser otro tema problemático, porque se suele mover en todos los fotogramas, a diferencia de una ventana, y nos obliga a marcar continuamente rectángulos “dirty” para refrescar el área que ocupaba en cada momento.

Todo esto se puede optimizar mucho, claro. Por ejemplo, se puede diseñar un sistema GUI que implemente ya de entrada los dirty rectangles, de forma que sólo redibuje una ventana cuando ésta cambia, y no cada fotograma. También se puede mejorar el tratamiento del puntero para que guarde en una pequeña superficie el fondo que tenía por debajo y lo restaure automáticamente al moverse (eso es lo que hace el GUI de Windows, por ejemplo), sin tener que volver a plotear casillas y reconstruirlo todo, etc...

En fin, todo esto se mete ya en el tema de como implementar un buen GUI, y se sale un poco del objetivo del tutorial, así que lo dejo aquí, solo mencionando el tema por encima.

Y esto es todo. Sólo volver a recordar que no es necesario tener un sistema de Dirty Rectangles para que un motor isométrico funcione rápido (por ejemplo, nosotros no lo usábamos en el proyecto, y nunca tuvimos problemas de rendimiento con el motor), pero sí que es una buena técnica de optimización en según que casos.

Un saludo!