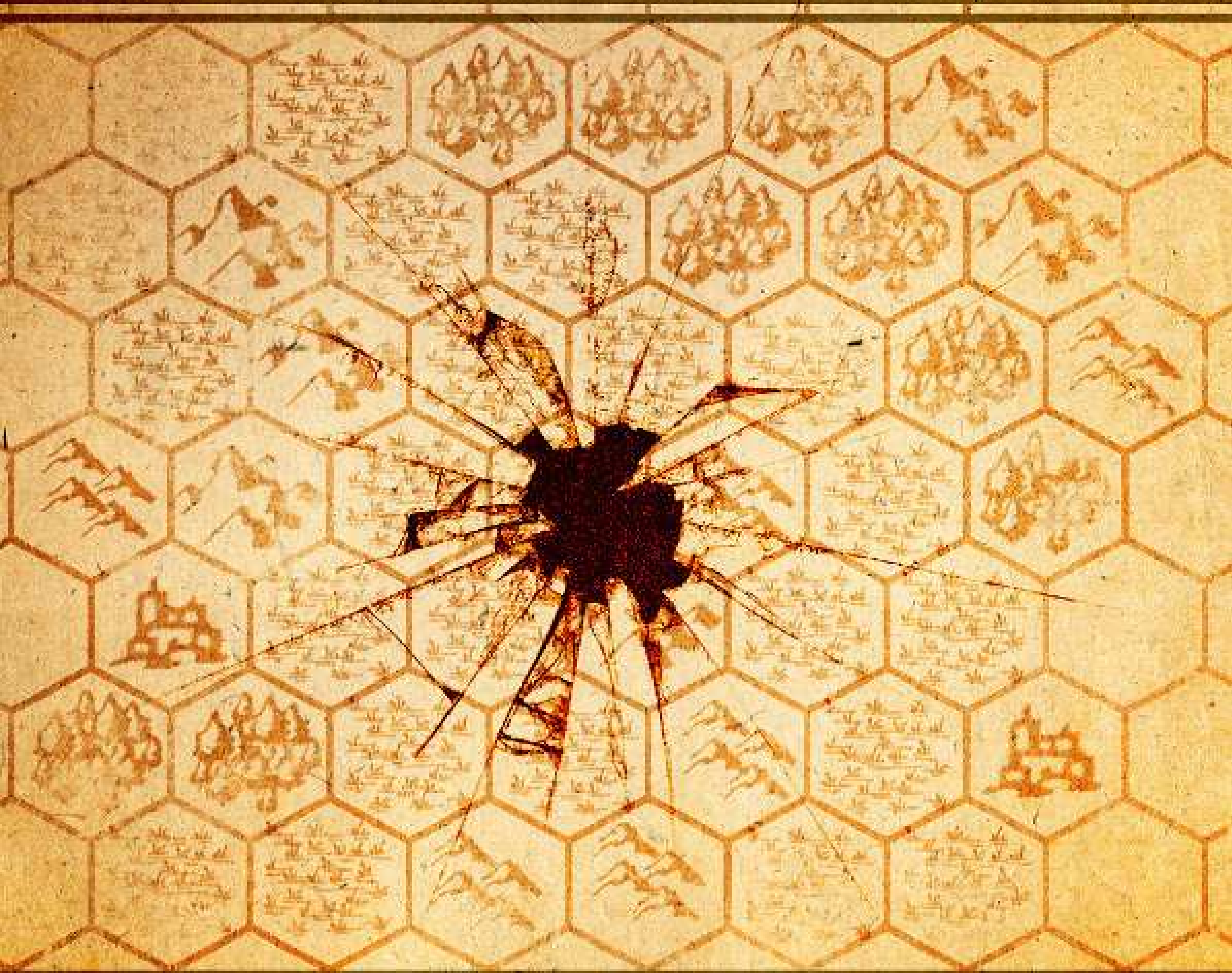
2D Game Collision Detection

An introduction to clashing geometry in games



Thomas Schwarzl

Table of Contents

Introduction **1**
- Features of This Book 1
- You Will Need 1
- You Won't Need 2
- Share Your Thoughts 2

Atoms of Geometry: Vectors **3**
- What Vectors Are 3
- Additions 4
- Scaling 7
- Length 9
- Null Vector 11
- Unit Vector 11
- Rotation 12
- Dot Product 15
- Projection 18

Shapes **20**
- Line 20
- Line Segment 22
- Circle 23
- Rectangle 24
- Oriented Rectangle 25

Collision Detection **26**
- Rectangle-Rectangle Collision 27
- Circle-Circle Collision 28
- Point-Point Collision 29
- Line-Line Collision 30
- Line-Segment-Line-Segment Collision 32
- Oriented-Rectangle-Oriented-Rectangle Collision 35
- Circle-Point Collision 38
- Circle-Line Collision 39
- Circle-Line-Segment Collision 40
- Circle-Rectangle Collision 41
- Circle-Oriented-Rectangle Collision 42
- Rectangle-Point Collision 43
- Rectangle-Line Collision 44
- Rectangle-Line-Segment Collision 45
- Rectangle-Oriented-Rectangle Collision 46

Point-Line Collision 50
Point-Line-Segment Collision 51
Point-Oriented-Rectangle Collision 52
Line-Line-Segment Collision 53
Line-Oriented-Rectangle Collision 54
Line-Segment-Oriented-Rectangle Collision 55

Bounding Shapes 56

Bounding Rectangle 57
Bounding Circle 57
Circle or Rectangle? 58

Shape Grouping 59

Bounded Shape Groups 59
The Code 60

Shapes in Motion 63

The Tunneling Problem 63
Linear Impact Search 64
Binary Impact Search 65
When Both Objects Move 69

Optimization Tricks 70

Abstraction is King 70
Size Matters 71
Rotating by Right Angles 71
Pass Arguments by Reference 72
Avoid Square Root 72
Short-Circuit Coding 73
Avoid Superfluous Tests 74
Level of Detail 75

Appendix: The Code 76

About the Author 77

Introduction

Are you curious how 2D collision detection in games works? If so this book is made for you.

In case you don't know what collision detection is: it's the determination of whether objects simulated in programs collide. This is a basic feature of computer games, e.g. for determining shot impacts, finding out which enemies are covered by lines of sight, recognizing collisions of race cars or simply checking if the mouse cursor floats above a button.

The book is written for game developers but is suited for other coder species as well.

Features of This Book

This book was written with the following intentions in mind:

- be aimed at beginners,
- use successive knowledge building,
- leverage ´a picture paints a thousand words´,
- provide working code,
- allow it to also function as a reference book and
- enable fast navigation by cross-linking

This book has less than 100 pages. That's not much for a textbook. Nonetheless, that's intentional. Serving up the necessary information on a minimum of pages is better than throwing a 500+ page tome at you. Brevity is the soul of wit.

You Will Need

... knowledge in basic procedural programming. All code is written in the language C. C is the

"mother" of modern imperative programming languages. So the language choice for this book was a no-brainer. If you're more familiar with C++, Java, Objective-C or C# you won't have any problems understanding what's going on in this book.

The code was written for comprehensibility. Some details were simplified or left out to get short and understandable code. Therefore the book's code may not be 100% correct C code.

Further there are no optimizations or fancy tricks in the code. That would compromise understandability.

Check out the appendix, which provides all code from this book as a download.

You Won't Need

... an academic degree. As long as you understand addition, multiplication and can read equations you should not encounter any problems throughout this book.

Oh, wait! You'll need to know what a square root is.

And what's *PI*.

I'm afraid elementary school children are out. Sorry.

Share Your Thoughts

If you have any questions about the book, collision detection or programming in general just drop me a line. Critique, praise and professional curses are welcome as well:

thomas@collisiondetection2d.net

I'm looking forward to hearing from you.

Atoms of Geometry: Vectors

Game objects need physical representations. Games use diverse geometric shapes for this. To find out if two objects collide we just have to check if their shapes intersect.

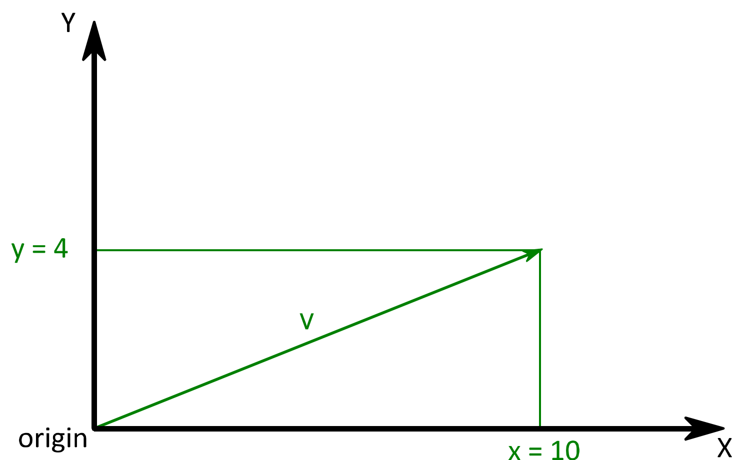
In computer games shapes usually get described by vectors. They are the building blocks for shapes and collision detection. So we first have to understand vectors and what we can do with them before we can tackle shapes and collision detection.

What Vectors Are

In 2D space a vector is simply a displacement in two dimensions. Vectors have length and direction but no position. A simple definition for 2-dimensional vectors is:

> *A 2D vector describes a straight movement in 2D space.*

The two dimensions are called X and Y. The following illustration and code shows exemplary 2D vector *v*:



```
typedef struct
{
    float x;
    float y;
} Vector2D;

Vector2D v = {10, 4};
```

The black arrows represent the coordinate system aka 2D space. The horizontal arrow illustrates the X-axis of the coordinate system, the vertical one illustrates axis Y. Their shared starting point is called the *origin*. The position of the *origin* is always *{0, 0}*.

> We will use notation *{x, y}* for vectors throughout the whole book. It's the same notation used for initializing vectors in code. Examples for this notation are *{10, 4}*, *{-208, 13}* or *{0, -47.13}*.

Vector *v* goes from the *origin* 10 units along axis X and 4 units along axis Y. Using our vector notation:

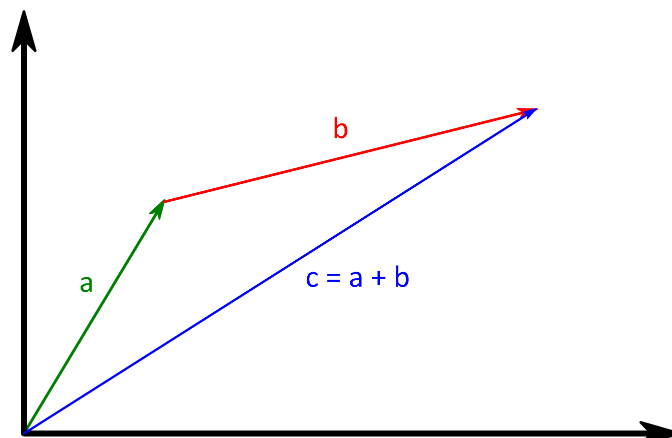
$$v = \{10, 4\}$$

This is the very same expression you can find in the code example above.

The vectors *origin* and *v* can also be seen as points in the coordinate system. The words *point* and *vector* are interchangeable in this case.

Additions

Vector addition can be imagined as chaining vectors. As mentioned in the preceding section, a vector is a displacement in 2D space. Let's assume we have point *a*, add vector *b* and get the resulting point *c*:

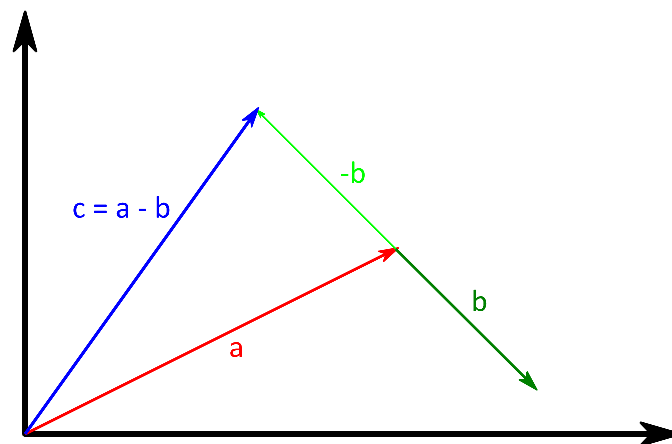


```
Vector2D add_vector(Vector2D a, Vector2D b)
{
    Vector2D r;
    r.x = a.x + b.x;
    r.y = a.y + b.y;
    return r;
}

Vector2D a = {3, 5};
Vector2D b = {8, 2};

Vector2D c = add_vector(a, b);
```

Vector *c* points to the position which vector *a* points to after it is displaced by vector *b*. This displacement is known as vector addition or, in math tongue, vector translation.

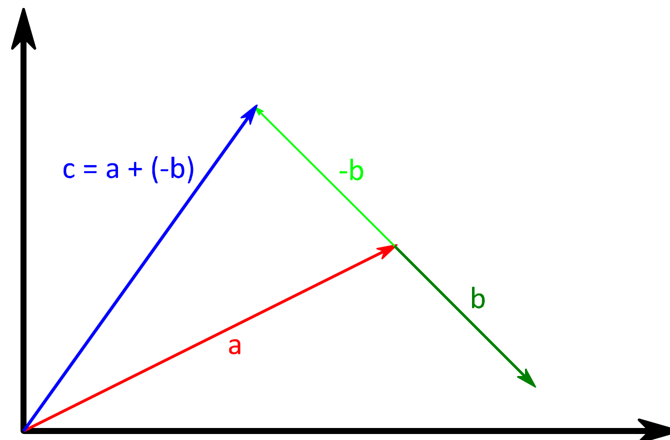
Now that we know how vector addition works, what about vector subtraction?

Subtraction is as simple as addition:




```
Vector2D subtract_vector(Vector2D a, Vector2D b)
{
    Vector2D r;
    r.x = a.x - b.x;
    r.y = a.y - b.y;
    return r;
}

Vector2D a = {7, 4};
Vector2D b = {3, -3};

Vector2D c = subtract_vector(a, b);
```

Subtraction can also be seen as adding a negated vector. In our case it would be adding negative *b* to *a:*




```
typedef enum { no = 0, yes = 1 } Bool;

Vector2D negate_vector(Vector2D v)
{
    Vector2D n;
    n.x = -v.x;
    n.y = -v.y;
    return n;
}

Bool equal_floats(float a, float b)
{
    float threshold = 1.0f / 8192.0f;
    return fabsf(a - b) < threshold;
}

void assert_equal_vectors(Vector2D a, Vector2D b)
{
    assert(equal_floats(a.x, b.x));
    assert(equal_floats(a.y, b.y));
}

Vector2D a = {7, 4};
Vector2D b = {3, -3};

Vector2D c = add_vector(a, negate_vector(b));

assert_equal_vectors(c, subtract_vector(a, b));
```

This code needs a little bit of explanation. First data type *Bool* is defined. It has just two possible values: *yes* and *no*. Any logical statement will return one of these two values.

> If you're familiar with C you may ask: "Why not use well known *true* and *false*?". The terms *yes* and *no* were adopted from Objective-C because they are more readable.

Function *negate_vector()* should be self-explanatory: it takes a vector and returns it pointing in the opposite direction.

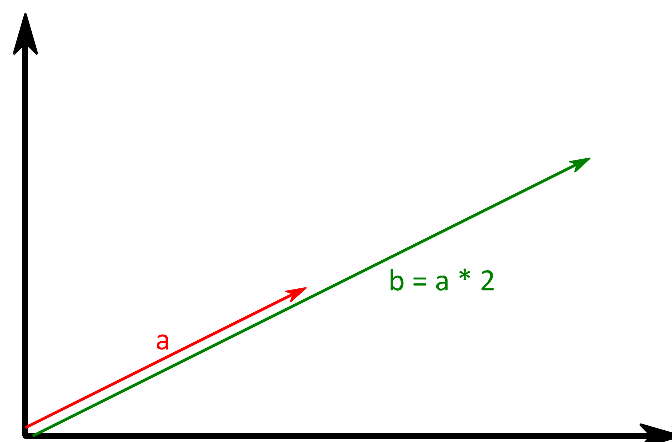
Function *equal_floats()* and *assert_equal_vectors()* are test functions. The former returns *yes* when the two parameters are equal. The latter checks if two vectors are equal. If not, function *assert()* is used to signal an error.

> Functions *equal_floats()*, *assert_equal_vectors()* and *assert()* will often be used throughout the book. The basic function *assert()* - which experienced coders surely know – takes the result of an assertion as a parameter. If the assertion is true the function does nothing. If it's wrong the function signals a problem, e.g. showing a small message box stating an error message. Function *assert()* is just used to verify that a result is as expected.
>
> Function *equal_floats()* takes two float values and returns *yes* if they are **nearly** equal. You may be puzzled why we can't just write *a == b*. The problem with floats is that they suffer from rounding errors. Tiny deviations already break code like *a == b*. Therefore we have to be more tolerant and test the difference of *a* and *b* to be below a certain threshold. For this we use the function *fabsf()* - a C standard function - which returns its parameter's absolute value. In other words it returns the parameter without its sign.

Finally *a - b = a + (-b)*. Vector addition (which includes subtraction) works the same way as addition of simple numbers.

Scaling

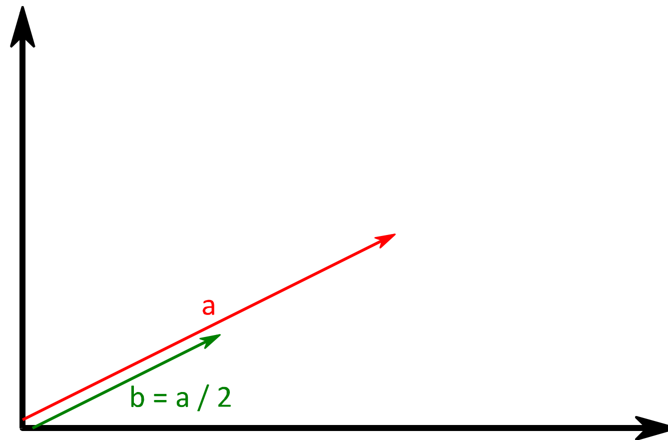
Scaling a vector is done by multiplying the vector's two values *x* and *y* with a number. In this case the number is called a scalar:



```
Vector2D multiply_vector(Vector2D v, float scalar)
{
    Vector2D r;
    r.x = v.x * scalar;
    r.y = v.y * scalar;
    return r;
}

Vector2D a = {6, 3};
Vector2D b = multiply_vector(a, 2);

assert(equal_floats(b.x, 12));
assert(equal_floats(b.y, 6));
```

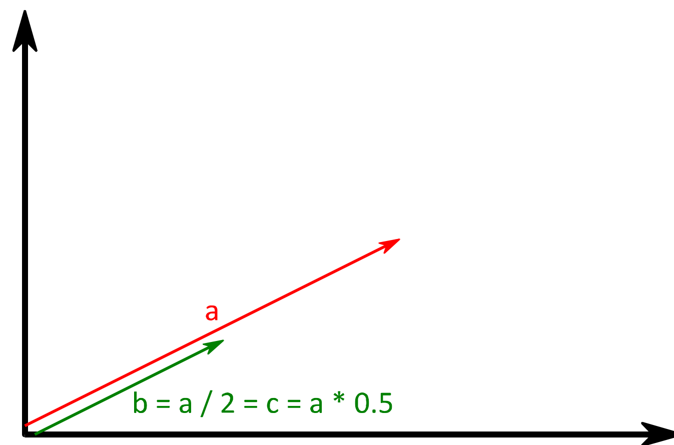
Scaling can change only a vector's length and not its direction.

Where there is multiplication there is also division. As you may guess it's about dividing the vector's values by a number, in this case called a divisor:




```
Vector2D divide_vector(Vector2D v, float divisor)
{
    Vector2D r;
    r.x = v.x / divisor;
    r.y = v.y / divisor;
    return r;
}

Vector2D a = {8, 4};
Vector2D b = divide_vector(a, 2);

assert(equal_floats(b.x, 4));
assert(equal_floats(b.y, 2));
```

Division can also be seen as a multiplication by *1/divisor:*




```
Vector2D a = {8, 4};
float divisor = 2;

Vector2D b = divide_vector(a, divisor);
Vector2D c = multiply_vector(a, 1 / divisor);

assert_equal_vectors(b, c);
```

Finally, the following scenarios exist in vector scaling:

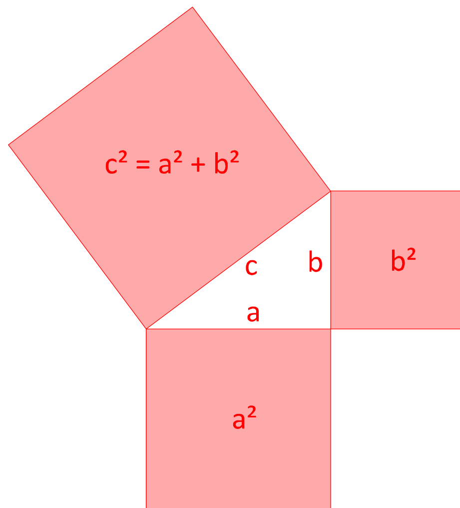
- $scalar > 1$: the vector gets longer
- $scalar = 1$: the vector stays the same
- $0 < scalar$ and $scalar < 1$: the vector shrinks
- $scalar = 0$: the vector becomes *{0, 0}*, it points nowhere!
- $scalar < 0$: the vector points in the opposite direction

Length

The length of a vector is the distance from its origin to its tip.

The Pythagorean theorem is well suited to calculating a vector's length. It says that in a triangle with side lengths a, b and c and a right angle between a and b the following equation is always true:

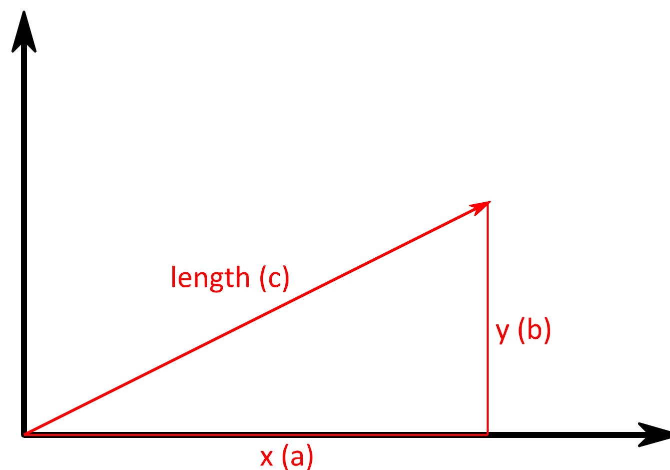
$$c^2 = a^2 + b^2$$



When we use this theorem for vectors we can imagine *a* as *x* and *b* as *y*. This way *c* is equal to the vector's length. Rearranging the equation for *c* gives us this formula:

$$c = squareroot(a^2 + b^2) = length = squareroot(x^2 + y^2)$$

We just have to write this equation as code:




```
float vector_length(Vector2D v)
{
    return sqrtf(v.x * v.x + v.y * v.y);
}

float a = 10;
float b = 5;
Vector2D v = {a, b};
float c = vector_length(v);
```

Function *sqrtf()* is a C standard library function that computes the square root of a value.

Null Vector

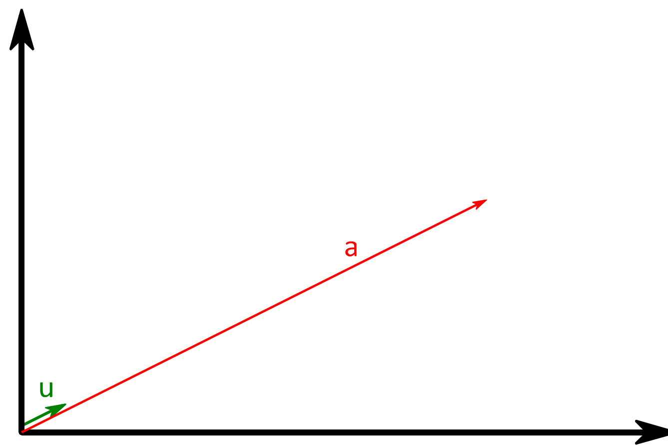
The null vector is *{0, 0}*. Logically it has a length of zero.

End of story.

Unit Vector

A unit vector is any vector with a length of 1. Examples would be *{0, 1}*, *{-1, 0}* or *{0.7071, -0.7071}*.

You get unit vector *u* from vector *v* by simply dividing *v* by its length. Both vectors *u* and *v* point in the same direction but (may) have different lengths:




```
Vector2D unit_vector(Vector2D v)
{
    float length = vector_length(v);
    if(0 < length)
        return divide_vector(v, length);
    return v;
}

Vector2D a = {10, 5};
assert(1 < vector_length(a));

Vector2D u = unit_vector(a);
assert(equal_floats(1, vector_length(u)));
```

There is a special case: the null vector *{0, 0}*. The length of the null vector is zero so calculating its unit vector would include a division by zero, which is undefined. In this case the result is the null vector itself.

left out pages

Get the full book from Amazon:

www.amazon.com

www.amazon.co.uk

www.amazon.de

Get the full book from the author:

www.collisiondetection2d.net

Use coupon OFF25 to get the book 25% off from the author:

www.collisiondetection2d.net/priceoff25

Shapes

After getting used to vector math it's time for the next level: geometrical shapes and their definitions.

We will investigate the following shapes in this book:

- lines,
- line segments,
- circles,
- axis-aligned rectangles and
- oriented rectangles

This chapter presents these different shapes and how to describe them in code. The next chapter will show you how to check them for collisions.

Line

In collision detection lines are always straight. When they are curvy they are, well, curves.

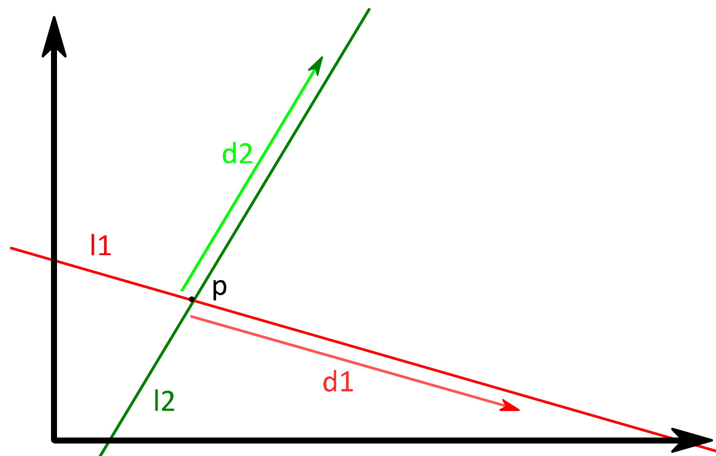
A possible definition of line:

> *A line is the shortest connection of two points with infinite distance.*

Huh? What does this mean?

Lines are straight and endless. Usually we consider a line to start at some point and stop at an other point. In geometry this is called a line segment. However lines extend infinitely in both directions.

In this book's code a line will be defined as a base point with a direction. The line goes through the point, infinitely heading in both the direction and its opposite direction.

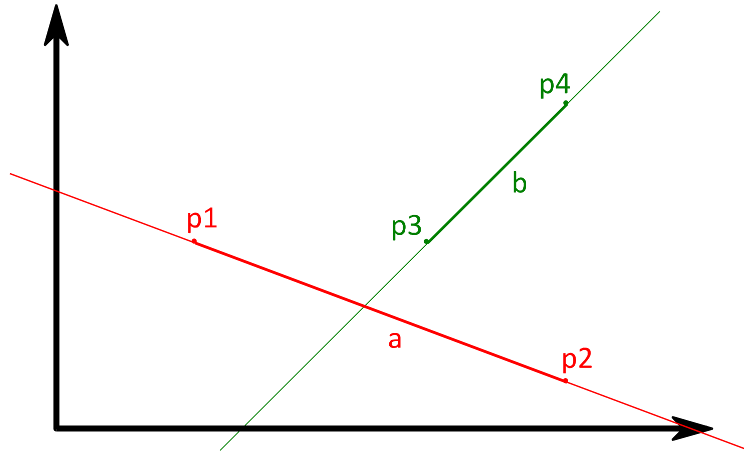



```
typedef struct
{
    Vector2D base;
    Vector2D direction;
} Line;

Vector2D p = {3, 3};
Vector2D d1 = {7, -2};
Vector2D d2 = {3, 5};

Line l1 = {p, d1};
Line l2 = {p, d2};
```

Because *p* is the base point for both lines *l1* and *l2* they cross exactly at *p*. The upcoming section Line-Line Collision explains how to determine whether two lines intersect.

Line Segment

Lines are straight and endless. But in collision detection we rather need lines to start at one point and end at an other point. This is called a line segment.

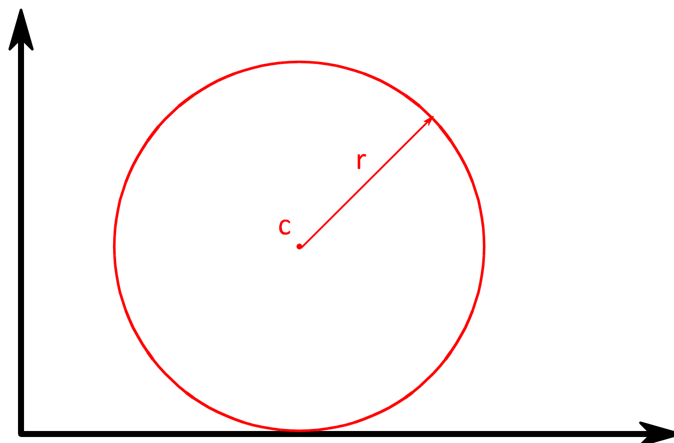



```
typedef struct
{
    Vector2D point1;
    Vector2D point2;
} Segment;

Vector2D p1 = {3, 4};
Vector2D p2 = {11, 1};
Vector2D p3 = {8, 4};
Vector2D p4 = {11, 7};

Segment a = {p1, p2};
Segment b = {p3, p4};
```

The two end points define the line segment. The code shows that the structure is named just *Segment* instead of *LineSegment*. The simple reason: the author is a lazy coder.

Circle

Circles have a center point and a radius.

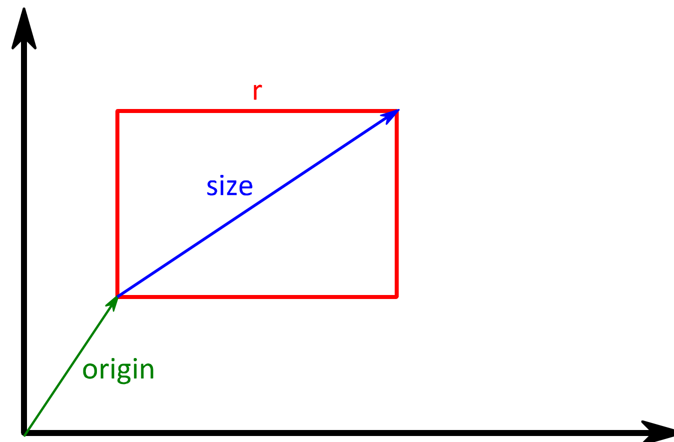



```
typedef struct
{
    Vector2D center;
    float radius;
} Circle;

Vector2D c = {6, 4};
float r = 4;

Circle a = {c, r};
```

The definition is as simple as the intersection algorithm. You will see in section Circle-Circle Collision.

Rectangle

A rectangle is a shape with four sides where each corner has a right angle.

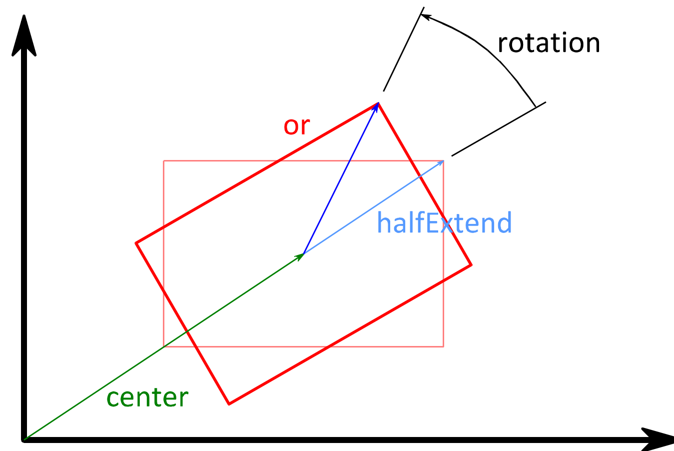



```
typedef struct
{
    Vector2D origin;
    Vector2D size;
} Rectangle;

Vector2D origin = {2, 3};
Vector2D size = {6, 4};

Rectangle r = {origin, size};
```

Usually rectangles are meant to have sides parallel to the coordinate system axes. So we are talking about axis-aligned rectangles.

Oriented Rectangle

Oriented rectangles have, like axis-aligned ones, position and size. But they also have a rotation.




```
typedef struct
{
    Vector2D center;
    Vector2D halfExtend;
    float rotation;
} OrientedRectangle;

Vector2D center = {6, 4};
Vector2D halfExtend = {3, 2};
float rotation = 30;

OrientedRectangle or = {center, halfExtend, rotation};
```

For describing oriented rectangles we use the center instead of the origin and the rectangle's half extend (= half size) instead of the size. This pays off when we have to check them for intersections.

Collision Detection

Finally we've arrived at chapter Collision Detection, the heart of this book. Now we get down to the nitty-gritty.

Collision detection answers a simple question:

Do two shapes intersect?

In its simplest form collision detection answers this question with just yes or no. The next question would be:

How/where do the shapes intersect?

This book focuses on the yes/no answers. That's enough for many 2D games.

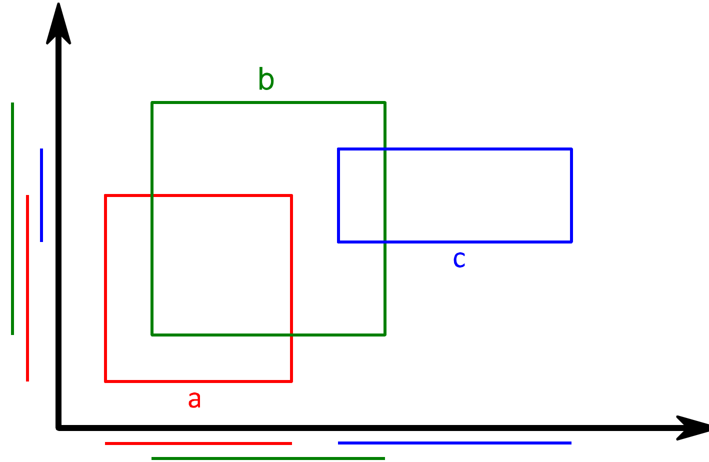
Because there are different types of shapes, collision detection needs special functions for each possible pairing. We have 6 different shapes (including points) so we need 21 functions to cover all possible combinations.

First the homogeneous collision checks get explained, for example circle-circle, line-line, etc. Then we'll have a look at all the heterogeneous collisions like rectangle-point, circle-line, etc.

Fasten your seatbelt! The mental roller coaster ride starts now.

Rectangle-Rectangle Collision

Collision detection for axis-aligned rectangles is really easy. We just have to check for overlaps in both dimensions.

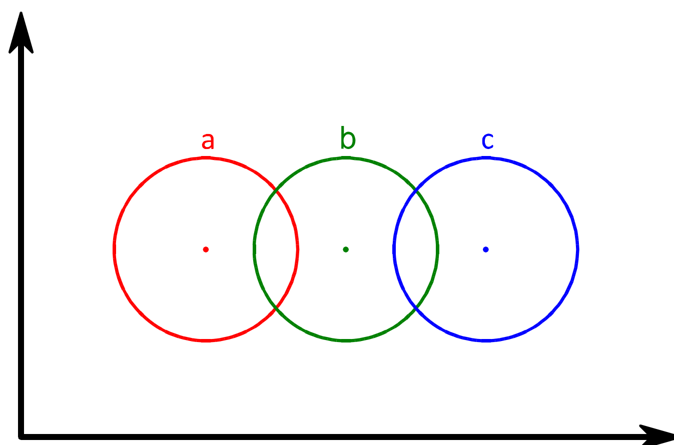



```
Bool overlapping(float minA, float maxA, float minB, float maxB)
{
    return minB <= maxA && minA <= maxB;
}

Bool rectangles_collide(Rectangle a, Rectangle b)
{
    float aLeft = a.origin.x;
    float aRight = aLeft + a.size.x;
    float bLeft = b.origin.x;
    float bRight = bLeft + b.size.x;

    float aBottom = a.origin.y;
    float aTop = aBottom + a.size.y;
    float bBottom = b.origin.y;
    float bTop = bBottom + b.size.y;

    return overlapping(aLeft, aRight, bLeft, bRight)
           &&
           overlapping(aBottom, aTop, bBottom, bTop);
}

Rectangle a = {{1, 1}, {4, 4}};
Rectangle b = {{2, 2}, {5, 5}};
Rectangle c = {{6, 4}, {4, 2}};

assert(yes == rectangles_collide(a, b));
assert(yes == rectangles_collide(b, c));
assert(no == rectangles_collide(a, c));
```

You can see the rectangle projections drawn along both coordinate system axes. The horizontal projections of *a* and *b* overlap. So do their vertical projections. Therefore rectangles *a* and *b* intersect. The same is true for *b* and *c*. Rectangles *a* and *c* overlap vertically but not horizontally. So these two rectangles can't intersect.

Circle-Circle Collision

Two circles intersect when the distance between their centers is less than the sum of their radii.

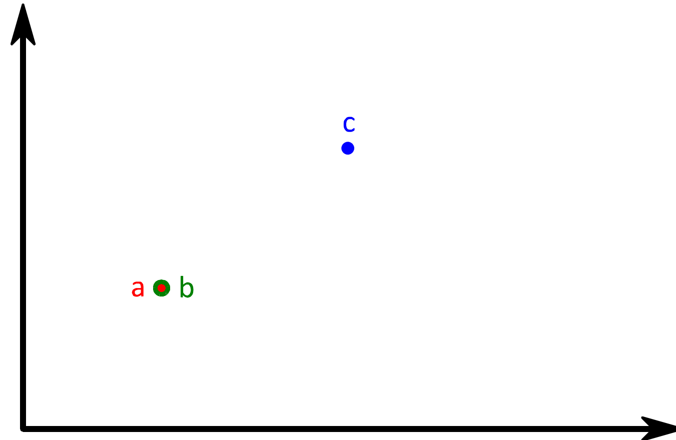



```
Bool circles_collide(Circle a, Circle b)
{
    float radiusSum = a.radius + b.radius;
    Vector2D distance = subtract_vector(a.center, b.center);
    return vector_length(distance) <= radiusSum;
}

Circle a = {{4, 4}, 2};
Circle b = {{7, 4}, 2};
Circle c = {{10, 4}, 2};

assert(yes == circles_collide(a, b));
assert(yes == circles_collide(b, c));
assert(no == circles_collide(a, c));
```

The centers of circle *a* and *b* are 3 units apart. Both circles have a radius of 2 units. Because the distance is less than the sum of the radii ($3 < 4$) *a* and *b* intersect. The same is true for *b* and *c*. The center distance of *a* and *c* measures 6 units. Too far apart for *a* and *c*'s radius 2 to intersect ($6 > 4$).

Point-Point Collision

Determining point collision is trivial: two points intersect when they have equal coordinates.

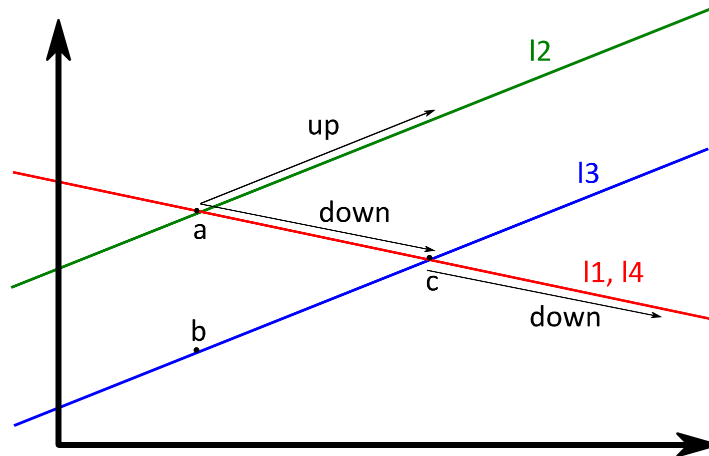



```
Bool points_collide(Vector2D a, Vector2D b)
{
    return equal_floats(a.x, b.x) && equal_floats(a.y, b.y);
}

Vector2D a = {2, 3};
Vector2D b = {2, 3};
Vector2D c = {3, 4};

assert(yes == points_collide(a, b));
assert(no == points_collide(a, c));
assert(no == points_collide(b, c));
```

Point-point collision detection is rarely used in games. Why? Imagine two soldiers firing their guns at each other. How likely would two bullets collide?

Line-Line Collision

From section Line we know that lines are endless. So there are just two situations where lines don't collide: when they are parallel and not equivalent.




```
Vector2D rotate_vector_90(Vector2D v)
{
    Vector2D r;
    r.x = -v.y;
    r.y = v.x;
    return r;
}

Bool parallel_vectors(Vector2D a, Vector2D b)
{
    Vector2D na = rotate_vector_90(a);
    return equal_floats(0, dot_product(na, b));
}

Bool equal_vectors(Vector2D a, Vector2D, b)
{
    return equal_floats(a.x - b.x, 0) && equal_floats(a.y - b.y, 0);
}

Bool equivalent_lines(Line a, Line b)
{
    if(!parallel_vectors(a.direction, b.direction))
        return no;

    Vector2D d = subtract_vector(a.base, b.base);
    return parallel_vectors(d, a.direction);
}

Bool lines_collide(Line a, Line b)
{
    if(parallel_vectors(a.direction, b.direction))
```

```
        return equivalent_lines(a, b);
    else
        return yes;
}

Vector2D a = {3, 5};
Vector2D b = {3, 2};
Vector2D c = {8, 4};

Vector2D down = {5, -1};
Vector2D up = {5, 2};

Line l1 = {a, down};
Line l2 = {a, up};
Line l3 = {b, up};
Line l4 = {c, down};

assert(yes == lines_collide(l1, l2));
assert(yes == lines_collide(l1, l3));
assert(no == lines_collide(l2, l3));
assert(yes == lines_collide(l1, l4));
```

There are quite some functions involved in line collision detection. Let's go through it top-down.

As defined above only non-parallel or equivalent lines collide. Function *lines_collide()* first checks if the lines are parallel. If they are it returns their equivalence. Otherwise the lines collide so *yes* gets returned.

Function *parallel_vectors()* uses a simple trick to test vectors for parallelism. We know from section Dot Product that two vectors enclose a right angle when their dot product is zero. So when two vectors are parallel we just need to rotate one of them by 90° and check their dot product. That's exactly what *parallel_vectors()* does.

Instead of utilizing *rotate_vector()*, which takes arbitrary angles, we use the specialized function *rotate_vector_90()*. It's faster due to a simple trick which is explained in section Rotating by Right Angles.

A prerequisite for explaining *equivalent_lines()* is to know what equivalence means in this regard. There's a slight difference between *equal* and *equivalent*. Equal lines have the same base point and the same direction. One could be the clone of the other. Equivalent lines, on the other hand, just need parallel direction vectors and must have their base point somewhere on the other line. So line base and direction can be different from the other line's base and direction. Only the outcome, the drawn lines if you will, must be identical.

Function *equivalent_lines()* first tests for parallelism. If the lines are parallel we need to find out if the base point of one line lies on the other line. This is the case if the distance vector between the two base points is parallel to any of the lines.

left out pages

Get the full book from Amazon:

www.amazon.com

www.amazon.co.uk

www.amazon.de

Get the full book from the author:

www.collisiondetection2d.net

Use coupon OFF25 to get the book 25% off from the author:

www.collisiondetection2d.net/priceoff25

About the Author

Thomas Schwarzl is a game designer, developer and digital artist. At least some people say so.

Writing computer games since 2001 let him acquire quite some knowledge about their inner workings. Now the time has come to share the lessons learned with other developers. Thomas is working in several game development areas like programming, game design, 2D graphics and 3D modeling. But by far his favorite occupation is cranking out code while sipping steaming coffee.

He resides in the Alps, right in the heart of Austria. It's a nice place there, covered by green forests, surrounded by high mountains. The perfect place for writing games and books.

Aside from this book Thomas dumps his knowledge and thoughts about game development at www.blackgolem.com. Feel free to come by or drop him a line at thomas@blackgolem.com. Messages from readers are always welcome.