Collision Detection and Collision Resolution

Aaron Po

June 24, 2024

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?

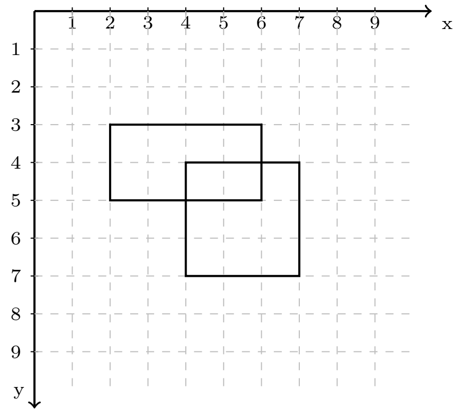

Figure 1: Intersecting Boxes

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 Axis Aligned Bounding Boxes (AABB)

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.

Point inside AABB 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:

$$\textit{Point } p \textit{ is inside rectangle with corners } c_1 \textit{ and } c_2 \textit{ if and only if:}$$
$$(p.x > c_1.x) \textbf{ AND } (p.x < c_2.x) \textbf{ AND } (p.y > c_1.y) \textbf{ AND } (p.y < c_2.y)$$

If the rectangle is defined by a top left corner c, width w, and height h, the equation can be modified to:

$$\textit{Point } p \textit{ is inside rectangle with top left corner } c\textit{, width } w \textit{ and height } h \textit{ if and only if:}$$
$$(p.x > c.x) \textbf{ AND } (p.x < c.x + w) \textbf{ AND } (p.y > c.y) \textbf{ AND } (p.y < c.y + h)$$

When 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.
$$p.x > c_1.x$$
$$p.x > c.x$$

2. The point's x-coordinate is to the left of the right side of the rectangle.
$$p.x < c_2.x$$
$$p.x < c.x + w$$

3. The point's y-coordinate is above the bottom side of the rectangle.
$$p.y > c_1.y$$
$$p.y > c.y$$

4. The point's y-coordinate is below the top side of the rectangle.
$$p.y < c_2.y$$
$$p.y < c.y + h$$

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

struct AABB {
    Vec2 topLeft;
    float width;
    float height;
};

bool pointInsideAABB(const Vec2 &p, const AABB &aabb) {
    const bool inside = p.x > aabb.topLeft.x &&
                        p.x < aabb.topLeft.x + aabb.width &&
                        p.y > aabb.topLeft.y &&
                        p.y < aabb.topLeft.y + aabb.height;

    return inside;
}
```

Listing 2: *Point Inside AABB in C++*

AABB Intersection Determining if two AABBs intersect is a relatively simple task. Given two AABBs, one may be inclined to check for collision using the point inside AABB method. However, this is not the most efficient method, as it requires checking all four corners of one rectangle against the other rectangle. Instead, we can use an algorithm that detects both horizontal and vertical overlap.

Given two AABBs with top left corners c_1 and c_2, we can determine if they overlap horizontally by checking if the right side of the first rectangle is to the right of the left side of the second rectangle and vice versa. This is done by the following formula:

$$\begin{gathered}\textit{Rectangles overlap horizontally if and only if:}\\ (c_1.x < c_2.x + c_2.w)\ \textbf{AND}\ (c_1.x + c_1.w > c_2.x)\end{gathered} \tag{3}$$

In a similar manner, we can determine if the rectangles overlap vertically by checking if the top side of the first rectangle is above the bottom side of the second rectangle and vice versa. This is done by the following formula:

$$\begin{gathered}\textit{Rectangles overlap vertically if and only if:}\\ (c_1.y < c_2.y + c_2.h)\ \textbf{AND}\ (c_1.y + c_1.h > c_2.y)\end{gathered} \tag{4}$$

Put all together, the AABBs overlap if and only if they overlap both horizontally and vertically. This is done by the following formula:

$$\begin{aligned} &(c_1.x < c_2.x + c_2.w)\\ \textbf{AND}\ &(c_1.x + c_1.w > c_2.x)\\ \textbf{AND}\ &(c_1.y < c_2.y + c_2.h)\\ \textbf{AND}\ &(c_1.y + c_1.h > c_2.y)\end{aligned} \tag{5}$$

Example of AABB intersection:

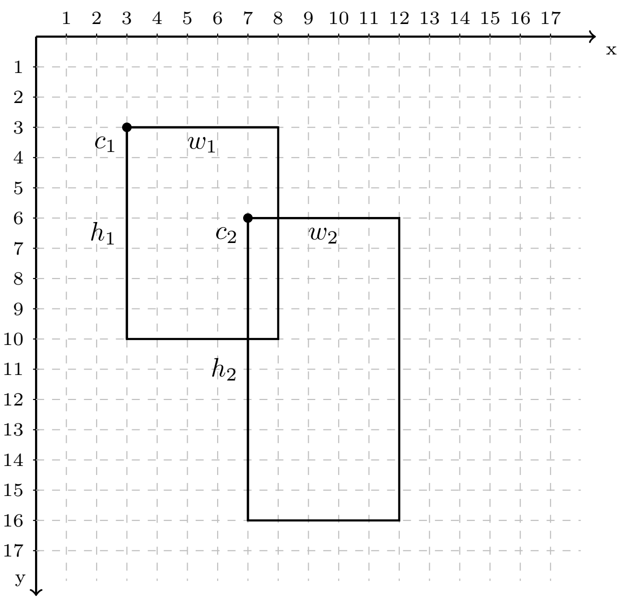

Figure 1: *AABB Intersection Example*

It is important to note that the above formula can only check whether or nor not two AABBs intersect. It does not provide information on how much they intersect. To determine the overlap, we can use a calculation that takes a center point and the width and height of the bounding box. This is done as follows:

Given two AABBs with centers c_1 and c_2, widths w_1 and w_2, and heights h_1 and h_2,

$$\begin{aligned} \Delta &= \{|c_1.x - c_2.x|, |c_1.y - c_2.y|\} \\ O &= \left\{\left[\left(\frac{w_1}{2} + \frac{w_2}{2}\right) - \Delta_x\right], \left[\left(\frac{h_1}{2} + \frac{h_2}{2}\right) - \Delta_y\right]\right\} \\ (O_x > 0) \textbf{ AND } (O_y > 0) &\rightarrow \text{AABBs intersect} \end{aligned} \tag{6}$$

In this formula, the vector Δ represents the absolute difference between the center coordinates of the two AABBs along the x and y axes.

The vector O represents the overlap between the two AABBs along the x and y axes. The overlap along each axis is calculated by subtracting the difference between the centers from the sum of the half-widths (for the x axis) or half heights (for the y axis) of the two AABBs. Here is an implementation of the AABB intersection algorithm in C++:

```cpp

#include "Vec2.h"
#include <iostream>

struct AABB {
    Vec2 center;
    float width;
    float height;
};

bool checkAABBIntersection(const AABB &a, const AABB &b) {
    const Vec2 delta = {
        std::abs(a.center.x - b.center.x),
        std::abs(a.center.y - b.center.y)
    };

    const Vec2 overlap = {
        (a.width / 2 + b.width / 2) - delta.x,
        (a.height / 2 + b.height / 2) - delta.y
    };

    const bool AABBIntersects = overlap.x > 0 && overlap.y > 0;

    return AABBIntersects;
}
```

```cpp
#include "Vec2.h"
class CTransform : public Component {
    public :
        Vec2 position = {0, 0};
        Vec2 previousPosition = {0, 0};
        Vec2 scale = {1.0, 1.0};
        float angle = 0;
};

class CBoundingBox : public Component {
    public :
        Vec2 size;
        Vec2 halfSize;
        CBoundingBox(const Vec2 &size) : size(size), halfSize(size / 2) {}
};
```

Appendix

Vec2.h

```cpp
#ifndef VEC2_H
#define VEC2_H

class Vec2 {
public:
};
    double x, y;

    Vec2(float x = 0, float y = 0);

    bool operator==(const Vec2 &rhs) const;
    bool operator!=(const Vec2 &rhs) const;

    Vec2 operator-(const Vec2 &rhs) const;
    Vec2 operator+(const Vec2 &rhs) const;
    Vec2 operator*(const float val) const;
    Vec2 operator/(const float val) const;

    void operator+=(const Vec2 &rhs);
    void operator-=(const Vec2 &rhs);
    void operator*=(const float val);
    void operator/=(const float val);

    Vec2 normalize();
    float length() const;
};

#endif // VEC2_H
```