# 1 Introduction

This document contains notes on collision detection and collision resolution. These notes are based on course material from COMP 4300 at the Memorial University of Newfoundland taught by Prof. David Churchill.

# 2 Collision Detection

## 2.1 Collision Detection Problems

- Given two entities which have a current position, do they intersect?
  1. If they do intersect, how do we resolve the collision?
  2. If they do not intersect, how do we determine if they will intersect in the future?

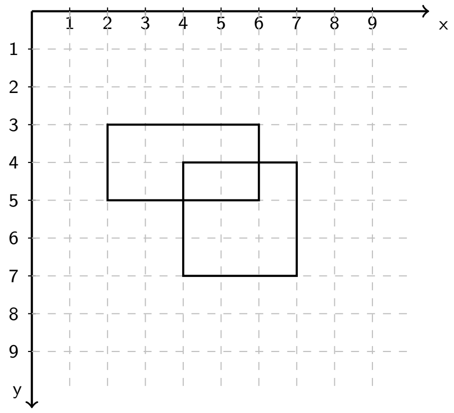


Figure 1: Intersecting Boxes

## 2.2 Entity Bounding Shapes

Everyday objects have arbitrary shapes and interaction surfaces. For example, consider a teacup. A teacup has a complex shape consisting of many curves and edges. Accurately simulating the collision of a teacup requires considering both the teacup's shape and the shape of the object it collides with. With many complex shapes, collision detection becomes computationally expensive.

To address this issue, we use bounding shapes that approximate an object's shape to calculate collisions more efficiently. The most effective way to do this is by using primitive types:

- **2D:** Circle, Rectangle, Triangle, Octagon
- **3D:** Sphere, Box, Cylinder, Cone

The circle is a common primitive shape and is the simplest possible intersection bounding shape. Another commonly used shape is the rectangle, which encloses an object in a box.

By using these primitive shapes, collision detection becomes simpler and less computationally demanding.

### 2.2.1 Bounding Circles

As mentioned, the circle is the simplest possible intersection bounding shape in regards to collision detection. This is because a circle is only defined only by its center and radius, making calculations for collision detection easier.

Calculating the distance between two circles is a simple task. Given two circles with centers $c_1$ and $c_2$ and radii $r_1$ and $r_2$, the distance between the two circles is given by the formula:

$$d = \sqrt{(c_2.x - c_1.x)^2 + (c_2.y - c_1.y)^2} \tag{1}$$

This is derived from the Pythagorean theorem, where the length of the hypotenuse of a right triangle is given by the square root of the sum of the squares of the other two sides.

$$c = \sqrt{a^2 + b^2} \tag{2}$$

If the distance between the two circles is less than the sum of their radii, then the circles intersect.

```cpp
#include "Vec2.h"
#include <cmath>
#include <iostream>

struct Circle {
    Vec2 center;
    float radius;
};

bool checkCollision(const Circle &c1, const Circle &c2) {
    const float dx = c2.center.x - c1.center.x;
    const float dy = c2.center.y - c1.center.y;
    const float distance = std::sqrt(dx * dx + dy * dy);
    const float sumOfRadii = c1.radius + c2.radius;
    const bool collision = distance < sumOfRadii;

    return collision;
}
```

**Listing 1:** *Collision Detection for Bounding Circles in C++*

### 2.2.2 Bounding Boxes

In 2D games, objects are often enclosed in rectangles known as bounding boxes. These are usually the smallest possible rectangle that completely encompasses the texture's width and height. However, this is not always the case, as some imperfections may exist such as the entity being larger than the bounding box.

Rectangles can be oriented in any direction as long as all four sides meet at 90-degree angles. This creates complexity in collision detection, requiring calculations for line-line intersections. To simplify this, we can use something called an axis-aligned bounding box (AABB). An AABB is a rectangle that is aligned with the x and y axes, i.e. with sides parallel to the coordinate axes.

The simplest calculation for AABBs is to check whether a point is inside the rectangle. Given a point $p$ and a rectangle with corners $c_1$ and $c_2$, we can determine if the point is inside the rectangle by checking if the point's x and y coordinates are within the rectangle's x and y coordinates.

This is done by the following formula:

$$\begin{aligned} &\textit{Point } P \textit{ is inside rectangle with corners } c_1 \textit{ and } c_2 \textit{ if and only if:} \\ &(p.x > c_1.x)\,\&\,(p.x < c_2.x)\,\&\,(p.y > c_1.y)\,\&\,(p.y < c_2.y) \end{aligned} \tag{3}$$

broken down, the formula is evaluating four conditions:

1. The point's x-coordinate is to the right of the left side of the rectangle.

$$p.x > c_1.x \tag{4}$$

2. The point's x-coordinate is to the left of the right side of the rectangle.

$$p.x < c_2.x \tag{5}$$

3. The point's y-coordinate is above the bottom side of the rectangle.

$$p.y > c_1.y \tag{6}$$

4. The point's y-coordinate is below the top side of the rectangle.

$$p.y < c_2.y \tag{7}$$