Spike: Task 26
Title: Collisions
Author: Sam Huffer, 101633177

Goals / deliverables:

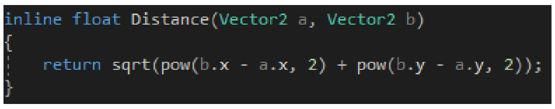
- At least box-based (axis-aligned rectangles) intersection testing, and circle-circle intersection testing
  - Display at least two boxes, one in a fixed position, the other(s) moving. Detect collisions using axis-aligned rectangle testing and visually display the "collided" status to the user
  - Display at least two circles, one in a fixed position, the other(s) moving. Detect collisions using circle-circle collision testing, and visually display the "collided status to the user.
- Incorporate extensibility in your collision data structure, so that you would be able to swap out one collision detection method for another.

Technologies, Tools, and Resources used:

- Visual Studio 2019
- Microsoft Word
- SDL2
- Online Resources
  1. Drawing circles: https://stackoverflow.com/questions/28346989/drawing-and-filling-a-circle#28351443

Tasks undertaken:

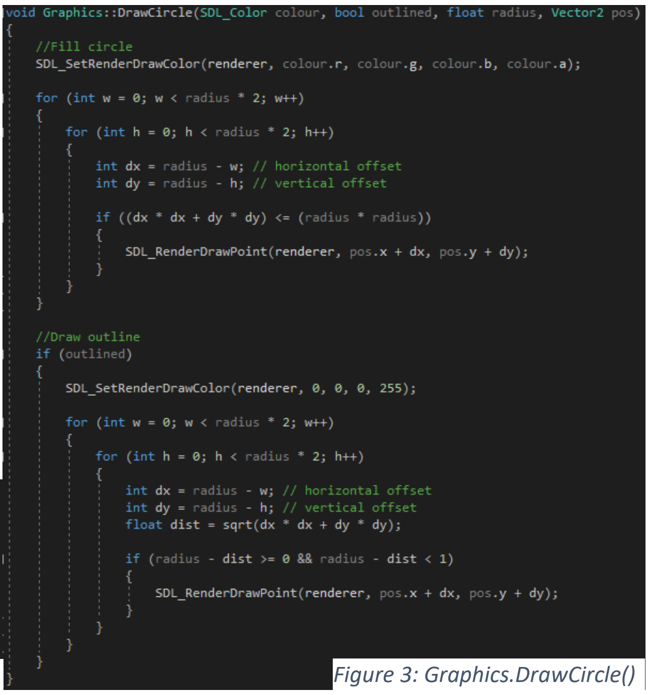
- I copied the task 24 spike report into the task folder, stripping out the spike report's original content and replacing it with goals and resources pertaining to the task at hand.



```
inline float Distance(Vector2 a, Vector2 b)
{
    return sqrt(pow(b.x - a.x, 2) + pow(b.y - a.y, 2));
}
```

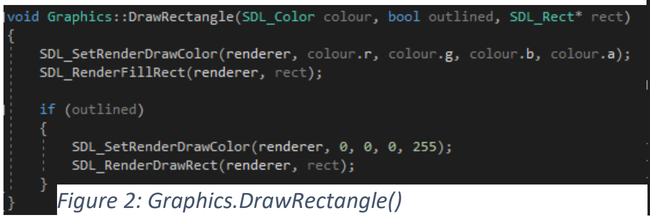
Figure 1: MathHelper.h's inline Distance() function

- I added to MathHelper.h a function Distance() to calculate the distance between two Vector2, and to GameEntity bools colliding and collidingLastFrame and public Get__() properties for both bools and a Set__() property for colliding that also relegates its value to collidingLastFrame before setting it.
- I added the Shape class (inheriting from GameEntity), which stores a ShapeType enum value, its current colour and whether it's outlined, and features appropriate public properties for each. Using it as a base class, I then added the Circle and Rectangle classes. Circle also stores a radius value (with appropriate public properties) and its Render() method makes a call to Graphics.DrawCircle(). Rectangle also has width and height values and an SDL_Rect field, the public properties for the former two assigning their values to the appropriate fields of both Rectangle and SDL_Rect to ensure it renders properly.



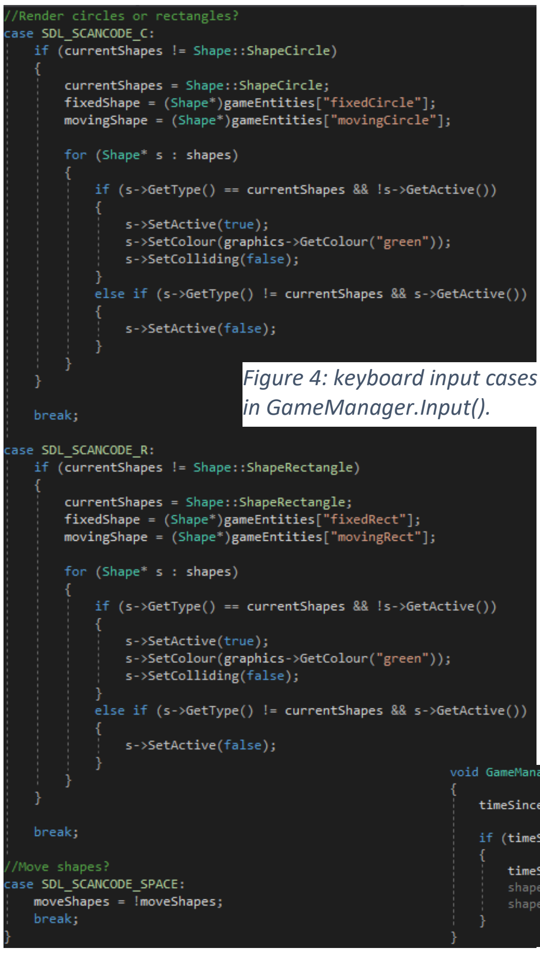
```
void Graphics::DrawCircle(SDL_Color colour, bool outlined, float radius, Vector2 pos)
{
    //Fill circle
    SDL_SetRenderDrawColor(renderer, colour.r, colour.g, colour.b, colour.a);

    for (int w = 0; w < radius * 2; w++)
    {
        for (int h = 0; h < radius * 2; h++)
        {
            int dx = radius - w; // horizontal offset
            int dy = radius - h; // vertical offset

            if ((dx * dx + dy * dy) <= (radius * radius))
            {
                SDL_RenderDrawPoint(renderer, pos.x + dx, pos.y + dy);
            }
        }
    }

    //Draw outline
    if (outlined)
    {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);

        for (int w = 0; w < radius * 2; w++)
        {
            for (int h = 0; h < radius * 2; h++)
            {
                int dx = radius - w; // horizontal offset
                int dy = radius - h; // vertical offset
                float dist = sqrt(dx * dx + dy * dy);

                if (radius - dist >= 0 && radius - dist < 1)
                {
                    SDL_RenderDrawPoint(renderer, pos.x + dx, pos.y + dy);
                }
            }
        }
    }
}
```

Figure 3: Graphics.DrawCircle()



```
void Graphics::DrawRectangle(SDL_Color colour, bool outlined, SDL_Rect* rect)
{
    SDL_SetRenderDrawColor(renderer, colour.r, colour.g, colour.b, colour.a);
    SDL_RenderFillRect(renderer, rect);

    if (outlined)
    {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderDrawRect(renderer, rect);
    }
}
```

Figure 2: Graphics.DrawRectangle()

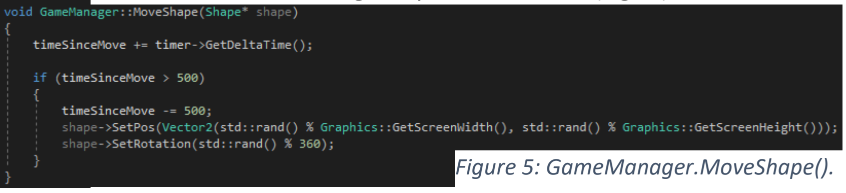


```
//Render circles or rectangles?
case SDL_SCANCODE_C:
    if (currentShapes != Shape::ShapeCircle)
    {
        currentShapes = Shape::ShapeCircle;
        fixedShape = (Shape*)gameEntities["fixedCircle"];
        movingShape = (Shape*)gameEntities["movingCircle"];

        for (Shape* s : shapes)
        {
            if (s->GetType() == currentShapes && !s->GetActive())
            {
                s->SetActive(true);
                s->SetColour(graphics->GetColour("green"));
                s->SetColliding(false);
            }
            else if (s->GetType() != currentShapes && s->GetActive())
            {
                s->SetActive(false);
            }
        }
    }

    break;

case SDL_SCANCODE_R:
    if (currentShapes != Shape::ShapeRectangle)
    {
        currentShapes = Shape::ShapeRectangle;
        fixedShape = (Shape*)gameEntities["fixedRect"];
        movingShape = (Shape*)gameEntities["movingRect"];

        for (Shape* s : shapes)
        {
            if (s->GetType() == currentShapes && !s->GetActive())
            {
                s->SetActive(true);
                s->SetColour(graphics->GetColour("green"));
                s->SetColliding(false);
            }
            else if (s->GetType() != currentShapes && s->GetActive())
            {
                s->SetActive(false);
            }
        }
    }

    break;

//Move shapes?
case SDL_SCANCODE_SPACE:
    moveShapes = !moveShapes;
    break;
}
```

Figure 4: keyboard input cases in GameManager.Input().

Rectangle.Render() borrows from Texture.Render(), but makes a call to Graphics.DrawRectangle() rather than DrawTexture().

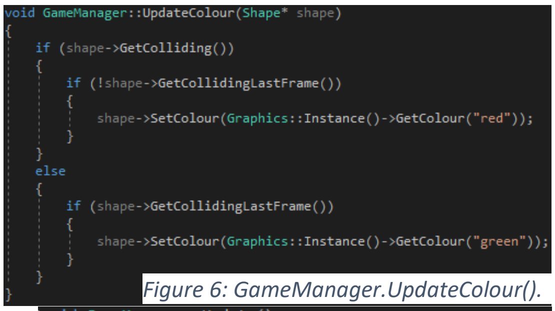
- I added to Graphics the methods DrawRectangle()and DrawCircle(). For the former, I used calls to SDL_RenderFillRect() and SDL_RenderDrawRect() (fig. 2). For the latter, however, I used custom code adapted from res. 1 (fig. 3).
- I added two Circles and Rectangles each to GameManager's list of GameEntities, one being designated as a fixed Shape in the centre of the screen the other being moved to different positions, and with Rectangles being designated to render by default.
- I moved input-, updating- and rendering-related code into their own methods (Input(), Update() and Render()) to clearly separate them, with calls to each being kept in Run()'s while(running) loop.
- I added to the keyboard input switch cases for switching between rendering Circles and Rectangles, resetting their colour and colliding status when selected, and for toggling the movement of moving Shapes on and off (fig. 4).



```
void GameManager::MoveShape(Shape* shape)
{
    timeSinceMove += timer->GetDeltaTime();

    if (timeSinceMove > 500)
    {
        timeSinceMove -= 500;
        shape->SetPos(Vector2(std::rand() % Graphics::GetScreenWidth(), std::rand() % Graphics::GetScreenHeight()));
        shape->SetRotation(std::rand() % 360);
    }
}
```

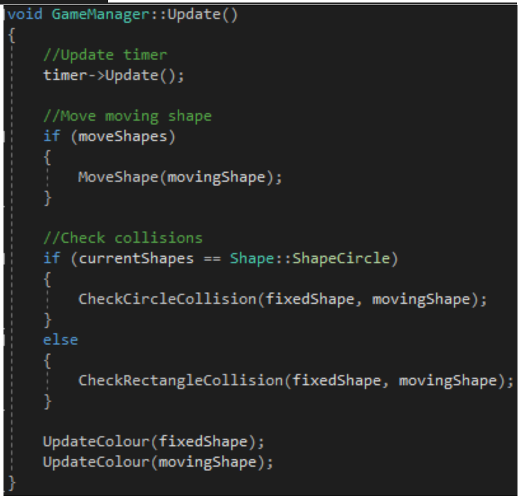
Figure 5: GameManager.MoveShape().

- I added the MoveShape() method to periodically randomise the position and rotation of the current moving Shape (fig. 5) and UpdateColour() to check the collision status of a Shape and set its colour accordingly (fig. 6), adding calls to each to Update() (fig. 7).
- I added to Update() calls to CheckCircleCollision() and CheckRectangleCollision() inside an if statement checking currentShapes. I then implemented CheckCircleCollision(), using the circles positions and radii with the Distance() method to determine if the circles were colliding (fig. 8).
- I had a look at my lecture notes and the task instructions to assess what I would need to do to calculate box-on-box collisions for this task, and what values would be required. From there, I added to Rectangle the public properties GetBottomLeft() and GetTopRight() to calculate and return the position of the appropriate points of the Rectangle. I implemented them with some inspiration from GameEntity.GetPos() and how it accounts for the GameEntity's parent's position and rotation (fig. 9). To test them, I added to Rectangle.Render() calls to graphics->DrawCircle() (using the calculated points for the circles' positions) and childed an additional rectangle to movingRect so that I could verify that GetBottomLeft() and GetTopRight() were producing the right values.



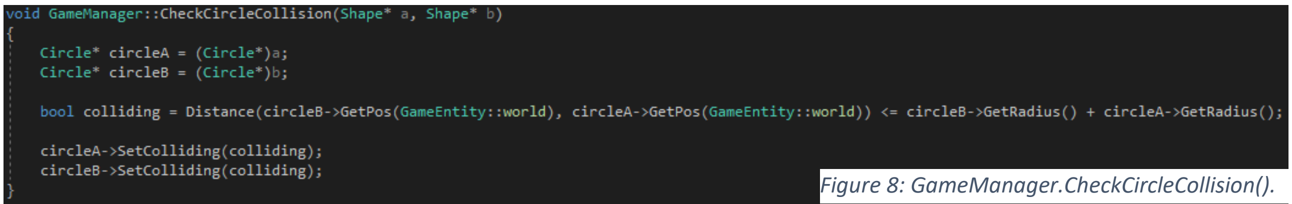
```
void GameManager::UpdateColour(Shape* shape)
{
    if (shape->GetColliding())
    {
        if (!shape->GetCollidingLastFrame())
        {
            shape->SetColour(Graphics::Instance()->GetColour("red"));
        }
    }
    else
    {
        if (shape->GetCollidingLastFrame())
        {
            shape->SetColour(Graphics::Instance()->GetColour("green"));
        }
    }
}
```

Figure 6: GameManager.UpdateColour().



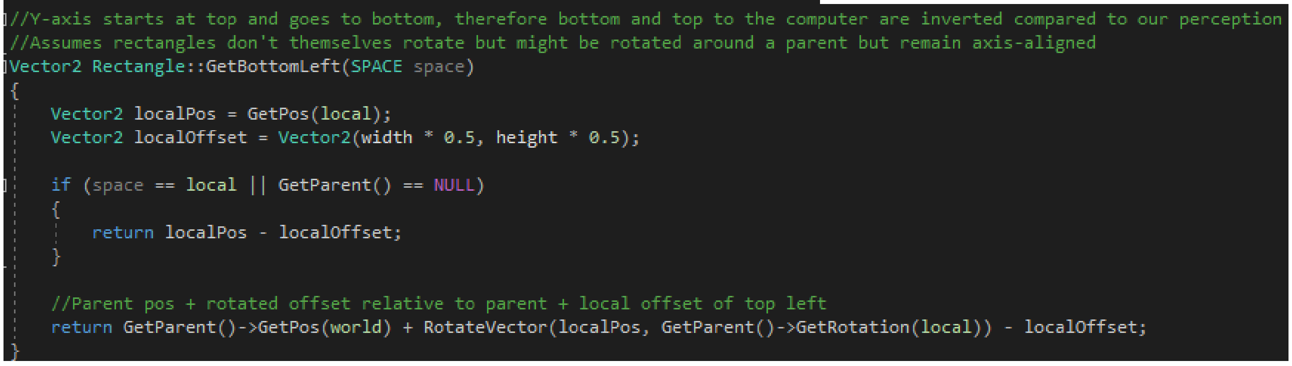
```
void GameManager::Update()
{
    //Update timer
    timer->Update();

    //Move moving shape
    if (moveShapes)
    {
        MoveShape(movingShape);
    }

    //Check collisions
    if (currentShapes == Shape::ShapeCircle)
    {
        CheckCircleCollision(fixedShape, movingShape);
    }
    else
    {
        CheckRectangleCollision(fixedShape, movingShape);
    }

    UpdateColour(fixedShape);
    UpdateColour(movingShape);
}
```

Figure 7: GameManager.Update().



```
void GameManager::CheckCircleCollision(Shape* a, Shape* b)
{
    Circle* circleA = (Circle*)a;
    Circle* circleB = (Circle*)b;

    bool colliding = Distance(circleB->GetPos(GameEntity::world), circleA->GetPos(GameEntity::world)) <= circleB->GetRadius() + circleA->GetRadius();

    circleA->SetColliding(colliding);
    circleB->SetColliding(colliding);
}
```

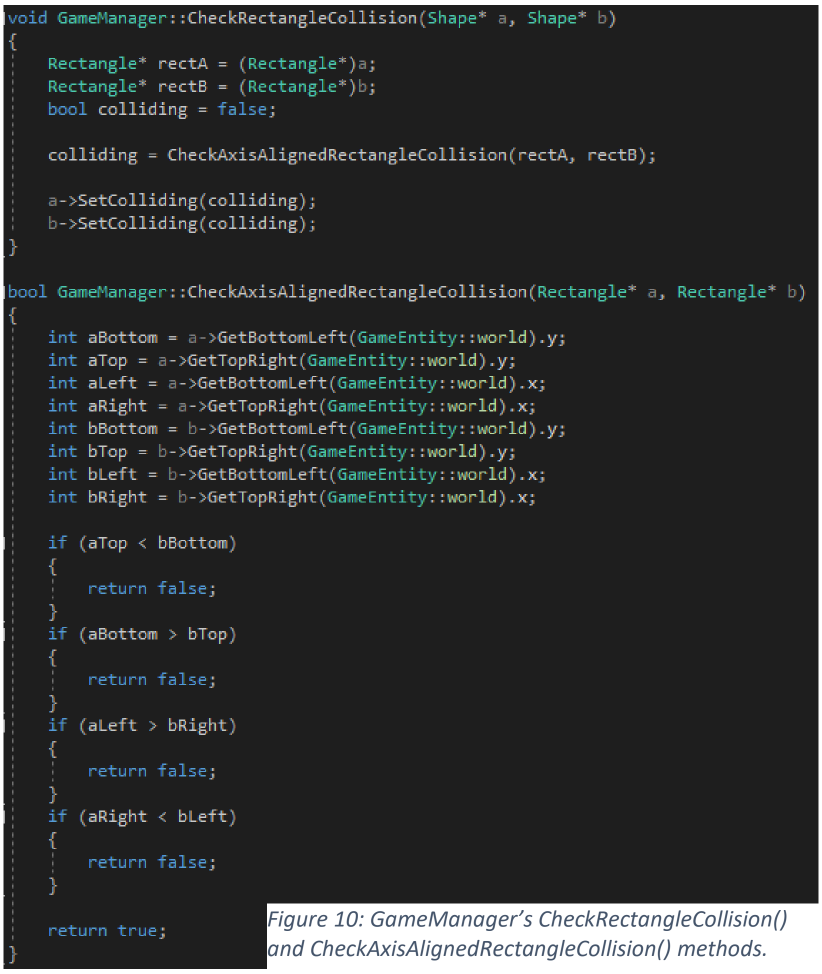
Figure 8: GameManager.CheckCircleCollision().



```
//Y-axis starts at top and goes to bottom, therefore bottom and top to the computer are inverted compared to our perception
//Assumes rectangles don't themselves rotate but might be rotated around a parent but remain axis-aligned
Vector2 Rectangle::GetBottomLeft(SPACE space)
{
    Vector2 localPos = GetPos(local);
    Vector2 localOffset = Vector2(width * 0.5, height * 0.5);

    if (space == local || GetParent() == NULL)
    {
        return localPos - localOffset;
    }

    //Parent pos + rotated offset relative to parent + local offset of top left
    return GetParent()->GetPos(world) + RotateVector(localPos, GetParent()->GetRotation(local)) - localOffset;
}
```

Figure 9: Rectangle.GetBottomLeft(). GetBottomRight() works the same, but instead adds the localOffset rather than subtracting it. Furthermore, accounting for local and parent rotations is restricted to ensuring the position of the Rectangle rotated around its parent is correct; SDL_Rect doesn't afford rotation, so it remains fixed and axis-aligned as far as actually rotating the rectangle shape at its position is concerned.

- I implemented CheckRectangleCollision() to make the Rectangle collision checks required for this task, box-on-box axis-aligned checks being encapsulated in CheckAxisAlignedRectangleCollision() rather than being incorporated in the method directly to allow extensible adding of further checks if Rectangle rotation were to be fully implemented. For the axis-aligned checks, I tried implementing the algorithm for them from my lecture notes, but it didn't seem to be working, returning that the rectangles were colliding even when they weren't. I erased it but took the same logic and typed it out manually, and that seemed to fix it (fig. 10).



```
void GameManager::CheckRectangleCollision(Shape* a, Shape* b)
{
    Rectangle* rectA = (Rectangle*)a;
    Rectangle* rectB = (Rectangle*)b;
    bool colliding = false;

    colliding = CheckAxisAlignedRectangleCollision(rectA, rectB);

    a->SetColliding(colliding);
    b->SetColliding(colliding);
}

bool GameManager::CheckAxisAlignedRectangleCollision(Rectangle* a, Rectangle* b)
{
    int aBottom = a->GetBottomLeft(GameEntity::world).y;
    int aTop = a->GetTopRight(GameEntity::world).y;
    int aLeft = a->GetBottomLeft(GameEntity::world).x;
    int aRight = a->GetTopRight(GameEntity::world).x;
    int bBottom = b->GetBottomLeft(GameEntity::world).y;
    int bTop = b->GetTopRight(GameEntity::world).y;
    int bLeft = b->GetBottomLeft(GameEntity::world).x;
    int bRight = b->GetTopRight(GameEntity::world).x;

    if (aTop < bBottom)
    {
        return false;
    }
    if (aBottom > bTop)
    {
        return false;
    }
    if (aLeft > bRight)
    {
        return false;
    }
    if (aRight < bLeft)
    {
        return false;
    }

    return true;
}
```

Figure 10: GameManager's CheckRectangleCollision() and CheckAxisAlignedRectangleCollision() methods.

What we found out:

- Between SDL2, SDL_Mixer and SDL_Image, there is not a method for drawing circles to the screen like there is for rectangles, requiring custom methods. The one I found works, but seems to be slower than drawing equivalent rectangles.
- SDL_Rects don't inherently have rotating functionality.
- How to calculate corner positions of an axis-aligned rectangle, accounting for rotation around a parent object but not the rectangle's own rotation.
- How to check if un-rotated boxes have collided.
- Either I copied out or implemented the axis-aligned box-on-box collision checking algorithm incorrectly, or there is a mistake in the algorithm.