Production Systems and Information Engineering
Volume 11 (3), pp. 10-26
doi: https://doi.org/10.32968/psaie.2023.3.2

COLLISION DETECTION IN 2D GAMES

PÉTER MILEFF
University of Miskolc, Hungary
Institute of Information Technology
mileff@iit.uni-miskolc.hu

Abstract. Collision detection is a fundamental aspect of game development, as it enables game objects to interact with each other in a realistic and intuitive way. In 2D games, collision detection is typically done by checking for overlaps between the bounding boxes or shapes of game objects. However, as games have become more complex, more sophisticated algorithms and techniques have been developed. This publication provides an overview of the various algorithms and techniques used for 2D collision detection in games. We discuss the pros and cons of different approaches, such as the Axis-Aligned Bounding Box (AABB) approach, the Oriented Bounding Box, the Separating Axis Theorem (SAT) and other useful techniques. We also discuss the challenges and limitations of these approaches and provide guidance on how to choose the appropriate approach based on specific needs and constraints.

Keywords: Collision detection, AABB, Separating Axis Theorem

1. Introduction

Modern game development is a dynamic and rapidly-evolving field that involves creating cutting-edge interactive experiences for players across a wide range of platforms and devices. With advances in hardware and software technology, game developers now have access to powerful tools and engines that allow them to create highly-realistic graphics, immersive soundscapes, and sophisticated gameplay mechanics. One major trend in modern game development is the rise of cross-platform development, which allows games to be played on multiple devices and platforms, including consoles, PCs, and mobile devices. This has opened up new opportunities for game developers to reach wider audiences and create innovative gaming experiences that can be enjoyed by players across a range of devices.

One important aspect of game development is collision detection, which is the process of detecting when two or more objects in a game world come into contact or intersect with each other. Collision detection is essential for creating realistic and immersive gaming experiences. Whether we are creating a first-person shooter, a racing game, a platformer or a puzzle game, collision detection is used to ensure that the game world behaves as expected and that players can interact with objects and characters in a meaningful way.

Collision detection in modern game development is a complex process that requires a deep understanding of both the physical properties of objects in the game world and the computational techniques used to simulate their behavior. There are several different approaches to collision detection, each with its own strengths and weaknesses.

The central topic and aim of this article is to provide an overview of the most important techniques that are essential for creating a two-dimensional game. The algorithms presented from a practical point of view can help to efficiently implement collision detection.

2. Collision detection

The interaction between objects is an essential element of computer games, meaning the examination of when two objects collide or make contact with each other. This principle applies not only to the world of games but also when, for example, we place a mouse over a menu item. Of course, in computer games, the dominant role is played by the proper detection of interactions, as the gaming experience is shaped by these interactions. For example, in an action game, the bullet hits the enemy, or the hero cannot walk through the maze wall.

In a very simplified way, the essence of collision detection is to somehow algorithmically detect whether the two-dimensional images of two or more objects overlap each other. To be more precise, the problem is a bit more complicated than that: it means determining whether one object has a pixel that overlaps with a pixel of another object.

During the development of a game, there will certainly come a crucial moment when we have to decide what collision detection system or model to use. The decision is not always easy and straightforward. There are types of games where interactions can be very complex, and often not all problems are visible in advance. Nevertheless, the model applied is important, as it greatly affects both the development time and the gaming experience itself. Essentially, collision detection systems can be divided into two groups:

- **Pixel-based collision detection:** examines the overlap of the pixels of the images belonging to the objects that collide. It can detect a precise, real collision.

- **Bounding object-based collision detection:** The overlapping of objects is not determined at the pixel level, but at the level of some enclosing object(s) (box, circle, polygon, etc.). It usually doesn't allow precise collision.

Pixel-based collision detection can be computationally intensive and complicated, depending on the complexity of the texture associated with the object. For this reason, where possible, game developers try to enclose the moved elements in some object(s) and perform the collision detection on this. Usually, the choice falls on the circle or box, because they are very simple elements. The subsequent calculations with them (collision test, rotation, translation, etc.) are not nearly as computationally intensive as, for example, in the case of a bounding polygon or the pixel-level test. Although they do not approximate the object well, they are still effective and can be used well in practice.

In the following, this paper focuses on bounding object based collision solutions. It presents the most important procedures necessary for this to be achieved.

3. Bounding circle collision

3.1. Bounding circle collision

The bounding circle (or possibly ellipse) based collision detection is the simplest possible known solution for deciding whether two objects overlap. In this case, all objects are enclosed in a (somewhat appropriate) circle. The center and radius of the circle are usually determined in such a way as to make the collision test as efficient as possible.

If the circle is too large or does not fit well on the object, there will be false collisions at the edges of the object. And if it is too small, the object may even partially hang into the other object, for example into the wall. Of course, it is also possible to calculate the circle even automatically. In this case, after loading the object's representing graphic element (image), its width and height values can provide a clue. However, automation is not suitable in many cases, because the goal of defining the bounding circle is not always that all pixels of the object fall within it. The figure below shows this:

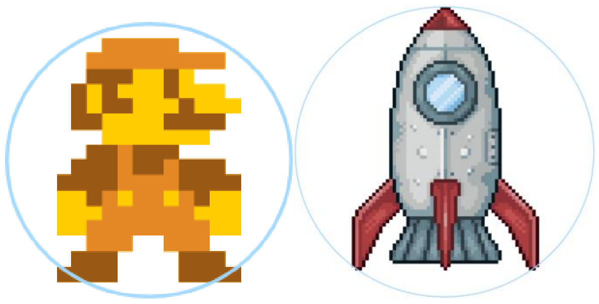
Figure 1. Bounding circle around the game object. It can be seen that in the case of the rocket on the right, the application of the bounding circle is no longer ideal

The question arises as to how it is worthwhile to specify the bounding circles. Usually by hand, by creating an additional description file for the different game objects, which contains not only the image elements of the animation phases, but also the corresponding bounding circles.

3.2. Collision detection

Collision detection is the easiest for this type of bounding object. This is why many developers, especially beginner game developers, like to use it. Since the two objects are regular, we can speak of a collision when the bounding circles overlap each other, so we must examine this fact. And two bounding circles only overlap when the distance between the centers of the circles is less than the sum of the radii:

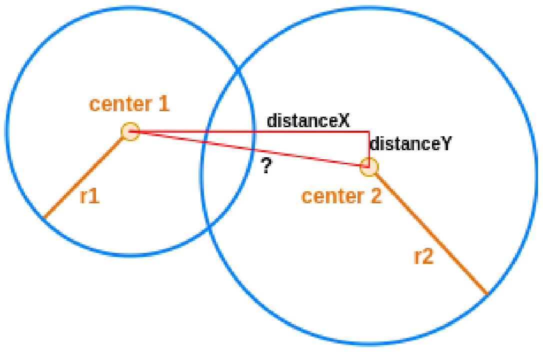

Figure 2. Collision test with bounding circles

Collision detection pseudo code:

```
bool checkCollision(center1, r1, center2, r2) {

        // Calculate the distance of the centers
        distanceX = center1.X - center2.X
        distanceY = center1.Y - center2.Y

        // Calculate distance based on Pythagorean theorem
        d = sqrt((distanceX * distanceX) + (distanceY * distanceY))

        // Check collision
        if (d <= (r1 + r2))
                return true; // Collision

        return false; // No collision
}
```

The main virtue of the solution - as can be clearly seen from the sample code - is that it is extremely simple and requires little resources from a computational point of view. It is mainly recommended in cases where we have objects that fit well within the circle.

4. Bounding box based collision

Another simplest, but still most popular form of collision detection is the bounding box-based solution (rectangular collision detection). In this case, the object is surrounded by a "box", i.e. a square or rectangle (Figure 3). Its popularity stems, on the one hand, from the fact that its implementation is also simple, and on the other hand, due to its adaptability, non-circular objects involved in the collision can be relatively well handled with it.

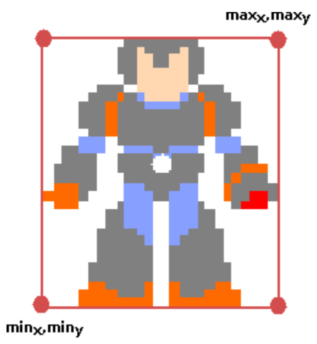

Figure 3. Best fitting bounding box. There are apparently no empty pixels between the outermost points of the shape and the sides of the box

In the simplest case, the bounding box is determined by the object's two-dimensional image and texture. This can be calculated very simply when loading, since the width and height of the texture image will be in the case that the Sprite is drawn correctly and does not contain unnecessary transparent pixels on the edges. When defining the box, they usually try to establish or specify the best fitting rectangle (Best fit rectangle, minimum bounding rectangle (MBR)). The reason for this is to reduce/avoid false collisions. Just think about the fact that if we increased the size of the box along the x axis in the case of the above object, it would result in us sensing a collision even when we have not yet reached the wall.

The following code shows a class description for a possible bounding box implementation:

```
class CboundingBox2D {

  CVector2 minpoint;            // Box minpoint
  CVector2 maxpoint;            // Box maxpoint
  CVector2 bbPoints[4];         // bounding box points
  float boxHalfWidth;           // box half width
  float boxHalfHeight;          // box half height
  matrix4x4f tMatrix;           // Transformation matrix
  bool mEnabled;                // BB is enabled or not

public:
...
};
```

In two dimensions, a bounding box can be specified with its 4 corner points (*bbPoints[4]*), but in order to speed up later calculations, it is advisable to store the minimum (*minpoint*) and maximum point (*maxpoint*) relative to the screen coordinate system. In the figure above, this means the lower left and upper right points. On top of all that, the calculations will require a matrix class that performs the transformation, and from the point of view of acceleration, it is worth storing half the width and height of the box.

4.1. Box-Box collision detection

The algorithm for determining collisions is very simple to formulate: when the bounding boxes of two objects overlap, the objects collide. The following figure illustrates this:

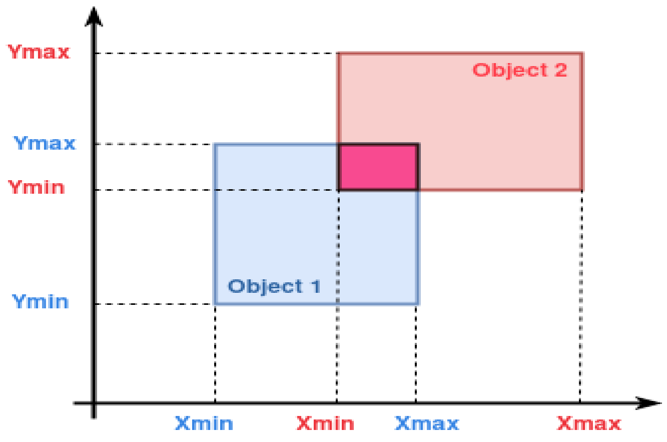

Figure 4. AABB collision

The computational requirements of overlapping two boxes are not high. However, if there are many objects on the screen and we do not want a serious part of the CPU time to be spent on their calculation. From an implementation point of view it is better to examine the case where there are no collisions. This results in a faster solution:

```
boolean CheckBoxOverLap(CBoundingBox2D box1, CboundingBox2D box2)
{
        if (box1.maxpoint.x < box2.minpoint.x ||
            box1.minpoint.x > box2.maxpoint.x){
           return false;
        }

        if (box1.maxpoint.y < box2.minpoint.y ||
           box1.minPoint.y > box2.maxpoint.y){
           return false;
        }
        return true;
}
```

The error of the bounding box-based solution should also be mentioned. If objects that are "holey" collide, and only the holey parts overlap each other, there is no actual collision, as shown in the image above. Although they do not approximate the object well, they are still effective and can be used well in practice. Despite the error, this solution was the most widespread in the field of game development. The reason for this is its simplicity and reduced computational requirements, as well as the fact that in the case of most games, during fast movement, we don't notice that "object didn't even collide with the pixel of the object".

Although we stated above that it is advisable to define the narrowest box, in many cases this is unfortunately not enough for realism. A typical example is platform games, where the character is constantly pulled by gravity and does not fall until his box collides with a box on the ground. It is not advisable to match the box to the figure's texture in every phase, otherwise the object may not fall when it should. The following image is from a classic game where the character's box is exactly the size of its image, allowing it to stand up to 1 pixel tall.

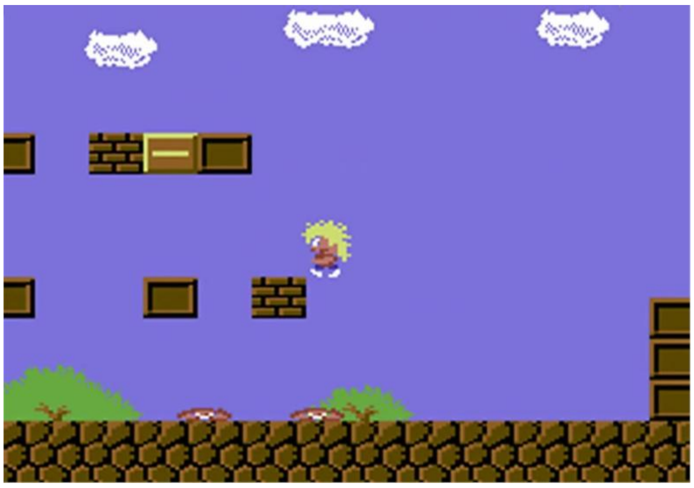
Figure 5. Giana Sisters - 1987. The protagonist was able to stand stably on the ground when the outermost pixel of his containment box had already collided. In the game, however, this had to be used in many ways.

In order to eliminate the error, a very simple but easily implementable solution was developed in practice. The size of the bounding box is not calculated pixel-by-pixel exactly for the image of the object, but its size is reduced to some extent. The figure below demonstrates this well:

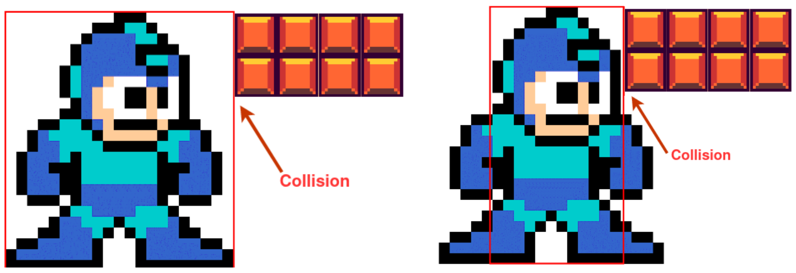

Figure 6. Reduced-sized bounding box allows more realistic collision detection

The size reduction can be done automatically with a custom algorithm or even manually. As already mentioned at the bounding circle, a separate description file is often used, in which it is possible to manually specify the size of each box.

4.2. Applying multiple bounding boxes

If we do not use objects in the graphic application that can be properly covered with a rectangle, then the normal AABB does not provide a satisfactory solution. Empty areas at the edges can lead to frequent false collisions, which can greatly degrade the gaming experience. A better solution to bounding boxes is to use not one, but even several bounding objects to cover the real area involved in the collision. The figure below shows an example of this:

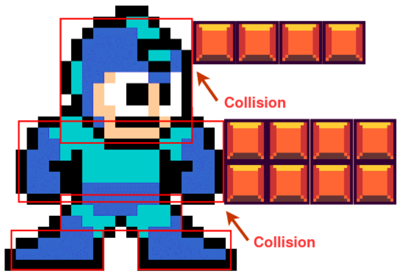

Figure 7. Using multiple bounding boxes greatly improves the efficiency of collision detection

In practice, this approach can be useful in many cases, and it can be important from several aspects. Its main purpose is to use it to make collision detection more precise. In cases when only one box alone is not enough to achieve sufficiently accurate collisions, the result can usually be improved with applying more boxes.

```
bool CheckCollision(CGameObject2D object1, CGameObject2D object2) {
    if (object1.isVisible() == false || object2.isVisible() == false)
        return false;

    vector <CBoundingBox2D> object1BBs = object1.getBoundingBoxes();
    vector <CBoundingBox2D> object2BBs = object2.getBoundingBoxes();

    for (int i = 0; i < object1BBs.size(); i++) {

        CBoundingBox2D object1Box = object1BBs[i];

        // Loop all the bounding boxes
        for (int j = 0; j < object2.size(); j++) {

           CBoundingBox2D object2Box =object2[j];
           bool result = CheckBoxOverLap(currentObjBox, objectCurrentObjBox);

            // Check overlapping case
            if (result == true) {
                return true;
            }
        }
    }

    return false;
}
```

4.3. Moving a bounding box

It is important to say a few words about the practical implementation of bounding boxes. In practice, the box-based collision test means that behind a game object we have an "invisible" box, which is constantly moved according to the translation of the game object. From a practical point of view, the development of any computer game requires the ability to display the bounding boxes at certain times during the development of the game. There are many practical reasons for this. Most of all, it offers developers a kind of debugging option when the game's engine is made in-house, rather than being used as an external component.

However, there is a small gap between moving the boxes and displaying them. While the rendering is done by the GPU according to the world coordinates of the boxes and the exact position is calculated in the vertex shader, any other calculation of the boxes (e.g. overlap test) is done on the CPU side. The collision test will only be accurate and functional if the position of the box drawn by the GPU matches the calculations performed on the CPU side.

When moving the object (translate, rotate, scale), the coordinate of the box must also be transformed. From the point of view of practical implementation, it is worth noting that two versions of an object's box should be stored: the original and the transformed version. It is worth keeping the original copy, because during any movement it is worth taking it as a basis and calculating the transformed version according to the movement parameters. And it is advisable to store the transformed version because you may need this box several times within a game cycle.

In case of translation, no serious problem occurs, since the box moves while maintaining its orientation and size. Its calculation is simple: we transform the coordinates of the original AABB to the new position. The case of the scaling transformation is also similar to the translating, with the difference that the center of the transformation must be taken into account here. In the case of the game engine, an important decision is where the origin of the object's local coordinate system [REF] should be. In many cases, this is the center of the object or one of the corner points of the object. This decision usually affects the box scale calculation logic. However, in the case of rotation, we encounter other difficulties.

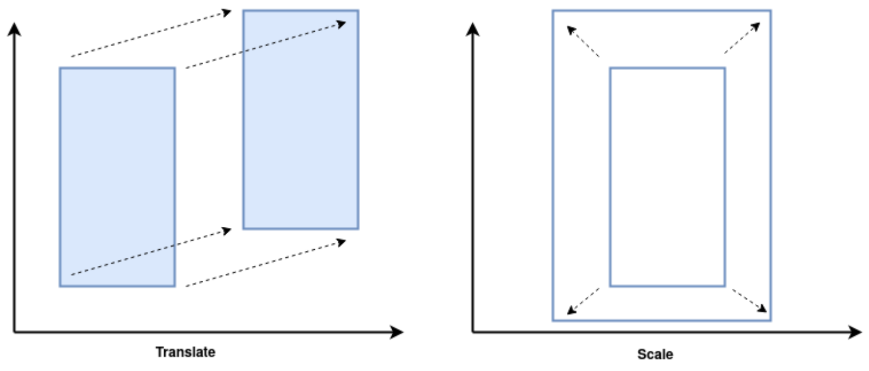

Figure 8. Translation and scaling of a bounding box

4.3.1. Rotate a bounding box

Based on what has been presented so far, we can consider it a natural way to rotate the boxes if the corner points of the box are rotated in the right direction, thus obtaining the rotated object. In practice, however, two approaches have emerged regarding the orientation of the box:

- **Axis-Aligned Bounding Boxes (AABB):** a rectangle with all edges parallel to a coordinate axis.
- **Oriented Bounding Box (OBB):** a rectangle that rotates with the rotation of the object.

The following figures illustrate the difference between the two solutions:

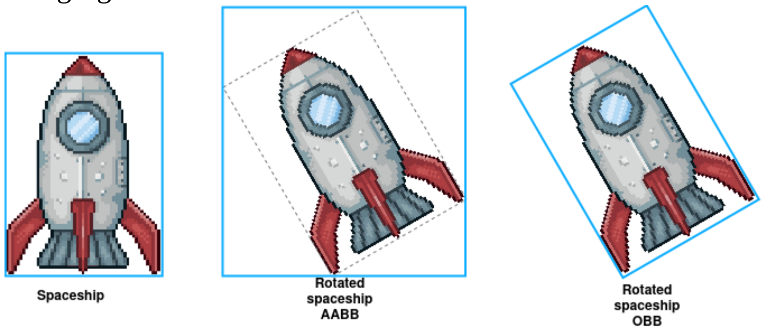

Figure 9. Difference between AABB and OBB

While OBB seems to be the natural solution for anyone, in practice AABB is used instead. The reason for this is that in the case of OBB, the overlap of two arbitrarily rotated boxes must be detected, which can be described with a mathematically more complicated algorithm. The implementation of AABB, the calculation of box overlap (collision test), the visibility test with the screen box (is it visible on the screen or not) is significantly simpler than with OBB. However, the disadvantage of the rotated AABB is obvious: during the rotation of the box, its original size changes, and thus the effectiveness of the collision detection can deteriorate to a large extent.

In the following, the procedure for rotating the AABB is presented. We will observe why the size of the box changes and how we get the box parallel to the coordinate axes again after rotation.

4.3.2. AABB rotation in practice

The rotation of an AABB box consists of three-steps:

1. Transforming the four points of a box according to the rotation angle
2. From the rotated points, we search the minimum and maximum points, i.e. the boundaries of the new box
3. Based on these points, we create a new box parallel to the coordinate axes

In the following, we present how the rotation of the AABB takes place in practice. During the solution, we need to rotate the 4 points of the box, and then determine AABB again based on the rotated points. The example algorithm rotates the box along the x-axis, its essence is briefly as follows:

```
// loop all the 4 points
for (unsigned int i = 0; i < AABB_POINTS; i++){
   // setup points as a vector
   CVector2 point(bbPoints[i].x,bbPoints[i].y);
   // rotate BB vector
   m_TransformationMatrix.rotate_x(&point, angle);
   bbPoints[i].x = point.x;
   bbPoints[i].y = point.y;
}
```

After that, the bounding box must be recreated to meet the AABB's requirements again. This is done with two functions:

```
// Search min and max points
```

```
searchMinMax();

// setup AABB box
setUpBBPoints();
```

The *searchMinMax()* function searches for the minimum and maximum points among the rotated points based on the x,y coordinates. After that, the *setUpBBPoints()* function determines the 4 new points of the box:

```
void setUpBBPoints()
{
    bbPoints[0].x = minpoint.x;
    bbPoints[0].y = minpoint.y;
    bbPoints[1].x = maxpoint.x;
    bbPoints[1].y = minpoint.y;
    bbPoints[2].x = maxpoint.x;
    bbPoints[2].y = maxpoint.y;
    bbPoints[3].x = minpoint.x;
    bbPoints[3].y = maxpoint.y;
}
```

It is important to note that the consequence of rotation is a change in the size of the box. The dashed line in Figure 9. indicates the "covered area" of the original object. Since the size of the box changed because of the rotation is in many cases no longer ideal for detecting collisions, therefore this type of rotation is not used in all cases. Perhaps the "error" can be eliminated if the size of the rotated box is reduced to some extent. In certain games, however, developers choose a different but simple solution. All rotated images of the game object (even up to 360 pieces) are created with an image editor and stored as a separate image. Thus, each such "phase" can have its own bounding object, which does not suffer from the change in the size of the box that occurred during the previous rotation. And during the gameplay, the image corresponding to the current rotation angle is displayed.

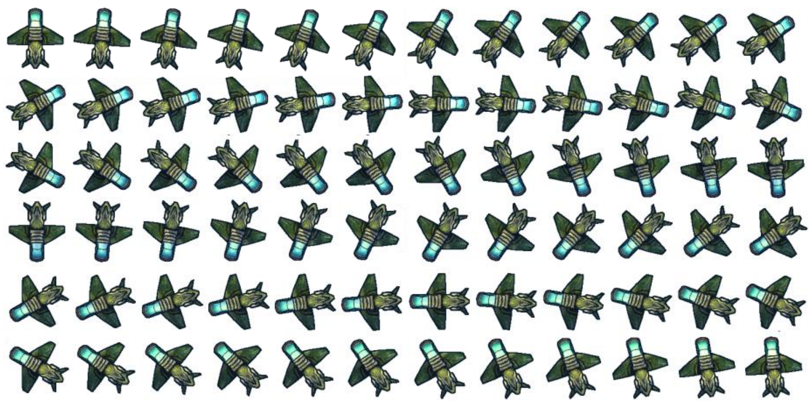

Figure 10. The rotated phases of the game object

4.4. Bounding circle and box collision test

However, we must not forget about a case that also occurs in practice when different types of bounding objects collide. For example, in a typical breakout game, the ball is covered by a bounding circle and the bricks by a bounding box.

Collision detection in this case will be a little more complicated than before. The logic of the algorithm is as follows: The closest point (P) to the circle on the bounding box must be found. If the distance between the point and the center of the circle is smaller than the radius of the circle, then a collision has occurred.

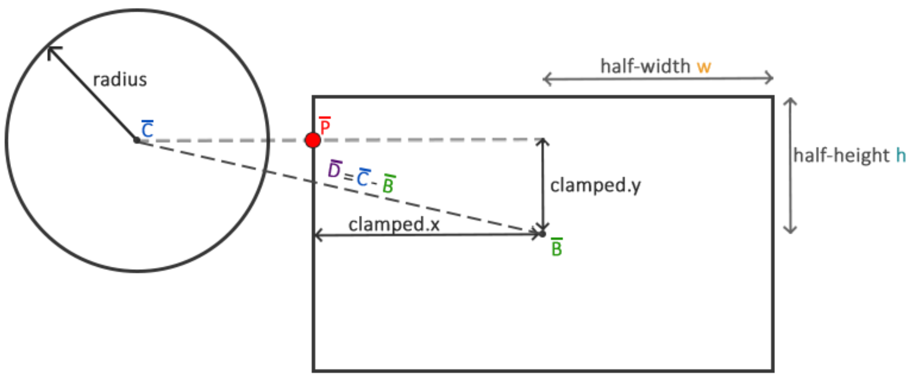

Figure 11. Geometric illustration of a Circle - AABB collision

The first step is to determine the distance vector (D) between the center of the circle (C) and the center of the box (B). This vector is then clamped according to the dimensions of the half of the box. Half of the box is the distance between the center of the box and the edges. The resulting point after clamping is always located somewhere on one edge of the box.

The clamp operation means that a given value is always adjusted to a set of value ranges. We can interpret it as follows:

```
float clamp(float value, float minValue, float maxValue) {
      return max(minValue, min(maxValue, value));
}
```

The point P after cutting is the point of AABB that is closest to the circle. To determine the collision, all we have to do is calculate the distance between the center of the circle (C) and P. For this we need the vector D, which is the difference between the vectors C and P. The length of vector D will give the distance. If this value is smaller than the radius of the circle, then there was a collision.

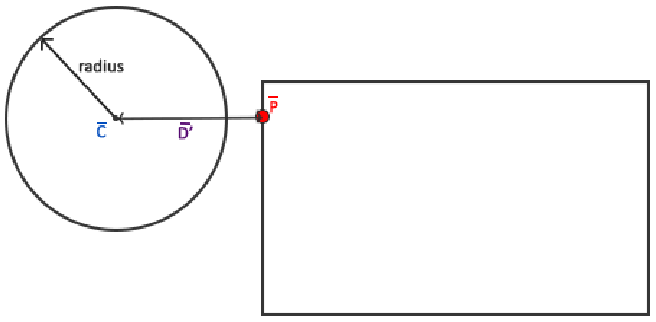

Figure 12. Geometric illustration of a Circle - AABB collision

The pseudo code of the collision detection:

```
checkAABB_Circle_collision(AABB, Circle)

        // Distance of the two center
        Vector2 d_vector = Circle.center - AABB.center;
        Vector2 aabb_half_extents(AABB.sizeX / 2.0, AABB.sizeY / 2.0);
        float clampedX = clamp(d_vector.x, -aabb_half_extents.x,
        aabb_half_extents.x);
        float clampedY = clamp(d_vector.y, -aabb_half_extents.y,
        aabb_half_extents.y);
        Vector2 closestPoint = AABB.center + Vector2(clampedX,clampedY);

        Vector2 new_d_vector = closestPoint - Circle.center;
        // Calculate the distance
        diff = sqrt((new_d_vector.x * new_d_vector.x) + (new_d_vector.y *
        new_d_vector.y))

        if (diff <= (Circle.radius)
                return true; // Hit

        return false; // No hit
}
```

5. Bounding polygon

In practice, of course, we can work with objects for which none of the solutions described so far provide adequate results. If even more accurate collision detection is required, it is advisable to use the bounding polygon approach. The bounding polygon is a convex or concave polygon consisting of arbitrary points, the purpose of which is to surround the game object with the best possible shape. The collision test is performed on this polygon.

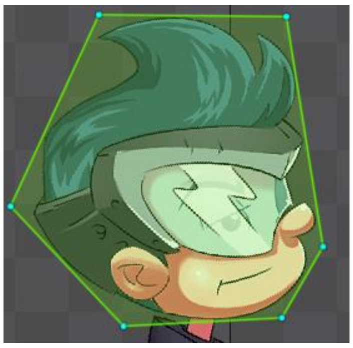

Figure 13. Application of a bounding polygon (Spine animation tool [14])

The polygon-based approach can effectively take the shape of the object. Nowadays, more and more game engines are starting to support this approach. Usually, a separate editing interface is provided for creating and modifying the polygon in the editor of the game engine (e.g. Unity, Cocos Creator).

Detecting the collision of two polygons is a more complex task. Several approaches are prevalent in the literature. One known solution is when the polygon is split into a set of triangles, and then the required type of test is performed on this set. The type of investigation may depend on the type of objects involved in the collision. If two polygons collide, we speak of a polygon-polygon collision test. Here, we can even apply triangle-triangle collision detection, or we can check whether a point of the given object's polygon is included in each triangle of the other object. In the following, we present a solution that is preferred in practice.

5.1. Separating Axis Theorem

The Separating Axis Theorem, or SAT, is a technique used to establish whether two convex shapes are intersecting. It can also be applied to identify the minimum penetration vector, a useful feature for physics simulation and other applications. This fast and versatile algorithm eliminates the need for collision detection code for each pair of shape types, ultimately reducing the amount of code needed and easing maintenance efforts.

SAT states that two convex shapes are not intersecting if and only if there exists at least one axis where the orthogonal projections of the shapes on this axis do not intersect.

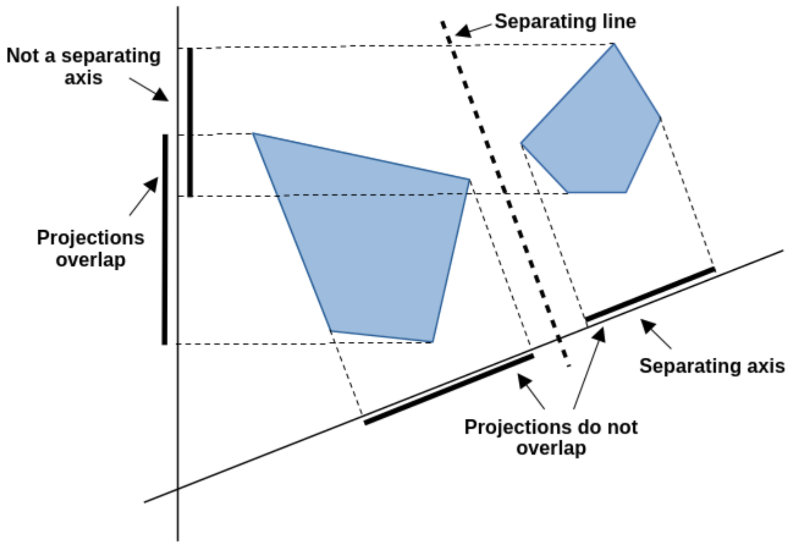

Figure 14. Operation of Separating Axis Theorem

So, if we can draw a line between two shapes without touching either one, they do not overlap. In a drawing, this is quite easy to do. In practice we can accomplish this by projecting the shapes onto an axis and checking for overlap. If we can find one axis where the projections don't overlap, then we can say that the two shapes don't collide. The projection of a two-dimensional object is a one dimensional "shadow". A line where the projections (shadows) of the shapes do not overlap is called a separation axis.

From a programming point of view, it would be too intensive to check every possible angle. Due to the nature of the polygons, there are only a few key angles we need to check. When dealing with 2-dimensional polygons, it is only necessary to examine the axes that are perpendicular to the edges of the shape. If the projections don't overlap in at least one of them, the shapes do not overlap. The above image shows only two of those axes, but that particular case has 9 of them (one for each edge).

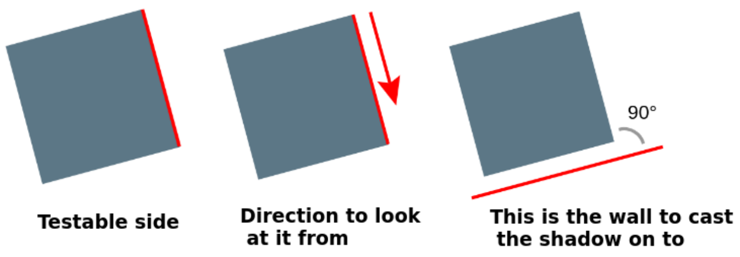
Testable side
Direction to look at it from
90°
This is the wall to cast the shadow on to

Figure 15. Object's side (edge) testing

<u>The logic of the algorithm can be summarized as follows:</u>

```
For each edge of both polygons:
  Find the perpendicular axis to the current edge.
    Loop through every point on the first polygon and project it onto the axis.
    (Keep track of the highest and lowest values found for this polygon).
    Do the same for the second polygon
    Check the projections for overlap.
    If the polygons don't intersect exit the loop
```

5.1.1. The projection

Projection can be accomplished by the following way: consider each edge as vector A, and the projection axis as vector B, as in the following figure [13]:

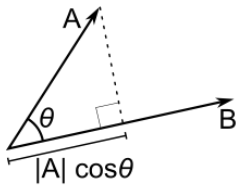

Figure 16. Object's side (edge) testing

There are two formula that can help to interpret this figure: the trigonometric definition of cosine and the geometric definition of the cross-product:

$$cos\theta = \frac{|projection\ of\ A\ onto\ B|}{|A|} \tag{1}$$

$$A \cdot B = |A||B|cos\theta \tag{2}$$

These equations can be combined to get the projection formula:

$$projection\ of\ A\ onto\ B = A \cdot \overline{B}, where\ \overline{B} \text{ is a unit vector in the direction of B}$$

Thus, given two vectors - one for the axis (which needs to be a unit vector), and one to a corner of our collision polygon, we can project the corner onto the axis. If we do this for all corners, we can find the minimum and maximum projection from the polygon [13].

A helper method for finding min and max values:

```
private MinMax FindMaxMinProjection(BoundingPolygon poly, Vector2 axis)
{
    var projection = Dot(poly.Corners[0], axis);
    var min, max = projection;

    for (var i = 1; i < poly.Corners.length; i++)
    {
        projection = Dot(poly.Corners[i], axis);
        max = max > projection ? max : projection;
        min = min < projection ? min : projection;
    }
    return new MinMax(min, max);
}
```

If we determine the minimum and maximum projection for both shapes, we can see if they overlap or not.

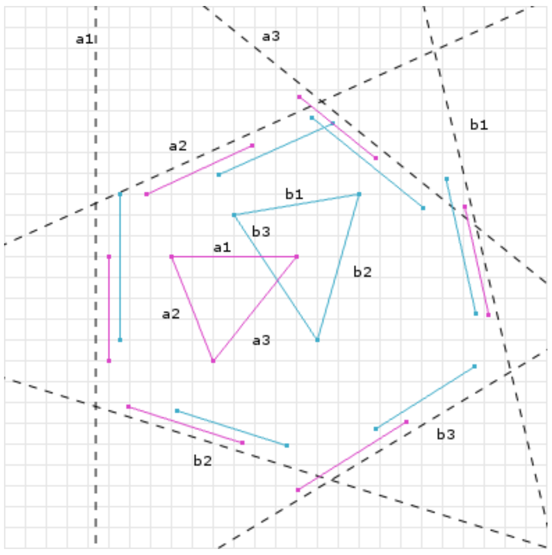

Figure 17. Two Convex Shapes Intersection testing with many sides check [15]

If there is no overlap, then we have found a separating axis, and can terminate the search. Geometrically, it can be shown that the bare minimum we need to test is an axis parallel to each edge normal of the polygon - that is, an axis at a right angle to the polygon's edge. Each edge has two normal, a left and right:

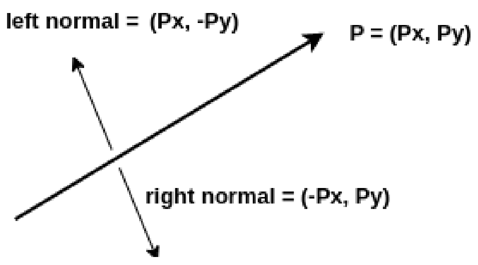

Figure 18. Edge normal

An edge normal is a unit vector perpendicular to the edge vector (a vector along the edge) in 2D. It can be calculated by swapping the x and y components and negating one of them. One of these normal will face out of and the other will face in the polygon depending on the order of the points (clockwise / counter-clockwise). Both directions can work if we keep the direction. So, normal are calculated by iterating over our points and creating vectors to represent each edge, then calculating a perpendicular vector to that edge.

```
public struct BoundingPolygon {
      Vector2[] mCorners;
      Vector2 mCenter;
      Vector2[] mNormals;


  public BoundingPolygon(Vector2 _center, Vector2 _corners) {
        mCenter = _center;
        mCorners = _corners;
        var normals = new HashSet<Vector2>();

        var edge = Corners[mCorners.length - 1] - mCorners[0];
        var perp = new Vector2(edge.Y, -edge.X);
        perp.Normalize();
        normals.Add(perp);
        for (var i = 1; i < mCorners.length; i++) {
            edge = mCorners[i] - mCorners[i - 1];
            perp = new Vector2(edge.Y, -edge.X);
            perp.Normalize();
            normals.Add(perp);
        }
        mNormals = normals.ToArray();
    }
}
```

To detect a collision between two *BoundingPolygons*, we iterate over their combined normals, generating the *MinMax* of each and testing it for an overlap. Implemented sample code:

```
public bool Collides(BoundingPolygon p1, BoundingPolygon p2) {
    foreach(var normal in p1.mNormals){
        var mm1 = FindMaxMinProjection(p1, normal);
        var mm2 = FindMaxMinProjection(p2, normal);
        if (mm1.Max < mm2.Min || mm2.Max < mm1.Min) return false;
    }
    foreach (var normal in p2.mNormals) {
        var mm1 = FindMaxMinProjection(p1, normal);
        var mm2 = FindMaxMinProjection(p2, normal);
        if (mm1.Max < mm2.Min || mm2.Max < mm1.Min) return false;
    }
    return true;
}
```

SAT may test many axes for overlap, however, the first axis where the projections are not overlapping, the algorithm can immediately exit, determining that the shapes are not intersecting. Because of this early exit, SAT is ideal for applications that have many objects but few collisions (games, simulations, etc.).

6. Conclusion

Collision detection is an integral part of today's computer visualization, be it a game or any application where interaction is required. Although today's software is more and more complex and works with more and more moving objects, collisions implemented in the background are detected using well-known professional solutions. Although the performance of hardware has increased a lot in recent years, pixel-level collision detection still does not offer an effective alternative due to its high computational demand, instead, algorithms based on bounding objects dominate. These solutions have the error of false collision detection. Therefore, a modern graphics engine usually implements several solutions to offer the possibility of some level of combination of methods to achieve the right result. This paper reviewed the most important approaches from a practical point of view. Examining the advantages and disadvantages of the presented methods, they provide an opportunity to design and create an effective collision system.

References

[1] Wil van der Aalst: Process Mining: Data Science in Action, Springer Berlin, Heidelberg, Second Edition, 15 April 2016

[2] João M. P. Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz: Embedded Computing for High Performance, Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, pp. 17-56, 2017

[3] Charles Kelly: Programming 2D Games, A K Peters/CRC Press; 1st edition, 2012.

[4] Jason Gregory: Game Engine Architecture, A K Peters/CRC Press; 3rd edition, 2018.

[5] MARÍN-LORA, C., CHOVER, M., REBOLLO, C., yREMOLAR, I.: A game development environment to make 2D games, Communication Papers, Vol.9 – No18, pp. 7/23, 2020
[6] Tomas Akenine-Moller, Eric Haines, Naty Hoffman: Real-Time Rendering, 3rd Edition, A K Peters/CRC Press; 2008.
[7] Lazaridis, L., Papatsimouli, M., Kollias, KF., Sarigiannidis, P., Fragulis, G.F.: Hitboxes: A Survey About Collision Detection in Video Games. In: Fang, X. (eds) HCI in Games: Experience Design and Game Mechanics. HCII 2021. Lecture Notes in Computer Science(), vol 12789. Springer, Cham. https://doi.org/10.1007/978-3-030-77277-2_24, 2021.
[8] K. Guo and J. Xia: An Improved Algorithm of Collision Detection in 2D Grapple Games, 2010 Third International Symposium on Intelligent Information Technology and Security Informatics, Jian, China, pp. 328-331, 2010, https://doi.org/10.1109/IITSI.2010.176 .
[9] Benjamin Rodrigue: Algorithmic and Architectural Gaming Design: Implementation and Development, Chapter 10: Collision Detection in Video Games, DOI: 10.4018/978-1-4666-1634-9.ch010, 2012.
[10] Thomas Schwarzl: 2D Game Collision Detection: An introduction to clashing geometry in games, Createspace Independent Publishing Platform, 2012.
[11] Intro to Collision Detection: Collision Detection Basics, http://www.kilobolt.com/collision-detection-basics, 2023.
[12] Learn OpenGL - Collision detection: http://learnopengl.com/#!In-Practice/2D-Game/Collisions/Collision-detection, 2023
[13] Nathan Bean, Foundations of Game Programming, CIS 580 Textbook, Kansas State University, 2021.
[14] EsotericsSoftware: Spine - 2D animation tool, http://esotericsoftware.com/, 2023
[15] SAT (Separating Axis Theorem), https://dyn4j.org/2010/01/sat/, 2023