Fundamentos de programación gráfica
Práctica 10 - Colisiones

La base de la implementación de detección de colisiones en el motor se ha realizado mediante la clase abstracta `Collision`, que cuenta con los siguientes métodos:

- `bool DoesCollide(Collision* other)`: Colisión con otra primitiva.
- `bool DoesCollide(double cx, double cy, double cradius)`: Colisión con un circulo.
- `bool DoesCollide(double rx, double ry, double rwidth, double rheight)`: Colisión con un rectangulo.
- `bool DoesCollide(CollisionPixelData* pixels, double px, double py)`: Colisión con píxeles.

Existen tres subclases concretas, cada uno para un tipo de primitiva: `CircleCollision`, `PixelCollision` y `RectCollision`, que implementan cada uno de estos métodos.

Las colisiones entre cada par de primitivas se realizan mediante la clase `CollisionManager`, que utiliza el patrón singleton, con la siguiente interfaz:

```
class CollisionManager {
public:
	static const CollisionManager& Instance();
	virtual bool CircleToCircle(double x1, double y1, double r1, double x2, double y2, double r2) const;
	virtual bool CircleToPixels(double cx, double cy, double cr, const CollisionPixelData* pixels, double px, double py) const;
	virtual bool CircleToRect(double cx, double cy, double cr, double rx, double ry, double rw, double rh) const;
	virtual bool PixelsToPixels(const CollisionPixelData* p1, double x1, double y1, const CollisionPixelData* p2, double x2, double y2) const;
	virtual bool PixelsToRect(const CollisionPixelData* pixels, double px, double py, double rx, double ry, double rw, double rh) const;
```

```
	virtual bool RectToRect(double x1, double y1, double w1, double h1, double x2,
double y2, double w2, double h2) const;
protected:
	CollisionManager() {}
	virtual ~CollisionManager() {}
private:
	static CollisionManager* manager;
};
```

Las tres clases concretas mencionadas deben de llamar a los métodos de esta clase para detectar colisiones entre los datos de su geometría y los datos de geometría pasados como parámetro. Cuando se llama al método que recibe otra primitiva como parámetro, el objeto debe llamar al método `DoesCollide` de la otra primitiva pasándole sus datos geométricos, de forma que la otra primitiva cuente con toda la información necesaria para detectar la colisión (técnica del **doble despacho**).

La clase Sprite cuenta con un enumerado con los distintos tipos de colisión para el sprite. Sus valores son `COLLISION_NONE`, `COLLISION_CIRCLE`, `COLLISION_PIXEL` y `COLLISION_RECT`. Mediante el método `SetCollision` de la clase `Sprite`, se indica el tipo de colisión, y el sprite genera la primitiva de colisión adecuada y la almacena en su variable miembro `collision`:

- Cuando el tipo de colisión es `COLLISION_NONE`, se debe eliminar el objeto almacenado en `collision` y cambiar su valor a `NULL`.
- Cuando el tipo de colisión es `COLLISION_CIRCLE`, se debe eliminar el objeto almacenado en `collision` y crear un objeto `CircleCollision`. Sus valores geométricos los dan las coordenadas del sprite y el valor del radio establecido mediante el método `SetRadius`.
- Cuando el tipo de colisión es `COLLISION_PIXEL`, se debe eliminar el objeto almacenado en `collision` y crear un objeto `PixelCollision`. Sus valores geométricos los dan las coordenadas del sprite y el objeto `CollisionPixelData` establecido mediante el método `SetCollisionPixelData`.
- Cuando el tipo de colisión es `COLLISION_RECT`, se debe eliminar el objeto almacenado en `collision` y crear un objeto `RectCollision`. Sus valores geométricos los dan las coordenadas del sprite y el tamaño de la imagen del sprite. En el método `Update` del sprite se realizan las llamadas necesarias a los métodos sobrecargados `UpdateCollisionBox` para actualizar los valores geométricos de la colisión.

La realización de esta práctica está dividida en dos partes:

Primera parte (0,5 puntos)

Se deben de añadir al motor las clases `Collision`, `CircleCollision` y `RectCollision` e implementar una clase `CollisionManager` con la interfaz proporcionada. Se deben

implementar todos los métodos, aunque aquellos relativos a la detección de colisiones a nivel de píxel de deben dejar en blanco (devolviendo el valor `false` únicamente).

Para la implementación de la clase `CollisionManager`, debemos recurrir a diversas funciones del módulo `Math` que aún están por implementar. A continuación se indican dichos métodos y su funcionalidad:

- `PointInRect`: Si la `x` dada está dentro de los límites horizontales del rectángulo pasado, y la `y` dada está dentro de los límites verticales, devolvemos `true`.
- `RectsOverlap`: Buscar una implementación para este método, que debe indicar si dos rectángulos están solapados.
- `OverlappingRect`: Debe ajustarse al siguiente pseudocódigo:

```
si izquierda1 está dentro de los límites horizontales del segundo rectángulo
      outx = izquierda1
si no
      outx = izquierda2
si arriba1 está dentro de los límites verticales del segundo rectángulo
      outy = arriba1
si no
      outy = arriba2
si derecha1 está dentro de los límites horizontales del segundo rectángulo
      outwidth = derecha1 - outx
si no
      outwidth = derecha2 - outx
si abajo1 está dentro de los límites verticales del segundo rectángulo
      outheight = abajo1 - outy
si no
      outheight = abajo2 - outy
```

También hay que implementar los métodos de colisiones de la clase `Sprite`:

- `SetRadius` y `GetRadius` establecen y devuelven respectivamente el valor de la variable miembro `radius`.
- `SetCollision` debe implementar la funcionalidad indicada anteriormente.
- `GetCollision` devuelve el valor de la variable miembro `collision`.
- `CheckCollision(Sprite* other)` debe implementarse en base al siguiente pseudocódigo:

```
si this.collision no es null y other.collision no es null
      si this.collision.collision(other.collision) es true
            this.colSprite = other
            this.collided = true
            other.colSprite = this
            other.collided = true
```

```
                devolvemos true
        si no
                devolvemos false
si no
        devolvemos false
```

- `CheckCollision(Map* map)` no debe implementarse en esta práctica.
- `CollisionSprite` debe devolver el valor de la variable miembro `colSprite`.
- `DidCollide` debe devolver el valor de la variable miembro `collided`.
- `UpdateCollisionBox()` debe implementarse en base al siguiente pseudocódigo:

```
cx = coordX - handleXimagen * fabs(escalaX)
cy = coordy - handleYimagen * fabs(escalaY)
cw = anchoImagen * fabs(escalaX)
ch = altoImagen * fabs(escalaY)
UpdateCollisionBox(cx, cy, cw, ch)
```

- `UpdateCollisionBox(double x, double y, double w, double h)` debe implementarse en base al siguiente pseudocódigo:

```
colx = x
coly = y
colwidth = w
colheight = h
```

Hay que descomentar por último el código de detección de colisiones de la clase `Scene`.

Para probar que la funcionalidad del motor está correctamente implementada, realizaremos el siguiente ejercicio:

Crearemos una escena que contenga los siguientes elementos:

- Un sprite que aparezca en el cuarto superior izquierdo de la pantalla, utilizando la imagen **“data/ball.png”**, con colisiones por círculo y un radio apropiado para el tamaño de la imagen.
- Un sprite que aparezca en el cuarto inferior derecho de la pantalla, utilizando la imagen **“data/box.jpg”**, con colisiones por rectángulo.
- Un sprite que siga al puntero del ratón, cuyo modo de colisión e imagen puedan ser modificados.

Con respecto a este último sprite, podremos modificar su imagen y modo de colisión de la siguiente forma:

- Cuando pulsemos el botón izquierdo del ratón, utilizaremos la imagen **“data/circle.png”** y colisiones por círculo, con una radio apropiado al tamaño de la imagen.

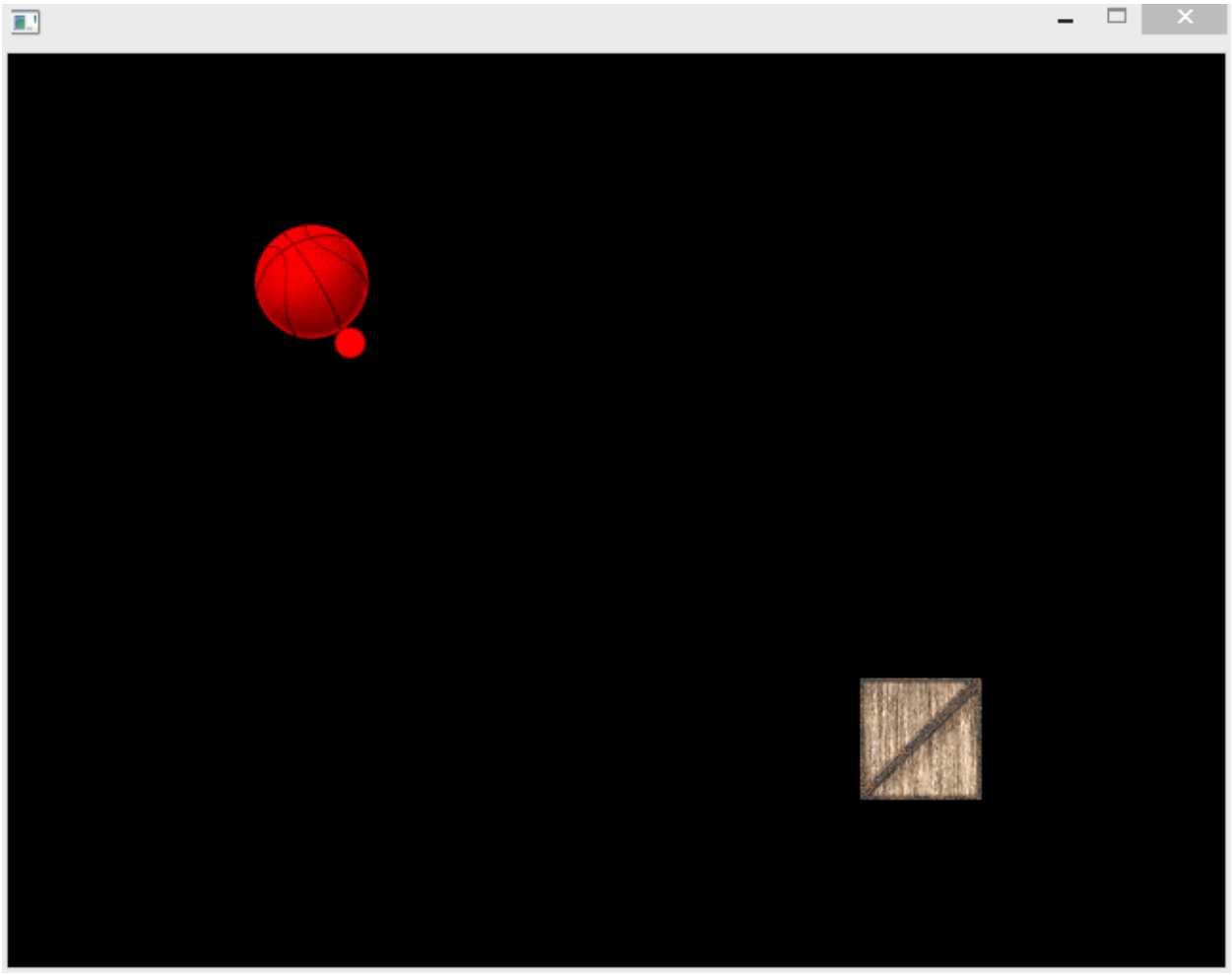
- Cuando pulsemos el botón derecho del ratón, utilizaremos la imagen **“data/rect.png”** y colisiones por rectángulo.

Inicialmente, el sprite estará en el primero de estos casos.

Al actualizar la escena, se actualizará automáticamente la información de colisión de todos sus sprites. Debemos hacer lo siguiente:

- Si un sprite ha colisionado, lo pintamos de color rojo (**RGB=255,0,0**).
- Si el sprite no ha colisionado, lo pintamos de color blanco (**RGB=255,255,255**).

El resultado quedará como puede verse en la siguiente imagen:



Segunda parte (0,5 puntos)

En la segunda parte de la práctica, incluiremos en el motor la clase `PixelCollision` e implementaremos la clase `CollisionPixelData` con la siguiente interfaz:

```
class CollisionPixelData {
public:
    CollisionPixelData(const String& filename);
    virtual ~CollisionPixelData();

    virtual bool IsValid() const;
    virtual const String& GetFilename() const;
    virtual uint16 GetWidth() const;
    virtual uint16 GetHeight() const;
    virtual bool GetData(uint32 x, uint32 y) const;
private:
    String filename;
    uint16 width, height;
    bool* data;
};
```

En el constructor, utilizaremos nuevamente la librería **STB_Image** para cargar la imagen especificada. Generaremos un buffer de booleanos con el tamaño correcto (el de la imagen cargada), y lo rellenaremos con los datos de cada píxel en base al siguiente patrón:

- Si el alpha del píxel de la imagen es 0, el valor de ese píxel del buffer será `false`.
- Si el alpha del píxel de la imagen es distinto de 0, el valor de ese píxel del buffer será `true`.

El destructor debe eliminar el buffer creado.

El resto de métodos se deben comportar de la siguiente forma:

- El método `IsValid` debe devolver `true` si el buffer es distinto de `NULL` (cosa que ocurrirá si la imagen pudo ser cargada).
- El método `GetFilename` debe devolver el nombre del fichero de imagen a partir del cual se generó el objeto.
- Los métodos `GetWidth` y `GetHeight` deben devolver el ancho y el alto en píxeles respectivamente del buffer.
- El método `GetData` debe devolver el valor del buffer en las coordenadas especificadas.

En la clase `CollisionManager`, debemos añadir el soporte para las colisiones a nivel de píxel. Por ejemplo, el método `PixelsToPixels` tiene el siguiente pseudocódigo:

```
si ambos CollisionPixelDatas se solapan
       obtenemos rectángulo del solapamiento
       obtenemos primer pixel de rectangulo 1
       obtenemos primer pixel de rectangulo 2
       for py = 0 until alto_solapamiento
              for px = 0 until ancho_solapamiento
                     si el pixel de ambos CollisionPixelDatas es true
```

```
                              devolvemos true
    si no
          devolvemos false
```

Debemos añadir a la clase `Sprite` el soporte para poder definir colisiones a nivel de píxel.

Además, en la clase `ResourceManager`, descomentaremos los métodos correspondientes a la carga de objetos `CollisionPixelData`.

Modificaremos el ejercicio de la parte anterior para añadir un sprite en el cuarto inferior izquierdo de la pantalla que utilice la imagen **“data/alien.png”** y colisión por píxeles utilizando la información del fichero **“data/aliencol.png”**.

Además, en el sprite que sigue al puntero de ratón, cuando pulsemos el botón central del ratón, el sprite utilizará colisión por píxeles con los mismos datos que el sprite mencionado anteriormente.

La solución del ejercicio quedará como muestra la siguiente imagen: