Programación 2D

Práctica 4: Colisiones

Clase Collider

La base de la implementación de detección de colisiones en el motor se realizará mediante la clase abstracta Collider, que cuenta con los siguientes métodos:

- virtual bool collides(const Collider& other) const = 0;
- virtual bool collides(const Vec2& circlePos, float circleRadius) const = 0;
- virtual bool collides(const Vec2& rectPos, const Vec2& rectSize) const = 0;
- virtual bool collides(const Vec2& pixelsPos, const Vec2& pixelsSize, const uint8_t* pixels) const = 0;

Implementaremos tres **subclases** concretas, cada una para un tipo de primitiva: CircleCollider, RectCollider y PixelsCollider, que implementan cada uno de estos métodos de colisión con cualquiera de los otros (añadiremos las variables miembro que sean necesarias).

Como vemos, para realizar la colisión a nivel de píxel, necesitamos el buffer con los píxeles de imagen. Este buffer no lo tenemos, ya que una vez se genera la textura en memoria de vídeo, se borran sus píxeles de la RAM. Esto lo podemos remediar si cada vez que cambiamos la imagen del sprite, obtenemos con **ltex_getpixels** los píxeles de la textura y los guardamos en una variable miembro de Sprite para poderlos pasar a la detección de colisiones.

Para no repetir código en distintos métodos (por ejemplo, la implementación de CircleCollider::collides(const Vec2& rectPos, const Vec2& rectSize) y la de RectCollider::collides(const Vec2& circlePos, float circleRadius) será prácticamente igual), vamos a implementar funciones que comprueben colisiones entre cada para de tipos de primitivas, y en los métodos de las clases **utilizaremos las funciones adecuadas**:

- bool **checkCircleCircle**(const Vec2& pos1, float radius1, const Vec2& pos2, float radius2);
- bool **checkCircleRect**(const Vec2& circlePos, float circleRadius, const Vec2& rectPos, const Vec2& rectSize);
- bool **checkRectRect**(const Vec2& rectPos1, const Vec2& rectSize1, const Vec2& rectPos2, const Vec2& rectSize2);

- bool **checkCirclePixels**(const Vec2& circlePos, float circleRadius, const Vec2& pixelsPos, const Vec2& pixelsSize, const uint8_t* pixels);
- bool **checkPixelsPixels**(const Vec2& pixelsPos1, const Vec2& pixelsSize1, const uint8_t* pixels1, const Vec2& pixelsPos2, const Vec2& pixelsSize2, const uint8_t* pixels2);
- bool **checkPixelsRect**(const Vec2& pixelsPos, const Vec2& pixelsSize, const uint8_t* pixels, const Vec2& rectPos, const Vec2& rectSize);

Cuando se llama al método que recibe otra primitiva como parámetro, el objeto debe llamar al método collides de la otra primitiva pasándole sus propios datos geométricos, de forma que la otra primitiva cuente con toda la información necesaria para detectar la colisión.

Clase Sprite

Vamos a hacer añadidos a la clase para que los sprites soporten detección de colisiones. En primer lugar, vamos a añadir un enumerado con los tipos de colisión soportados por el motor:

enum **CollisionType** {

COLLISION_NONE,

COLLISION_CIRCLE,

COLLISION_RECT,

COLLISION_PIXELS

};

Añadiremos los siguientes métodos a la clase y las variables miembro que consideremos necesarias, así como un constructor si lo necesitamos:

- void **setCollisionType** (CollisionType type);
- CollisionType **getCollisionType** () const;
- const Collider* **getCollider** () const;
- bool **collides** (const Sprite& other) const;

En el método setCollisionType, en función del tipo elegido, crearemos un Collider apropiado para el objeto y lo guardaremos como variable miembro (borrando previamente el Collider anterior que hubiese):

- COLLISION_NONE: Ninguno
- COLLISION_CIRCLE: Un CircleCollider cuya posición será la del centro del sprite, y su radio será la mitad del ancho o alto de la imagen (el mayor de los dos, por ejemplo).
- COLLISION_RECT: Un RectCollider cuya posición y tamaño lo tendremos que calcular teniendo en cuenta la posición del sprite, su pivote, escala, y el tamaño de la imagen.
- COLLISION_PIXELS: Un PixelsCollider, cuya posición dependerá de la posición del sprite, el pivote y el tamaño de la imagen. El tamaño será el tamaño de la imagen.

En el método Sprite::**collides**, en el caso de que ambos sprites tengan un collider, invocaremos sobre uno de los colliders el método Collider::collides pasándole el otro como parámetro. El método ya se encarga de obtener todos los datos y calcular la colisión.

NOTAS:

- Los datos de los Collider de los sprites deben estar actualizados siempre con los datos del sprite.
- No es necesario tener en cuenta la rotación de los sprites en el cálculo de las colisiones.
- No es necesario tener en cuenta la escala de los sprites en el caso de los PixelCollider.

Programa principal

Utilizaremos las siguientes imágenes:

				
data/ball.png	data/bee.png	data/box.png	data/circle.png	data/rect.png

Las cargaremos como texturas, y con las tres primeras crearemos tres sprites que aparecerán en el centro de la pantalla, ordenados de izquierda a derecha. Estos tres sprites utilizarán, respectivamente, los modos de colisión COLLISION_CIRCLE, COLLISION_PIXELS, COLLISION_RECT.

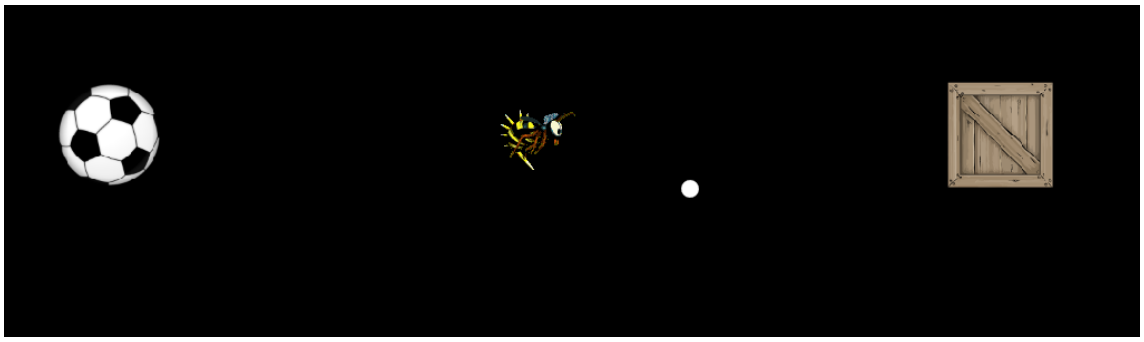
Los sprites de los lados (los que no usan colisión por píxel) deben oscilar su escala continuamente entre 0.9 y 1.1 puntos, a una tasa de 0.25 puntos por segundo.

Crearemos un cuarto sprite con la imagen "data/circle.png", que utilizará colisión por círculos, y estará siempre en las coordenadas del ratón. Su imagen y tipo de colisión se podrá modificar con los botones del ratón:

Botón	**Imagen**	**Collisión**
`GLFW_MOUSE_BUTTON_LEFT`	data/circle.png	`COLLISION_CIRCLE`
`GLFW_MOUSE_BUTTON_RIGHT`	data/rect.png	`COLLISION_RECT`
`GLFW_MOUSE_BUTTON_MIDDLE`	data/bee.png	`COLLISION_PIXELS`

Utilizaremos la función collides de los sprites en el bucle principal para ver si alguno colisiona con el sprite del puntero. Si lo hace, pintaremos ambos de rojo. En caso contrario, los pintaremos de blanco.

NOTA: La parte de colisión por píxeles cuenta el 50% del total de la calificación.



Se pueden usarlas imágenes que se quiera siempre que tengan ese tipo de forma.

- Añadir otra imagen con pixeles distintos que funcione con el comportamiento esperado.

Opcional: intentar añadir algún típico de colisión de primitiva más compleja.