Table of Contents

Introduction

This tutorial takes will walk trough the basics of 2D Math in video games. The math needed for 2D is not too complicated, it relies mainly on geometry and trigonometry with a little bit or linear algebra. We're going to cover all three of these topics in great detail.

We're going to build two projects troughout this tutorial, the first one being a classic raycaster like doom. The second project is going to be a free form platformer, a clone of cartoon networks Samurai Jack, Code of the Samurai. We're also going to take a look at new ways to orgonize game states.

Resources for this tutorial are mainly in the form of Khan-Academy videos, tough the raycasting section was based on a chapter of this book, if you would like a PDF copy let me know. The book is written for an extinct version Java but it's still a damn good book.

License

References

- Math Is Fun Site

Points and Vectors

Points and vectors are represented similarly in code, but they are logically interpreted differently. Before we take a look at the similarities and differences, let's take a look at a definition for both.

Point

A point is a location in space. It is finite. Visually, a point is represented by a small circle. A 2D point is usually defined by two floating point numbers, like so:

```
class point2 {
    public float x;
    public float y;

    public point2() {
        x = 0.0f;
        y = 0.0f;
    }

    public point2(float x, float y) {
        this.x = x;
        this.y = y;
    }
}
```

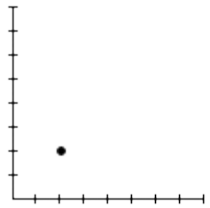
Vector

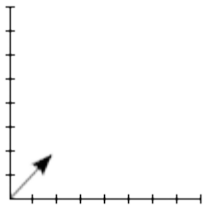
A vector represents an orientation and magnitude. Visually a vector is represented as an arrow. A vector is defined by two floating point numbers, like so:

```
class vec2 {
    public float x;
    public float y;

    public vec2() {
        x = 0.0f;
        y = 0.0f;
    }

    public vec2(float x, float y) {
        this.x = x;
        this.y = y;
    }
}
```

What's the difference?

Let's try to visualize the Point (2, 2) and the Vector (2, 2):



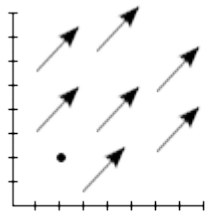


In the above example, Point is a Point. It's like a city. The city of Codeswille is at (2, 2).

A Vector is like a direction! The cidy of Codeswille is 2 units up and 2 units to the right from Debug Town.

Points are a finitie point in space, where as vectors are directions of where to go to get to a point. Or, how to get to a point.

This means that Vectors will lead to different places depending on where they start. Consider the following image:



All of the arrows are Vector (2, 2), but there is only one Point(2, 2). Any vector that goes two units up and two units to the right will be Vector (2, 2), regardless of where they start. Only the Point located at (2, 2) will be Point (2, 2)

When are they the same?

If a vector is measured from origin (0, 0) then the Vector (2, 2) points at the Point (2, 2). In this scenario the point and the vector can be used almost interchangably.

To clear up any confusion, so long as a world is created around the Point (0, 0) as it's origin you can represent any Point by a Vector.

Real World Example

In the real world, a Point can be used to represent a position, and a Vector can be used to represent a direction and some length. A good example might be a character controller:

```
class PlayerCharacter {
    point2 worldPosition;
    vec2 movementForce;
}
```

In general having different classes for Point and Vector tends to be a good idea. But the overhead of maintaining both can be a bit frustrating. At the same time Vector's provide some very useful functionality that points simply don't.

For this reason, most engines don't implement a Point class. Instead, they represent all position data as Vectors. So long as the world is modelled around the Point (0, 0) this approach will work. Like so:

```
class PlayerCharacter {
    vec2 worldPosition;
    vec2 movementForce;
}
```

Translation

Addition

When you add two vectors, the result is a new vector that represents a combination of both original vectors. This is the formula for adding Vector's P and Q:

$$\vec{P} + \vec{Q} = \vec{Q} + \vec{P} = [P_1 + Q_1, \quad P_2 + Q_2 \dots P_n + Q_n]$$

Addition is a simple component-wise operation. This means we add each component together individually, like so:

```
class vec2 {
    public float x;
    public float y;

    // Constructors

    public static vec2 operator+(vec2 v1, vec2 v2) {
        return new vec2(v1.x + v2.x, v1.y + v2.y);
    }
}
```

How can we visualize this? Lets start with a Vector (2, 2). This represents a transformation 2 units up and 2 units to the right.

[IMAGE]

Now, if we want to add the Vector (3, 3) to it, we start at origin and follow the first vector to it's end point (2, 2). From there we add Vector(3, 3) by going 3 units up and 3 units to the right

[IMAGE]

Once we have followed both vectors, we get the resulting Vector, Vector (5, 5).

[IMAGE]

The order of addition does not matter. **Vector (2, 2) + Vector(3, 3) == Vector(3, 3) + Vector (2, 2)**

[IMAGE]

Subtraction

Like addition, subtraction is also component-wise. This is the formula for subtracting Vector Q from Vector P:

$$\vec{P} - \vec{Q} = [P_1 - Q_1, P_2 - Q_2 \dots P_n - Q_n]$$

The code looks a lot like addition:

```
class vec2 {
    public float x;
    public float y;

    // Constructors
    // operator +

    public static vec2 operator-(vec2 v1, vec2 v2) {
        return new vec2(v1.x - v2.x, v1.y - v2.y);
    }
}
```

You can think of subtracting vectors like adding negative numbers, the same rules apply! For example, lets start with the Vector (3, 3)

[IMAGE]

If we want to subtract the Vector (1, 1) we start at origin, go 3 up and 3 to the right. Then we move one down and one to the left

[IMAGE]

This leaves us with our resulting Vector, Vector(2, 2):

[IMAGE]

The order of subtraction matters! **Vector(3, 3) - Vector(1, 1) != Vector(1, 1) - Vector(3, 3)**

[IMAGE]

Real World Example

Lets say you have a hierarchy of objects and you decide to store world position as a vector. Your calss might look something like this:

```
class GameObject {
    GameObject parent;
    vec2 localPosition;
}
```

You can find the World position of the game object by following all of the vectors of every parent of a given object. By adding all of the vectors together, you create a new transform, that points to where this game object is located.

```
class GameObject {
    GameObject parent;
    vec2 localPosition;

    vec2 WorldPosition {
        get {
            if (parent == null) {
                return localPosition;
            }
            return localPosition + parent.WorldPosition;
        }
    }
}
```

Cover scaling, scaling by a vector and inverse vector

Cover length and length squared