Fundamentos de programación gráfica
Práctica 12 - Motor isométrico

Llegamos al final de la asignatura introduciendo algunos conceptos de coordenadas espaciales y entornos 3D, implementando soporte para escenas isométricas en nuestro motor.

Para incluir el soporte de proyección isométrica en el motor, necesitamos modificar tres partes:

- Los sprites deben de ser capaces de manejar tres coordenadas **XYZ** de escena y convertirlas en coordenadas **XY** de pantalla. Además, se deben hacer algunas modificaciones a la gestión de colisiones del sprite.
- Los mapas deben adaptarse de igual forma a la proyección isométrica, convirtiendo las coordenadas **XY** isométricas de los tiles del suelo (la coordenada **z** siempre será 0 en el suelo) en coordenadas de pantalla. Igualmente, vamos a cargar una segunda capa de tiles del mapa para utilizarla como obstáculos en el escenario.
- La escena debe ordenar correctamente los sprites a dibujar para conseguir el efecto de profundidad.

Para poder implementar esto, vamos a crear subclases de `Sprite`, `Map` y `MapScene`, que serán respectivamente `IsometricSprite`, `IsometricMap` e `IsometricScene`.

IsometricSprite

La interfaz de un sprite isométrico es la siguiente:

```
class IsometricSprite : public Sprite {
public:
    IsometricSprite(Image* image);

    virtual double GetScreenX() const;
    virtual double GetScreenY() const;

    virtual void SetCollision(CollisionMode mode);
```

```
    virtual void Update(double elapsed, const Map* map = NULL);
protected:
    virtual void UpdateCollisionBox();
private:
    double screenX, screenY;
};
```

Las variables `screenX` y `screenY` sirven para guardar los resultados de la transformación de las coordenadas de escena en coordenadas de pantalla.

Con respecto a sus métodos, deben hacer lo siguiente:

- Constructor: Simplemente inicializamos las variables miembro añadidas.
- `GetScreenX`, `GetScreenY`: Devuelven el valor de las variables miembro correspondientes.
- `SetCollision`: Las colisiones a nivel de píxel no tienen sentido cuando se están evaluando proyecciones sobre el suelo. Este método llama a la implementación padre pasándole el tipo de colisión especificada excepto si es colisión por píxeles, en cuyo caso le indicará colisión por rectángulos.
- `Update`: Llamamos al método padre para actualizar el sprite, y a continuación utilizamos la función `TransformIsoCoords` para convertir las coordenadas XYZ del sprite en coordenadas de pantalla.
- `UpdateCollisionBox`: Las colisiones de los objetos en un entorno isométrico se comprueban entre sus proyecciones sobre el plano del suelo. La altura de la imagen es descartada, y se usa el ancho para calcular ambos ejes de la proyección (es decir, su proyección siempre será un cuadrado). Se debe redefinir el método en base al siguiente pseudocódigo:

```
x = isox - imagexhandle * fabs(scalex)
y = isoy - imagexhandle * fabs(scalex)
w = imagewidth * fabs(scalex)
h = imageheight * fabs(scalex)
Sprite::UpdateCollisionBox(x, y, w, h)
```

Se debe implementar la función `TransformIsoCoords` de módulo `Math`, con el algoritmo explicado en la teoría (consultar diapositivas del tema).

IsometricMap

Esta clase tiene la siguiente interfaz:

```
class IsometricMap : public Map {
public:
    IsometricMap(const String& filename, uint16 firstColId = 0);
    virtual void GenerateLayerSprites(IsometricScene* scene);
    virtual uint16 GetTileWidth() const;
    virtual void Render() const;
```

```
    virtual int32 GetLayerId(uint16 column, uint16 row);
private:
    Array<int32> topLayerIds;
};
```

El array `topLayerIds` se utiliza igual que el array de tiles de la clase padre `Map`, pero en este caso guarda información de una segunda capa de tiles.

Sus métodos se comportan de la siguiente forma:

- Constructor: Debe cargar la segunda capa de tiles y añadir sus identificadores al array de la misma forma que lo hacíamos con la clase Map. Se debe restar el valor del atributo **"firstgid"** a cada identificador del tile igual que en la clase padre. Además, hay que ajustar el handle de la imagen generada en la clase padre de la siguiente forma:
  ```
  handlex = handlex + tilewidth
  handley = imageheight - handley - tileheight
  ```
- `GenerateLayerSprites`: La segunda capa de tiles cargada se va a generar como sprites en el escenario. Para cada identificador de tile mayor que -1 en el vector, se debe generar un sprite con la imagen del tileset, asignar el frame correcto (el id del tile), establecer modo de colisión por rectángulos si el identificador de tile es colisionable, y poner sus coordenadas de la escena (la fila y columna del tile por el ancho y el alto respectivamente de un tile). La clase `IsometricScene` se encargará de llamar a este método para que el mapa añada a la escena todos estos tiles.
- `GetTileWidth`: Como estamos simulando la isometría mediante una proyección dimétrica, los tiles se representa con el doble de píxels en horizontal de su tamaño real. Debemos dividir entre dos el resultado devuelto por el método padre para obtener una escala uniforme en X y en Y.
- `Render`: Esta implementación es igual a la de la clase padre, pero antes de dibujar la imagen del tile en sus coordenadas, se debe utilizar la función `TransformIsoCoords` para convertirlas en coordenadas de pantalla.
- `GetLayerId`: Devuelve el identificador del tile especificado de la segunda capa.

Debemos descomentar la carga de mapas isométricos en la clase `ResourceManager`.

IsometricScene

Si interfaz es la siguiente:

```
class IsometricScene : public MapScene {
public:
      IsometricScene(IsometricMap* map, Image* imageBack = NULL, Image* imageFront =
NULL);
    virtual IsometricSprite* CreateSprite(Image* image, Layer layer = LAYER_BACK);
    virtual void Update(double elapsed, Map* map = NULL);
protected:
    static bool CompareSprites(Sprite*& first, Sprite*& second);
```

```
};
```

Explicamos cada uno de sus métodos:

- El constructor pasa los parámetros al constructor base (haciendo casting del mapa al tipo `Map*`), y llama al método `GenerateLayerSprites` del mapa para que genere los sprites en la escena.
- `CreateSprite` deriva de la clase `Scene`, y se debe redefinir para que devuelva objetos de la clase `IsometricSprite` en lugar de `Sprite`. Se deben añadir en la lista de sprites de la escena llamando al método padre `AddSprite`.
- `Update` llama al método padre y a continuación ordena todos los sprites para que sean pintados en el orden correcto. Con el método `GetSprites(layer)` obtenemos el array de sprites para cada una de las capas de la escena, y utilizando el método `Sort` sobre él ordenaremos los sprites. La función de ordenación se pasa como parámetro, y debe ser `CompareSprites`.
- `CompareSprites` es utilizada por los arrays de sprites para ver si están ordenados o no. Recibe como parámetro punteros a dos objetos `Sprite` de la lista, y simplemente debemos decir si ya están ordenados o no. Estarán ordenados si la coordenada Y de pantalla del primer objeto es menor a la del segundo.

Prueba

Una vez ampliada la funcionalidad del motor, realizaremos un ejercicio de pruebas. Vamos generar una escena isométrica que tenga como mapa **“data/isometric.tmx”**.

Incluiremos en la escena un sprite isométrico con la imagen **“data/isoplayer.png”**, que contiene 8x8 frames. Se colocará en la posición (mapTileWidth*1.5, mapTileHeight*1.5). La cámara debe seguir a este sprite y no tendrá límites.

En función de la dirección en la que esté mirando el sprite, el primer frame para la animación de correr será:

- Izquierda: 0
- Arriba: 24
- Derecha: 40
- Abajo: 56

En cualquier caso, la animación de correr tiene siempre 4 frames. La dirección en la que mira el sprite lo marca la tecla pulsada.

Cuando pulsamos una tecla de cursor, el sprite debe avanzar una casilla completa en la dirección correcta. Se deben detectar colisiones con el escenario para evitar que atraviese los árboles del bosque.

El resultado de la práctica debe quedar como en la siguiente imagen: