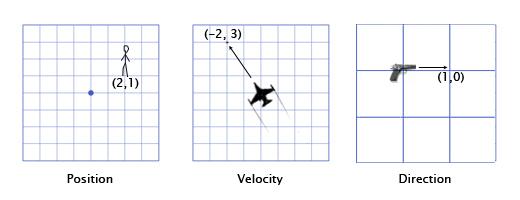
**Chapter 6**

**Math and Animation for game Developers**

# Linear algebra for game developers[[1]](#footnote-0)

### What is a vector?

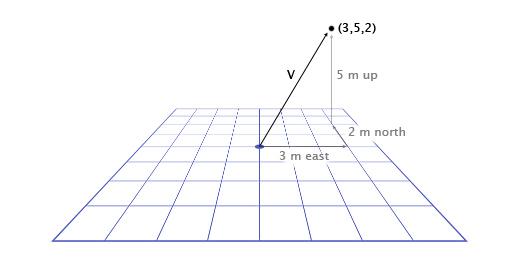
In games, vectors are used to store positions, directions, and velocities. Here are some 2-Dimensional examples:



The position vector indicates that the man is standing two meters east of the origin, and one meter north. The velocity vector shows that in one minute, the plane moves three kilometers up, and two to the left. The direction vector tells us that the pistol is pointing to the right.

As you can see, a vector by itself is just a set of numbers -- it is only given meaning by its context. For example, the vector (1,0) could be the direction for the gun as shown, but it could also be the position of a building one mile to the east of your current position, or the velocity of a snail moving right at a speed of 1 mph.

For this reason, it's important to keep track of your units. Let's say we have a vector V (3,5,2). This doesn't mean much by itself. Three what? Five what? In Overgrowth, positions are always given in meters, and velocities in meters per second. The first number is east, the second is up, and the third is north. Negative numbers represent the opposite directions: west, down, and south. The position represented by (3,5,2) is 3 meters east, 5 meters up, and 2 meters north, as shown here:

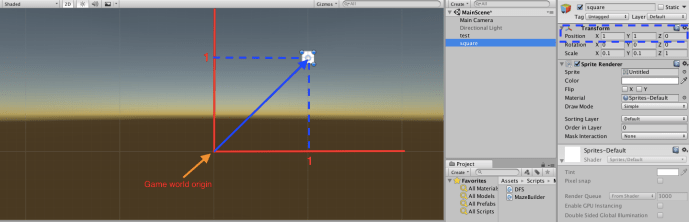


Now that we've gone over the basics of vectors, we need to know how to use them.

### Use vector to represent a point in space

Image we want to describe where an object is placed in our 3D or 2D game, how can we do that? Taken the origin of our 3D world, we can draw a vector from it to the game object and this is the vector that describes the position of the game object inside the game world.

In fact, imagine drawing an “arrow” from the origin of the 3D world to the position of the game object, that is the exact vector that we’re looking for.



If you look at the image you’ll notice that the vector in question is nothing more than the coordinate of X,Y,Z in a 3D cartesian system which in our case represent the game world space.

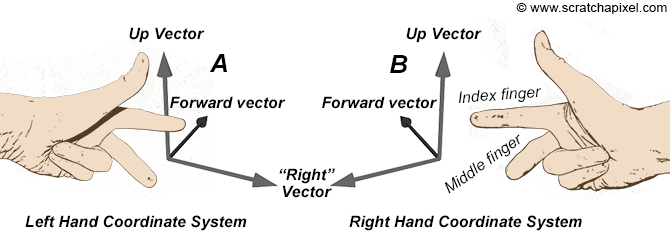
So we understand that vectors can not only be used to describe physical simulations, like the motion of a rigid body in a 3D world, but they can also be helpful to describe the position of an object inside the game world.

## Orientation of vectors[[2]](#footnote-1)

### Left-hand system vs right-hand system

In the right-hand system (RH), when curling your fingers around the z-axis with the thumb pointing toward the z positive direction, your fingers point from the x-axis toward the y-axis.

In the left system, it is the same method but using the left hand.



It’s important to know which system your engine is using and stick to that convention when using it.

### Vector addition

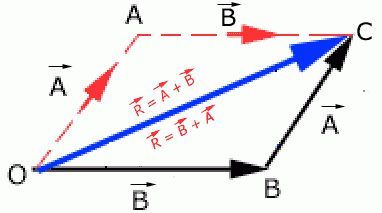
To add vectors together, you just add each component together separately. For example:

(0,1,4) + (3,-2,5) = (0+3, 1-2, 4+5) = (3,-1,9)

Adding and subtracting vectors are quite easy Given two vectors A and B, the addition of the 2 is the sum of the single components of the vector.

For example :

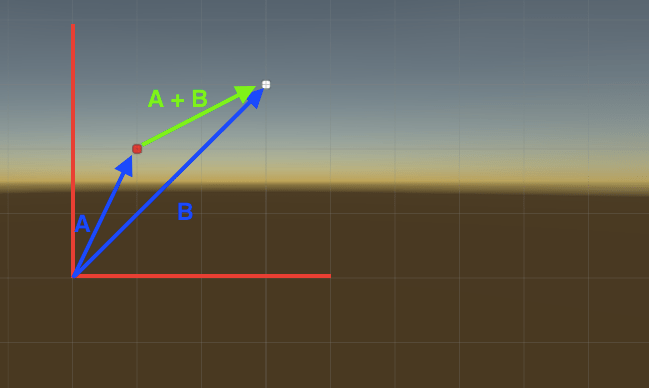
a + b = [(ax + bx), (ay + by), (az + bx)].



Same things goes with the subtraction.

a - b = [(ax – bx), (ay – by), (az – bx)].

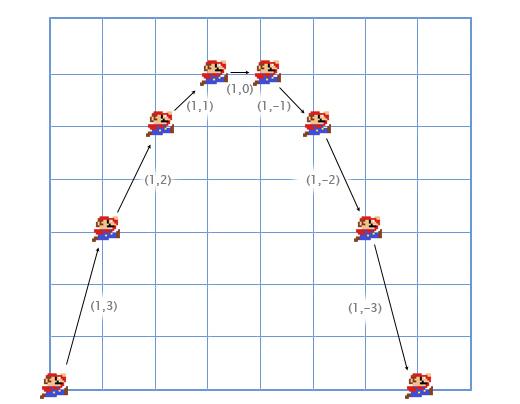
A good use of the addition and subtraction of vectors in games could be in finding the distance between 2 game objects.



In this case, the distance from A to B is defined as the sum of the 2 vectors (A + B).

Why do we want to add vectors together? One of the most common applications in games for vector addition is physics integration. Any physically-based object will likely have vectors for position, velocity, and acceleration. For every frame (usually 1/60th of a second), we have to integrate these vectors -- that is, add the velocity to the position, and the acceleration to the velocity.

Let's consider the example of Mario jumping. He starts at position (0,0). As he starts the jump, his velocity is (1,3) -- he is moving upwards quickly, but also to the right. His acceleration throughout is (0,-1), because gravity is pulling him downwards. Here is what his jump looks like over the course of seven more frames. The black text specifies his velocity for each frame.



We can walk through the first couple frames by hand to see how this works.

For the first frame, we add his velocity (1,3) to his position (0,0) to get his new position (1,3). Then, we add his acceleration (0,-1) to his velocity (1,3) to get his new velocity (1,2).

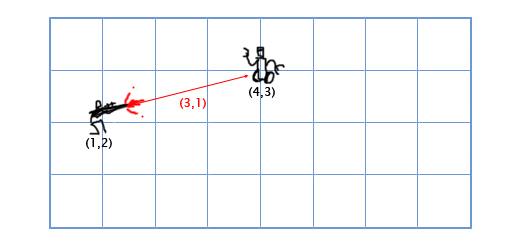
We do it again for the second frame. We add his velocity (1,2) to his position (1,3) to get (2,5). Then, we add his acceleration (0,-1) to his velocity (1,2) to get (1,1).

Usually in games the player controls a character's acceleration with the keyboard or gamepad, and the game calculates the new velocity and position using physics integration (via vector addition). Fun fact: this is the same kind of integration problem that you solve using integral calculus - we are just using an approximate brute-force approach. I found it much easier to pay attention to calculus classes by thinking about physical applications like this.

### Vector subtraction

Subtraction works in the same way as addition -- subtracting one component at a time. Vector subtraction is useful for getting a vector that points from one position to another. For example, let's say the player is standing at (1,2) with a laser rifle, and an enemy robot is at (4,3). To get the vector that the laser must travel to hit the robot, you can subtract the player's position from the robot's position. This gives us:

(4,3)-(1,2) = (4-1, 3-2) = (3,1).



### Scalar-vector multiplication

When we talk about vectors, we refer to individual numbers as scalars. For example, (3,4) is a vector, 5 is a scalar. In games, it is often useful to multiply a vector by a scalar. For example, we can simulate basic air resistance by multiplying the player's velocity by 0.9 every frame. To do this, we just multiply each component of the vector by the scalar. If the player's velocity is (10,20), the new velocity is:

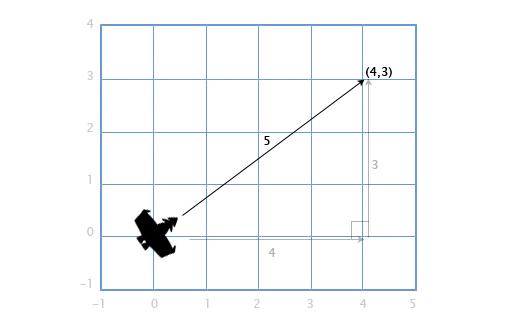
0.9\*(10,20) = (0.9\*10, 0.9\*20) = (9,18).

### Length

If we have a ship with velocity vector V (4,3), we might also want to know how fast it is going, in order to calculate how much the screen should shake or how much fuel it should use. To do that, we need to find the length (or magnitude) of vector V. The length of a vector is often written using || for short, so the length of V is |V|.

We can think of V as a right triangle with sides 4 and 3, and use the Pythagorean theorem to find the hypotenuse: x2 + y2 = h2. That is, the length of a vector H with components (x,y) is sqrt(x2+y2). So, to calculate the speed of our ship, we just use:

|V| = sqrt(42+32) = sqrt(25) = 5

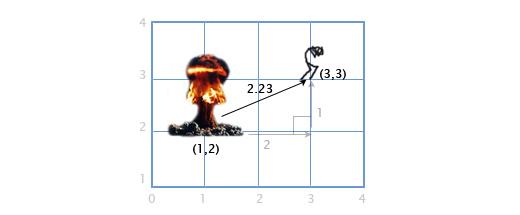


This works with 3D vectors as well -- the length of a vector with components (x,y,z) is sqrt(x2+y2+z2).

### Distance

If the player P is at (3,3) and there is an explosion E at (1,2), we need to find the distance between them to see how much damage the player takes. This is easy to find by combining two tools we have already gone over: subtraction and length. We subtract P-E to get the vector between them, and then find the length of this vector to get the distance between them. The order doesn't matter here, |E-P| will give us the same result.

Distance = |P-E| = |(3,3)-(1,2)| = |(2,1)| = sqrt(22+12) = sqrt(5) = 2.23



### Normalization

When we are dealing with directions (as opposed to positions or velocities), it is important that they have unit length (length of 1). This makes life a lot easier for us. For example, let's say there is a gun pointing in the direction of (1,0) that shoots a bullet at 20 m/s. What is the velocity of the bullet? Since the direction has length 1, we can just multiply the direction and the bullet speed to get the bullet velocity: (20,0). If the direction vector had any other length, we couldn't do this -- the bullet would be too fast or too slow.

A vector with a length of 1 is called "normalized". So how do we normalize a vector (set its length to 1)? Easy, we divide each component by the vector's length. If we want to normalize vector V with components (3,4), we just divide each component by its length, 5, to get (3/5, 4/5). Now we can use the pythagorean theorem to prove that it has length 1:

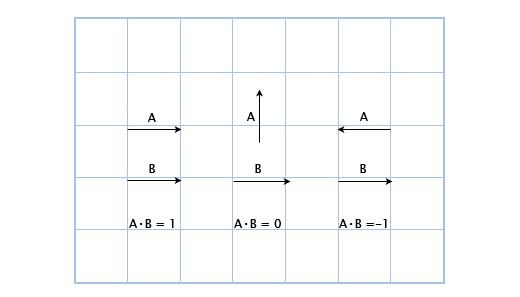
(3/5)2 + (4/5)2 = 9/25 + 16/25 = 25/25 = 1

### Dot product

What is the dot product (written •)? Let's hold off on that for a second, and look at how we calculate it. To get the dot product of two vectors, we multiply the components, and then add them together.

(a1,a2)•(b1,b2) = a1b1 + a2b2

For example, (3,2)•(1,4) = 3\*1 + 2\*4 = 11. This seems kind of useless at first, but lets look at a few examples:

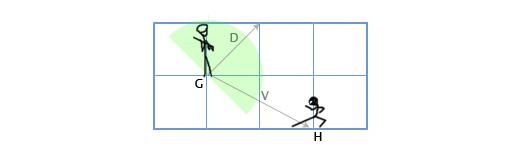


Here, we can see that when the vectors are pointing the same direction, the dot product is positive. When they are perpendicular, the dot product is zero, and when they point in opposite directions, it is negative. Basically, it is proportional to how much the vectors are pointing in the same direction. This is a small taste of the power of the dot product, and it's already pretty useful!

Let's say we have a guard at position G (1,3) facing in the direction D (1,1), with a 180 field of view. We have a hero sneaking by at position H (3,2). Is he in the guard's field of view? We can find out by checking the sign of the dotproduct of D and V (the vector from the guard to the hero). This gives us:

V = H-G = (3,2)-(1,3) = (3-1,2-3) = (2,-1)  
D•V = (1,1)•(2,-1) = 1\*2+1\*-1 = 2-1 = 1

Since 1 is positive, the hero is in the guard's field of view!



We know that the dot product is related to the extent to which the vectors are pointing in the same direction, but what is the exact relation? It turns out that the exact equation for the dot product is:

AB = |A||B|cosθ

Where θ (pronounced "theta") is the angle between A and B. This allows us to solve for θ if we want to find out the angle:

θ = acos([AB]/[|A||B|]).

As I mentioned before, normalizing vectors makes our life easier! If A and B are normalized, then the equation is simply:

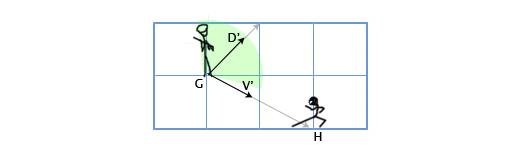
θ = acos(AB)

Let's revisit the guard scenario above, except the guard's field of view is only 120. First, we get the normalized vectors for the direction the guard is facing (D'), and the direction from the guard to the hero (V'). Then, we check the angle between them. If it is greater than 60 (half of the field of view), then the hero is not seen.

D' = D/|D| = (1,1)/sqrt(12+12) = (1,1)/sqrt(2) = (0.71,0.71)  
V' = V/|V| = (2,-1)/sqrt(22+(-1)2) = (2,-1)/sqrt(5) = (0.89,-0.45)

θ = acos(D'V') = acos(0.71\*0.89 + 0.71\*(-0.45)) = acos(0.31) = 72

The angle between the center of the guard's vision and the hero is 72, so the guard does not see him!



I know this looks like a lot of work, and it is, because I'm doing it by hand. However, in a program, this is pretty simple. Here is what this would like in Overgrowth using the C++ vector libraries I wrote (inspired by GLSL syntax).

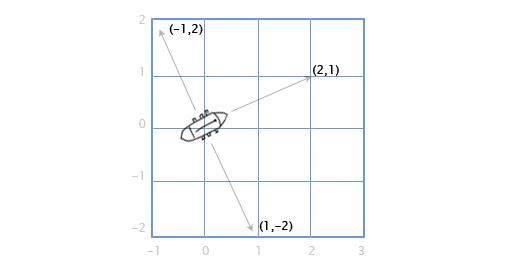
//Initialize starting vectors  
vec2 guard\_pos = vec2(1,3);  
vec2 guard\_facing = vec2(1,1);  
vec2 hero\_pos = vec2(3,2);

//Prepare normalized vectors  
vec2 guard\_facing\_n = normalize(guard\_facing);  
vec2 guard\_to\_hero = normalize(hero\_pos - guard\_pos);

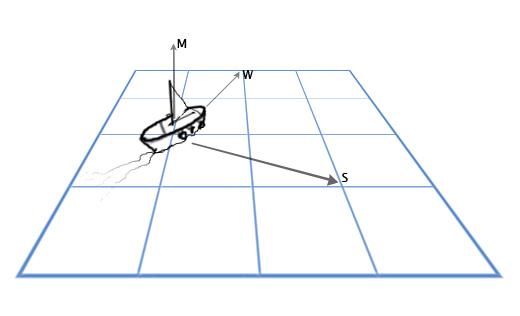
//Check angle  
float angle = acos(dot(guard\_facing\_n, guard\_to\_hero));

### Cross Product

Let's say you have a boat that has cannons that fire to the left and right. Given that the boat is facing along the direction vector (2,1), in which directions do the cannons fire? This is easy in 2D: to rotate 90 degrees clockwise, just flip the two vector components, and then switch the sign of the second component. (a,b) becomes (b,-a). So, if the boat is facing along (2,1), the right-facing cannons fire towards (1,-2). The left-facing cannons fire in the opposite direction, so we flip both signs to get: (-1,2).



So, what if we want to do this in 3D? Let's revisit our sailing ship. We have a vector for the direction of the mast M, going straight up (0,1,0), and the direction of the north-north-east wind W (1,0,2), and we want to find the direction the sail S should stick out in order to best catch the wind. The sail has to be perpendicular to the mast, and also perpendicular to the wind. To solve this, we can use the cross product: S = M x W.



The cross product of A(a1,a2,a3)) and B(b1,b2,b3)) is:

(a2b3-a3b2, a3b1-a1b3, a1b2-a2b1)

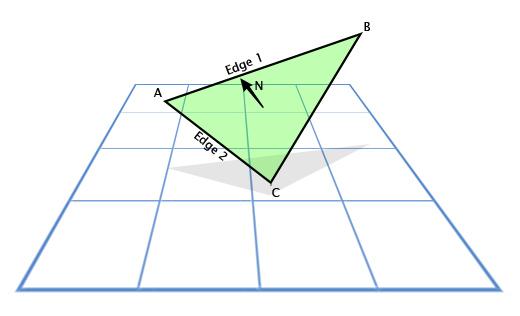
So now we can plug in our numbers and solve our problem:

S = MxW = (0,1,0)x(1,0,2) = ([1\*2-0\*0], [0\*1-0\*2], [0\*0-1\*1]) = (2,0,-1)

This is pretty ugly to do by hand. For most graphics and game work I would recommend just encapsulating it in a function like the one below, and never thinking about the details again.

vec3 cross(vec3 a, vec3 b) {  
    vec3 result;  
    result[0] = a[1] \* b[2] - a[2] \* b[1];  
    result[1] = a[2] \* b[0] - a[0] \* b[2];  
    result[2] = a[0] \* b[1] - a[1] \* b[0];  
    return result;  
}

Another common use for the cross product in games is to find surface normals -- the direction that a surface is facing. For example, let's take a triangle with vertex vectors A, B and C. How do we find the direction that the triangle is facing? It seems tricky, but we have the tools to do it now. We can use subtraction to get the direction from A to C (C-A) 'Edge 1' and A to B (B-A) 'Edge 2', and then use the cross product to find a new vector N perpendicular to both of them... the surface normal.



### LINEAR INTERPOLATION (LERP)

The linear interpolation (LERP) is one of the most common operations used in game development.

For example, if we want to smoothly animate from point A to point B over the course of two seconds at 30 frames per seconds, we would need to find 60 intermediate positions between A and B.

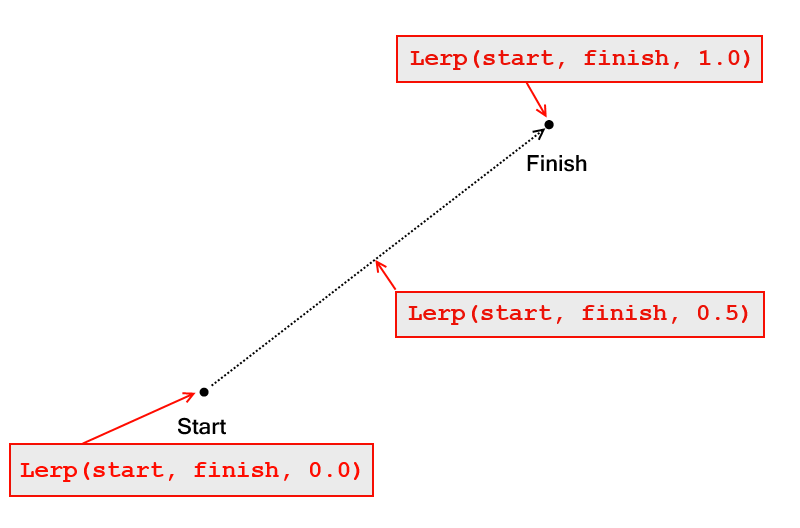
A linear interpolation is a mathematical operation to find an intermediate point between two known points. The operation can be defined as follows:

LERP Operation

Where β is a value between 0 and 1 where 0 represents the initial position of the Lerp and 1 as the final position.

As you can see from the definition, we’re representing the lerp as a **Vector**.

Here is an example to visualize the lerp in action:



Most game engines has a lerp function ready to be used. Here is an example of a code using the lerp function in Unity:

using UnityEngine;

using System.Collections;

public class ExampleClass : MonoBehaviour {

    public Vector3 startPosition;

    public Vector3 endPosition;

    public float speed = 1.0F;

    private float startTime;

    private float journeyLength;

    void Start() {

        startTime = Time.time;

        journeyLength = Vector3.Distance(startPosition, endPosition);

    }

    void Update() {

        float distCovered = (Time.time - startTime) \* speed;

        float fracJourney = distCovered / journeyLength;

        transform.position = Vector3.Lerp(startPosition,endPosition, fracJourney);

    }

}

## **Use vectors to describe objects and interactions in the game world.**

### Model space

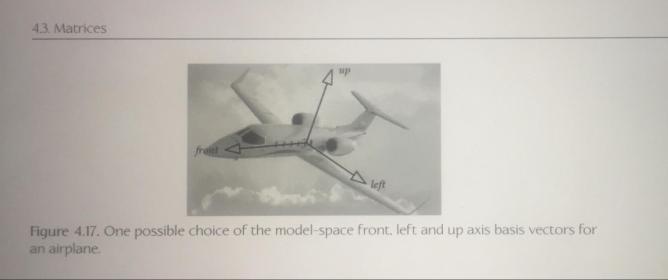
The model space is the reference system used by the object in question.

For example, a character game object’s forward axis points in the direction that the object is naturally facing.

Up: is the direction that point towards the top of the object

The left or right are the axes that points to its left or right sides respectively.

Example with a plane:



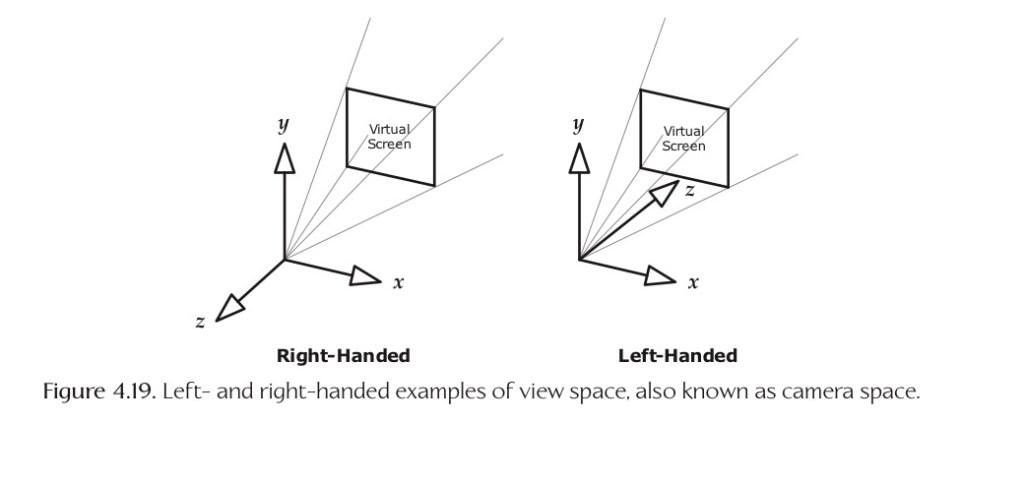
### World coordinate

A word coordinate is a fixed coordinate in which position, orientation, and scale of all the elements placed in the game are described.

The origin of the coordinate system is arbitrary, but it is a common practice to place it in the center of the world in order to reduce the number of floating point numbers needed to describe objects very far away from the origin.

The orientation of the axes is also arbitrary but usually, the convention is to have the y-axis as the axis pointing upward.

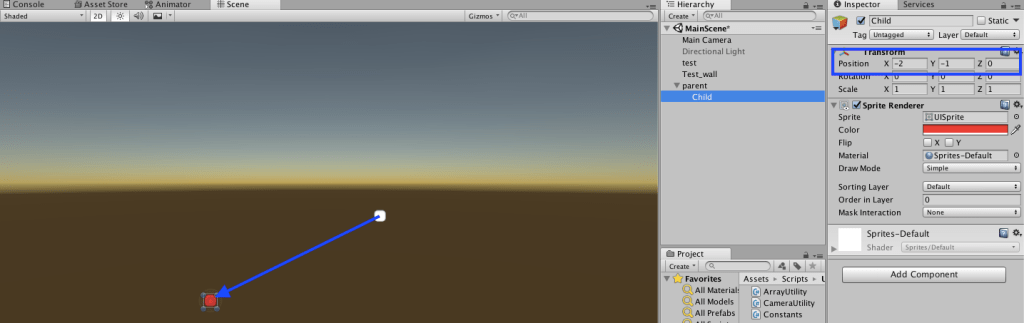
### View space

The view space coordinates are often called as the camera coordinates. These are the coordinates fixed or attached to the game camera.

As you can see from the image, you can have a right or left hand coordinate system attached to the camera.

### Coordinate space hierarchy

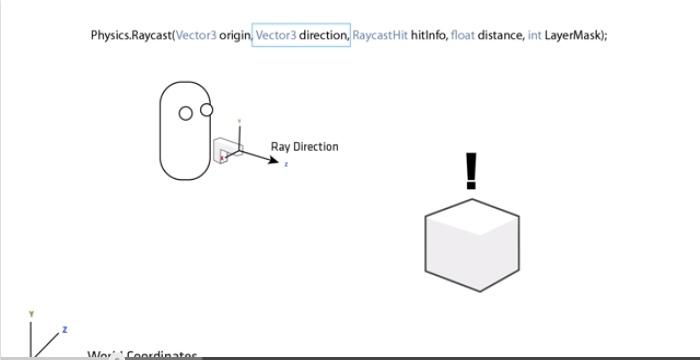
A coordinate space hierarchy is a coordinate system that expresses its position based on its parent object. For example, assuming you have a character with an arm attached. The vector that expresses the position of that arm is based on the coordinate system of its parent.

As you can see from the example, the coordinates of the red box (child) are expressed as a vector **from** the white box (parent) position.

# Raycasting

Ray casting is a common technique used in games to find when a line (ray) has hit a specific target. A common example is when a bullet is shot from a gun.  
As you can imagine, we’ll have to use Vectors to be able to correctly use the Ray Casting function available in most game engines.

### Let’s take this example from the Unity Tutorials:



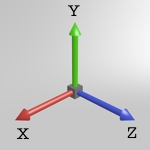
Here we want to shoot a projectile and find if we have hit the target (box). As you can see, the Raycast function takes two vectors as a parameter. One is the vector “origin” which describes the origin position of the shot in the game world. The other is the ray direction, which describes where the projectile is headed.

The other parameters are not important for this lecture, but the first 2 vectors are essential to discuss as they describe the position and the direction of the shot

# Spherical coordinates[[3]](#footnote-2):

What is spherical coordinates and where are they useful? These are some of the questions I’ll answer in this blog that also contains a implementation of spherical coordinates for the Unity Game game engine.

**The cartesian coordinate system**

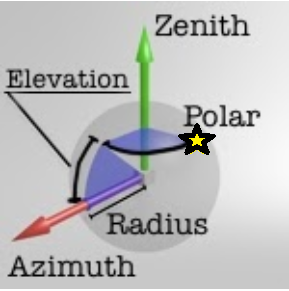
****

The cartesian coordinate system in 3D

First take a step back and look at how positions usually are specified in 3D space. Positions in space are usually written in the cartesian coordinate system, where the 3 axis X, Y, and Z span the space. A point in the cartesian coordinate system can be defined using a vector (x,y,z) from the origin to the point, where x is is the distance traveled along the x axis, y is the distance traveled along the y axis and z along the z axis. The cartesian coordinate works well as the reference coordinate system. Along with the cartesian coordinate system comes a lot of useful math using vectors and matrices.

Note that I have chosen to let the y axis point upwards as it does in Unity.

**The spherical coordinate system**

****

Spherical coordinate system (the star represents point with the measured angle)

Spherical coordinate system is an alternative coordinate system, where two orthogonale coordinate axis define the world space in 3D. The zenith axis points upwards and the azimuth axis points to the side. To define a point in this system the following is needed:

* *Radius*: the distance from the origin to the point
* *Elevation angle*: the angle between the plane (with zenith axis as normal) and the line from the origin to the point
* *Polar angle*: the rotation around the zenith axis

Elevation angle and polar angles are basically the same as latitude and longitude.

Note that a point specified in spherical coordinates may not be unique.

The spherical coordinate system I’ll be looking at, is the one where the zenith axis equals the Y axis and the azimuth axis equals the X axis.

All angles are in **radians**.

**Converting between spherical and cartesian coordinates**

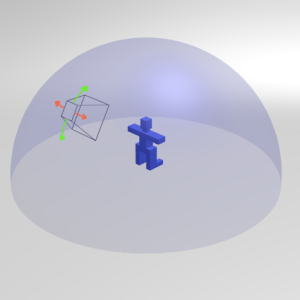
To convert from spherical coordinates to cartesian coordinates can be done the following way in c#-unity:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public static void SphericalToCartesian(float radius, float polar, float elevation, out Vector3 outCart){  float a = radius \* Mathf.Cos(elevation);  outCart.x = a \* Mathf.Cos(polar);  outCart.y = radius \* Mathf.Sin(elevation);  outCart.z = a \* Mathf.Sin(polar);  } |

And converting from cartesian coordinates to spherical coordinates (c#-unity example) :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static void CartesianToSpherical(Vector3 cartCoords, out float outRadius, out float outPolar, out float outElevation){  if (cartCoords.x == 0)  cartCoords.x = Mathf.Epsilon;  outRadius = Mathf.Sqrt((cartCoords.x \* cartCoords.x)  + (cartCoords.y \* cartCoords.y)  + (cartCoords.z \* cartCoords.z));  outPolar = Mathf.Atan(cartCoords.z / cartCoords.x);  if (cartCoords.x < 0)  outPolar += Mathf.PI;  outElevation = Mathf.Asin(cartCoords.y / outRadius);  } |

**Usages of spherical coordinates**

****

Chase camera

One usage of spherical coordinates is in third person chase camera systems, where the player controls both the player-character and the movement of the camera. On console games this is usually done by mapping one joystick to control player movement and another to control camera movement. The camera movement is restricted to move around the player giving – a perfect example of how to use spherical coordinates. The radius is simple the distance between the player character and the camera, the elevation angle can be mapped to the vertical joystick state and the polar angle can be mapped to the horizontal joystick state.

A complete implementation of a chase camera using spherical coordinates can be found as a part of a sample camera library in unity:

<http://github.com/mortennobel/CameraLib4U/>

# Polar, Spherical and Geographic Coordinates[[4]](#footnote-3)

Polar and spherical coordinate systems do the same job as the good old cartesian coordinate system you always hated at school. It describes every point on a plane or in space in relation to an origin **O** by a vector. But instead of 3 perpendicular directions xyz it uses the distance from the origin and angles to identify a position.

#### Conventions

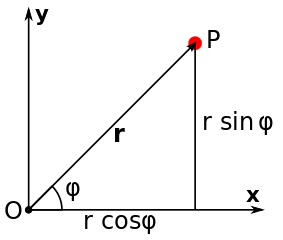
In the following descriptions the angle units are degree and the cartesian coordinate systems and drawings are the ones you would find in math textbooks.

### **2D**

In 2d the definition is straightforward. A position is defined by the distance to the origin and one angle. We just need the:

* origin **O**
* a reference direction where the angle is 0

For practical reasons mathematicians place the origin at the same position as it is in the cartesian system and the reference direction is the positive x-axis:



Then the conversion from a cartesian vector (x, y) of a position **P** to polar coordinates (radius, angle) is:

radius = sqrt(x^2 + y^2)

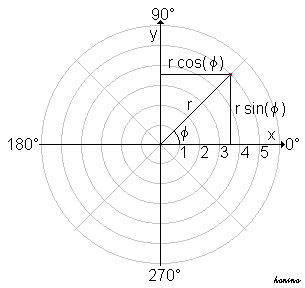
angle = atan2(y, x)

and the way around:

x = radius \* cos(angle)

y = radius \* sin(angle)

Here a positive angular velocity moves the position counter-clockwise on a circle:



Note that many 2d computer graphics coordinate systems have the y-axis pointing downwards so that everything is flipped upside down. In that case, using the same calculations as above, a positive angular velocity moves the position clockwise.

To get the same behavior in a 2d cartesian system with y-axis down the calculations would be:

radius = sqrt(x^2 + y^2)

angle = atan2(-y, x)

and:

x = radius \* cos(angle)

y = -radius \* sin(angle)

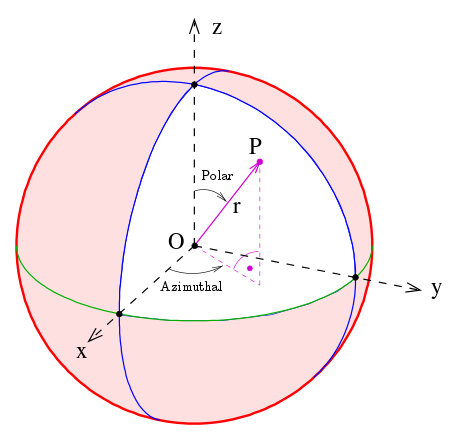
### **3D**

To define a point in space by spherical coordinates the distance to the origin **O** as well as two angles are required. The confusion starts here since many conventions for the notation and the order of the angles exist. This page lists most of them:<http://mathworld.wolfram.com/SphericalCoordinates.html>

But let's step back and have a look at what we need to define spherical coordinates. We will see that regardless of the notation the actual formula for the calculation is the same:

* the origin **O**
* for one angle we need a directed axis which defines the poles (like north and south pole of the earth), this angle is often called *polar angle, zenith angle, colatitude, inclination or elevation زاوية الارتفاع أو الانخفاض*
* for the other angle we need a reference direction in the equatorial plane, this angle is called *azimuthal angle الزاوية العرضية*

The origin is also the same as the one of the cartesian system. Traditionally mathematicians choose the z-axis as the polar axis and the xy-plane as the equatorial plane with reference direction as the positive x-axis:



The conversion formulas are then:

radius = sqrt(x^2 + y^2 + z^2)

polar = arccos(z/radius)

azimuthal = atan2(y, x)

and:

x = radius \* sin(polar) \* cos(azimuthal)

y = radius \* sin(polar) \* sin(azimuthal)

z = radius \* cos(polar)

As you can see in the drawing, if polar angle is 0 the vector points toward the positive z-axis and the azimuthal angle has no effect because it only rolls the vector around the z-axis.

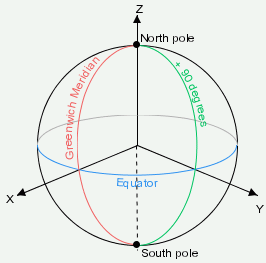
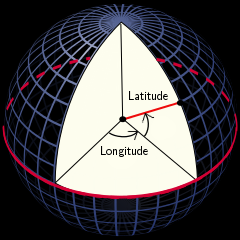
Positive polar velocity moves the point away from the pole at positive z towards positive x.

Positive azimuthal velocity moves the point from positive x towards positive y.

The drawing uses a right-handed system with z-axis up which is common in math textbooks. As in the 2d case it looks different depending on orientation of the xyz-axis of the cartesian coordinate system in which the position will be displayed.

#### Geographic Coordinates

The definition of the spherical coordinates has two drawbacks. First the polar angle has to have a value other than 0° (or 180°) to allow the azimuthal value to have an effect. Second the geographic system of latitude and longitude does not match with the two angles.

In order to match the spherical angles to latitude and longitude the polar angle needs to have a value of 90°. Then the position vector points towards the positive x-axis in the equatorial plane which matches a latitude of 0° and a longitude of 0°.  
 

The angular directions of latitude and longitude are the same. So the conversion is quite simple:

latitude = polar - 90°

longitude = azimuthal

and:

polar = latitude + 90°

azimuthal = longitude

With trigonometric substitutions a direct conversion between geographic and cartesian coordinates can be derived:

radius = sqrt(x^2 + y^2 + z^2)

latitude = arcsin(z/radius)

longitude = atan2(y, x)

and:

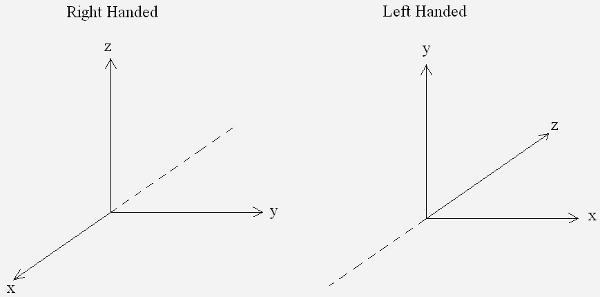
x = radius \* cos(latitude) \* cos(longitude)

y = radius \* cos(latitude) \* sin(longitude)

z = radius \* sin(latitude)

## Conversion between rightHanded and Left Handed systems[[5]](#footnote-4)

Many graphics libraries use a **left-handed** cartesian coordinate system with the **y-axis up** which is commonly used with DirectX. This means that all the above images and directions would be somehow **rotated** and **flipped** when used in such a coordinate system. But that's of course not what we want. The north pole of a sphere should still be up and the angular directions of the angles should also be the same as above.



The conversion of a vector between the systems is not very complicated:

xR = -zL

yR = xL

zR = yL

and:

xL = yR

yL = zR

zL = -xR

# Parametric curves[[6]](#footnote-5)

## Parametric curves

You are probably familiar with curves like **y = x²** or **f(x)=x²**. A parametric curve is similar to those, but the x- and y-values are calculated in separate functions. To do this we need another variable (I will call it **a**). So instead of **f(x)**, we say **X(a)** and **Y(a)**, where **X(a)** gives you an x-value for each value of **a**, and **Y(a)** gives you an corresponding y-value for the same value of **a**. Here is an example:

**X(a) = a/2**

**Y(a) = a+5**

If we substitute **a** with a number we can get a point that lies on the curve/line:

**a = 6**

**X(6) = 6/2 = 3 = x**

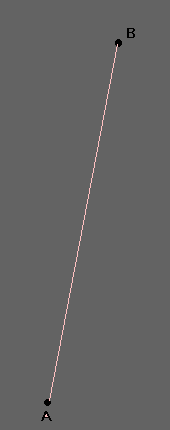
**Y(6) = 6+5 = 11 = y**

We now know that **(3,11)** is a point on the curve.

We can easily make a curve in 3D just by defining **Z(a)**.

# Bezier curves

## Simple line



The simplest form is a straight line from a control point **A**, to a control point **B**.

I will use **Ax** to denote the x-coordinate of control point **A**, and **Ay** to denote the y-coordinate. The same goes for **B**.

I am introducing another variable, **b**, just to make it a bit more simple to read the functions. However, the variable **b** is only **a** in disguise, **b** is always equal to **1-a**. So if **a = 0.2**, then **b = 1-0.2 = 0.8**. So when you see the variable **b**, think: **(1-a)**.

This parametric curve describes a line that goes from point **A**, to point **B** when the variable **a** goes from **1.0** to **0.0**:

**X(a) = Ax·a + Bx·b**

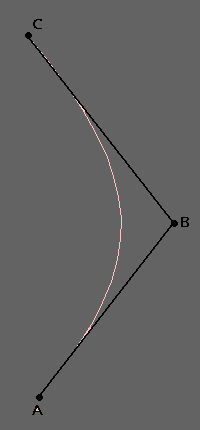
**Y(a) = Ay·a + By·b**

**Z(a) = Az·a + Bz·b**

If you set **a = 0.5** (and thus also **b = 0.5**, since **b = 1.0-a = 1.0-0.5 = 0.5**) then **X(a)**, **Y(a)** and **Z(a)** will give you the 3D coordinate of the point on the middle of the line from point **A** to point **B**.

Note that the reason this works is because **a + b** is always equal to one, and when you multiply something with one, it will stay unchanged. This makes the curve behave predictably and lets you place the control points anywhere in the coordinate system, since when **a = 1.0** then **b = 0.0** thus making the point equal to one of the control points, and completely ignoring the other one.

## Quadratic



Ok, now we have done lines, but what about curves?

This: **(a+b)=1**, where **b = (1.0-a)** is the key to everything. We know that a polynomial function is a curve, so why not try to make **(a+b)** a polynomial function? Let's try:

**(a+b)² = a² + 2a·b + b²**

Knowing that **b=1-a** we see that **a² + 2·a·b + b²** is still equal to one, since **(a+b) = 1**, and **1² = 1**.

We now need three control points, **A**, **B** and **C**, where **A** and **C** are the end points of the curve, and **B** decides how much and in which direction it curves. Except for that, it is the same as with the parametric line.

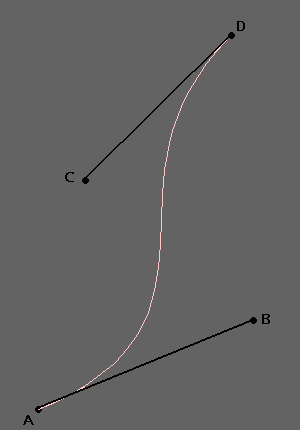
**X(a) = Ax·a² + Bx·2·a·b + Cx·b²**

**Y(a) = Ay·a² + By·2·a·b + Cy·b²**

**Z(a) = Az·a² + Bz·2·a·b + Cz·b²**

Still, if you set **a = 0.5**, then **a² = 0.25**, **2·a·b = 0.5** and **b² = 0.25** giving the middle point, **B**, the biggest impact and making point **A** and **C** pull equally to their sides. If you set **a = 1.0**, then **a² = 1.0**, **a·b = 0.0** and **b² = 0.0** meaning that result is the control point **A** itself, just the same as setting **a = 0.0** will return the coordinates of control point **C**.

## Cubic



We can also increase the number of control points using **(a+b)³** instead of **(a+b)²**, giving you more control over how to bend the curve.

**(a+b)³ = a³ + 3·a²·b + 3·a·b² + b³**

Now we need four control points **A**, **B**, **C** and **D**, where **A** and **D** are the end points.

**X(a) = Ax·a³ + Bx·3·a²·b + Cx·3·a·b² + Dx·b³**

**Y(a) = Ay·a³ + By·3·a²·b + Cy·3·a·b² + Dy·b³**

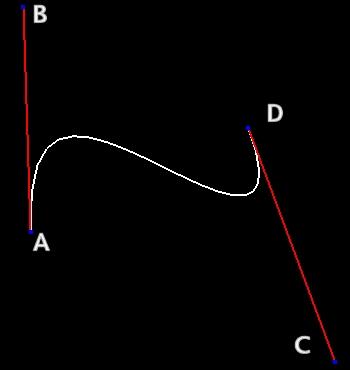
**Z(a) = Az·a³ + Bz·3·a²·b + Cz·3·a·b² + Dz·b³**

It still works the same way as the previous ones, as **a** goes from **1.0** to **0.0** (and thus **b** from **0.0** to **1.0**) the functions will return coordinates on a smooth line from control point **A** to control point **D**, curving towards **B** and **C** on the way.

You can easily use **(a+b)** to the power of **n** and get **n+1** control points (where **n** is any integer equal to, or greater than one). But remember that the more control points you have, the more work it is, both for you and for the computer. This might not seem like much when only talking about lines, but when it comes to surfaces it definitely has something to say.

Personally I like the cubic ones best. You can easily make circles with them, and you can control the direction of the curve independently at each control point.

**Example of how to draw a cubic curve using OpenGL and C++:**

  
**this is how it will look  
(excluding the red lines and the letters)**

// Control points (substitute these values with your own if you like)

double Ax = -2.0; double Ay = -1.0; double Az = 1.0;

double Bx = -1.0; double By = 3.0; double Bz = 1.0;

double Cx = 1.0; double Cy = -3.0; double Cz = -1.0;

double Dx = 2.0; double Dy = 1.0; double Dz = -1.0;

// Points on the curve

double X;

double Y;

double Z;

// Variable

double a = 1.0;

double b = 1.0 - a;

// Tell OGL to start drawing a line strip

glBegin(GL\_LINE\_STRIP);

/\* We will not actually draw a curve, but we will divide the curve into small

points and draw a line between each point. If the points are close enough, it

will appear as a curved line. 20 points are plenty, and since the variable goes

from 1.0 to 0.0 we must change it by 1/20 = 0.05 each time \*/

for(int i = 0; i <= 20; i++)

{

// Get a point on the curve

X = Ax\*a\*a\*a + Bx\*3\*a\*a\*b + Cx\*3\*a\*b\*b + Dx\*b\*b\*b;

Y = Ay\*a\*a\*a + By\*3\*a\*a\*b + Cy\*3\*a\*b\*b + Dy\*b\*b\*b;

Z = Az\*a\*a\*a + Bz\*3\*a\*a\*b + Cz\*3\*a\*b\*b + Dz\*b\*b\*b;

// Draw the line from point to point (assuming OGL is set up properly)

glVertex3d(X, Y, Z);

// Change the variable

a -= 0.05;

b = 1.0 - a;

}

// Tell OGL to stop drawing the line strip

glEnd();

/\* Normally you will want to save the coordinates to an array for later use. And

you will probably not need to calculate the curve each frame. This code

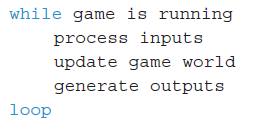
demonstrates an easily understandable way to do it, not necessarily the most

useful way. \*/

# Basics of Game programming[[7]](#footnote-6)

## The Game Loop

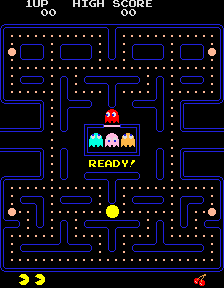
* The **game loop** is the overall flow control for the entire game program.
* It’s a **loop** because the game keeps doing a series of actions over and over again until the user quits.
* **Each iteration** of the game loop is known as a **frame** .
* Most real-time games update several times per second: 30 and 60 are the two most common **intervals**.
* If a game runs at 60 FPS ( **frames per second** ), this means that the game loop completes 60 iterations every second.
* A traditional game loop is broken up into **three** distinct phases: processing inputs, updating the game world, and generating outputs. At a high level, a basic game loop might look like this:



Where:

1. **processing inputs** : detecting any inputs from devices such as a keyboard, mouse, or controller.
2. **Updating the game world** : going through everything that is active in the game and **updating** its state and making **decisions**.
3. **generating outputs**, the most computationally expensive output is :
   * graphics, which may be 2D or 3D.
   * audio including sound effects, music,
   * and dialogue

for example, consider the Pacman classical game:



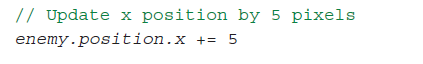
Example: Theoretical *Pac-Man* Game Loop

## Time and Games

* The majority of video games have some concept of time counting.
* For real-time games, that progression of time is typically measured in fractions of a second.
* As one example, a 30 FPS game has roughly 33ms *(1000ms divided by 30 frames)* elapse from frame to frame.

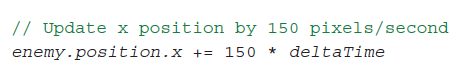
Logic as a Function of Delta Time

* Early games were often programmed with a specific processor speed in mind.
* as long as it worked properly on that processor it was considered acceptable.
* Example: the next code updates **enemy position by 5 pixels to the right every Frame.** ( **150** **pixels** in 1 second for **30** **FPS** devices)
* What if the game is run on another device with **60 FPS**? (300 pixels per second which is more than desired speed).

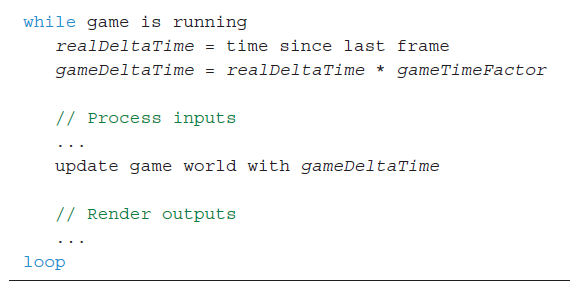


Solution: concept of **delta time**

* concept of **delta time** : *the amount of elapsed game time since the last frame.*
* think of the movement not in terms of *pixels per frame*, but in terms of pixels per second.
* So if the ideal movement speed is 150 pixels per second, this pseudocode would be preferable:



Game Loop with Delta Time



* The factor  **gameTimeFactor** سرعة can be speed factor, with values like :
  + 0 for no speed
  + 0.25 / 0.5 slower speed
  + 1 for normal speed,
  + 2 for double speed More than 1 for more speed

# 2D GRAPHICS

## Why 2D games are famous?

* Suitable to **Mobile** and **web** Platforms
* **Developers** are drawn to 2D because the typical **budget** and team can be much **smaller**.
* **Gamers** are drawn toward 2D because of the **simplicity** of the games.

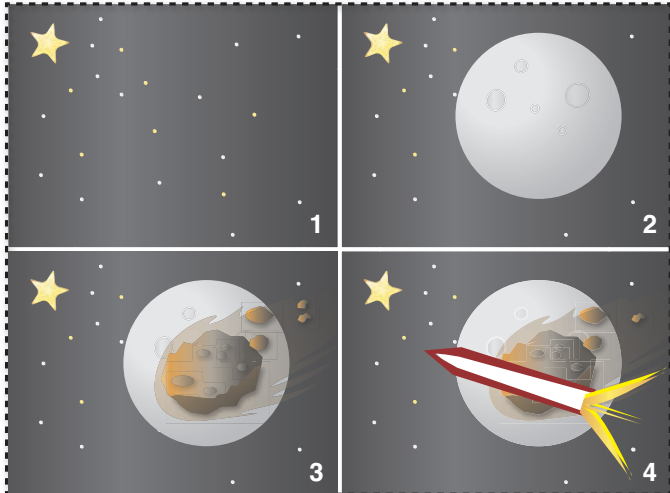
Many techniques are common between 2D and 3D games

Sprites صور ثنائية الأبعاد

* A **sprite** is a **2D visual object** within the game world that can be drawn using a single image on any given frame.
* Sprites can be used to represent:
  + characters
  + other dynamic objects.
  + backgrounds,
* Most 2D games have a lot of sprites,
* Mobile games the sprites often takes most of game size.
* it’s important to try to use sprites efficiently

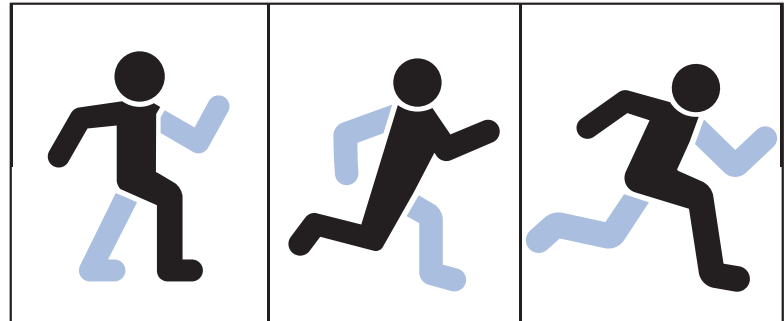
## Drawing Sprites

* Suppose you have a basic 2D scene with a **background** image and a **character** in the center.
* Painter’s Algorithm : The simplest approach to drawing this scene would be to:
  + first draw the background image
  + then draw the character.
* In the painter’s algorithm, all the sprites in a scene are **sorted from back to front (based on Z-Order/Depth value)**



## Animating Sprites

* Common technique: A series of static 2D images are **played** **fast** to create an **feeling** of motion
* minimum of 24 FPS is required
* This means that for every one second of animation, you need 24 individual images
* Some games use 60 FPS animations

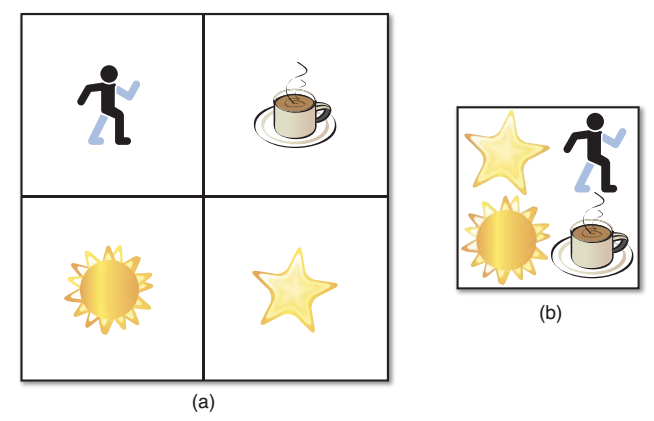


## Sprite Sheets

* it is preferable to have all the sprites for a particular character be the same size.
* many libraries also required that all image files had dimensions that were **powers of two**, (32x32, 64x64, ….) for performance optimization (faster drawing).
* A developer can have an individual image file (or **texture** ). But this will waste a great deal of memory.
* That’s because sprites usually aren’t rectangles, so, rectangle contains **unused areas**.

A solution to this problem is to use a single image file that contains all the sprites, called a **sprite sheet** .

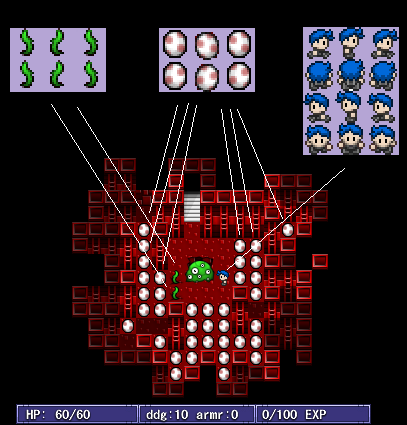
* In a sprite sheet, sprites are grouped **closely** and **overlap the unused space تقاطع**.
* This means that when the sprite sheet is opened, a bit of work is necessary to **reconstruct the correct images in memory**.
* This **reduces** total **installation size** of the game.



*Individual sprites (a), and those sprites packed in a sprite sheet (b)*

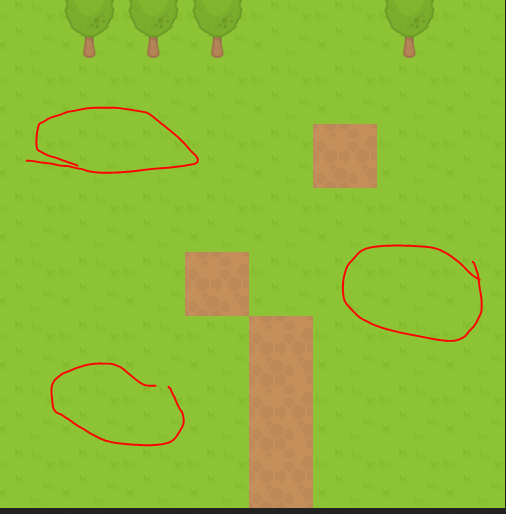
Sprite Loading

* You should only load a sprite sheet image once
  + Each behavior using the sprite maintains a reference to the sprite sheet
* Consider making a Resource class which loads in sprite sheets
  + Load in image
  + Handling image index for different sprites
  + Generalizable to other assets like maps, sounds, text, etc…

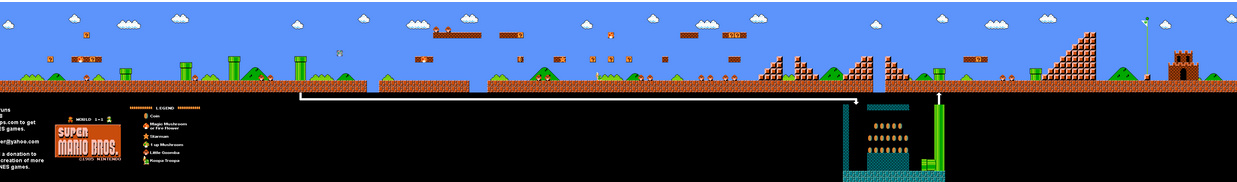


Tile Maps

* A tilemap is a technique for creating a game world out of **modular building blocks**.
* When you break a world down into small blocks/pieces, you get **memory**, **performance** and **creative** advantages.
* **Alternative** to creating large background images for games.
* Useful for **repeating patterns**

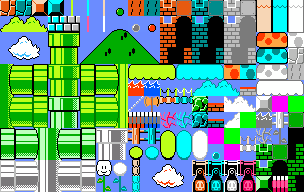


* Imagine trying to recreate Mario from scratch. Let’s say we decide to try loading each level as a giant image file. World 1–1 would be over 3500px wide
* We’d need a lot of pixels to store that 32 levels for example.
* it would be hard to sync up the image with logic with the game. Which pixels can Mario stand on? Which pixels correspond to pipes he can enter?



Example-continue

* The tilemap approach defines a set of modular, regularly-sized *tiles مربعات متساوية الأبعاد* that we can use to build our levels. That way, we only need one image, a *tileset*:



* We can represent tilemap as 2D matrix in code

# Some Techniques for Game Development

## Collision Detection

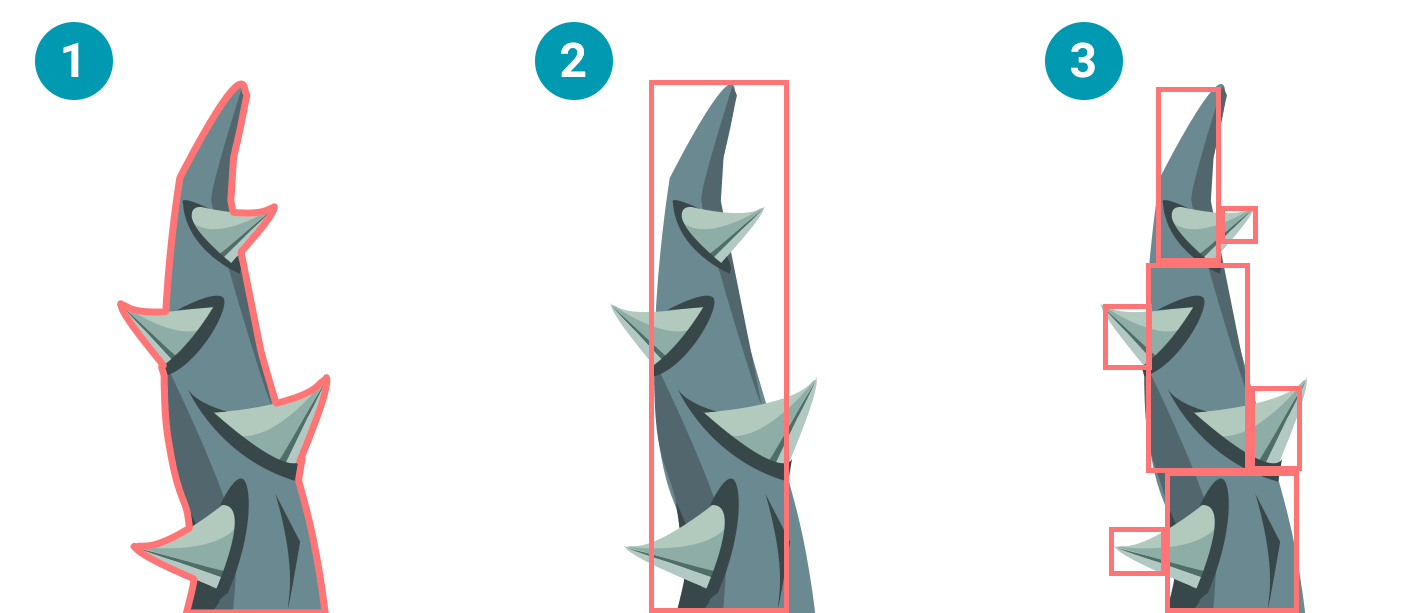
Almost every video game needs to respond to objects touching each other in some sense, a practice commonly known as collision detection.

Whether it’s simply to prevent the player character from walking through the walls of your maze with a simple collision grid array, or if it’s to test if any of the hundreds of projectiles fired by a boss character in a top-down shoot them em-up have struck the player’s ship, your game will likely require a collision detection system of one sort or another.

## Hitboxes

* Hitboxes[[8]](#footnote-7) are **imaginary geometric shapes** around game objects that are used to determine collision detection.
* Imagine you have a player figure. You won't check its arms and legs for collision but instead just check a big imaginary rectangle that's placed around the player.
* The following image demonstrates the different **types** of collision detection. They each have their own **advantages** and **disadvantages**:

1. Pixel perfect - Super precise collision detection, but it requires some serious system resources. In most cases this is an overkill.
2. Hitbox - Much better performance, but the collision detection can be pretty imprecise. In many game scenario's though, this doesn't really matter.
3. Multiple hitboxes - Less efficient then a single hitbox but it still outperforms the pixel perfect variant. And you can support complex shapes. This is a nice option to use for important game objects that need some extra precision, like the player for example. You could make a hitbox for the core and separates ones for arms, legs and the head.



**Common Hitboxes**

* In 2d there are some common geometric shapes for hitboxes:
  + Circles
  + Box (rectangle)

## Bounding sphere/circle test

The simplest of all methods for detecting intersections between objects is a simple bounding sphere test. Essentially, this represents objects in the world as circles or spheres, and test whether they touch, intersect or completely contain each other. This method is ideal when accuracy is not paramount, for objects roughly circular in shape, or in instances where these objects do a lot of rotations. Each object will have a bounding circle defined by a centre point and a radius.

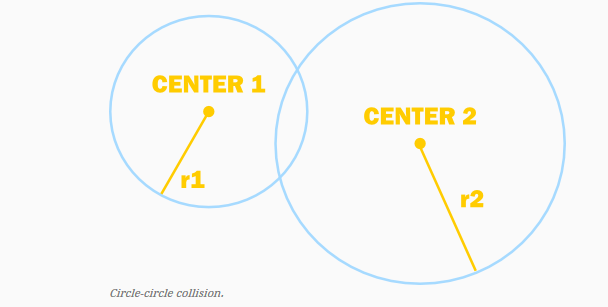


Steps of Circle test

* To test for collision with another bounding circle, all that needs to be done is compare the distance between the two center points with the sum of the two radii:

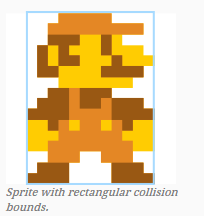
1. f the distance exceeds the sum, the circles are too far apart to intersect.
2. If the distance is equal to the sum, the circles are touching.

If the distance is less than the sum of the radii, the circles intersect



1-2 **Bounding box test**

* The second obvious solution to the problem is to represent obstacles as axis-aligned rectangles. This method is ideal for smaller objects that are roughly rectangular and because it is incredibly fast to process.



## Steps of Box Testing

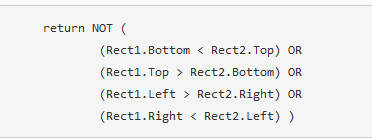
* The method that will be described uses a contradiction to determine whether the rectangles intersect. Because it is simpler instead to determine whether rectangles do ***not* intersect**, the function will calculate that and return the negation of its result.

Rectangles will be defined by their left-, top-, bottom- and right-edges. To determine whether two rectangles **do not intersect**, **one simply has to check for *any* of the following conditions:**

* Rectangle 1’s bottom edge is higher than Rectangle 2’s top edge.
* Rectangle 1’s top edge is lower than Rectangle 2’s bottom edge.
* Rectangle 1’s left edge is to the right of Rectangle 2’s right edge.
* Rectangle 1’s right edge is to the left of Rectangle 2’s left edge.

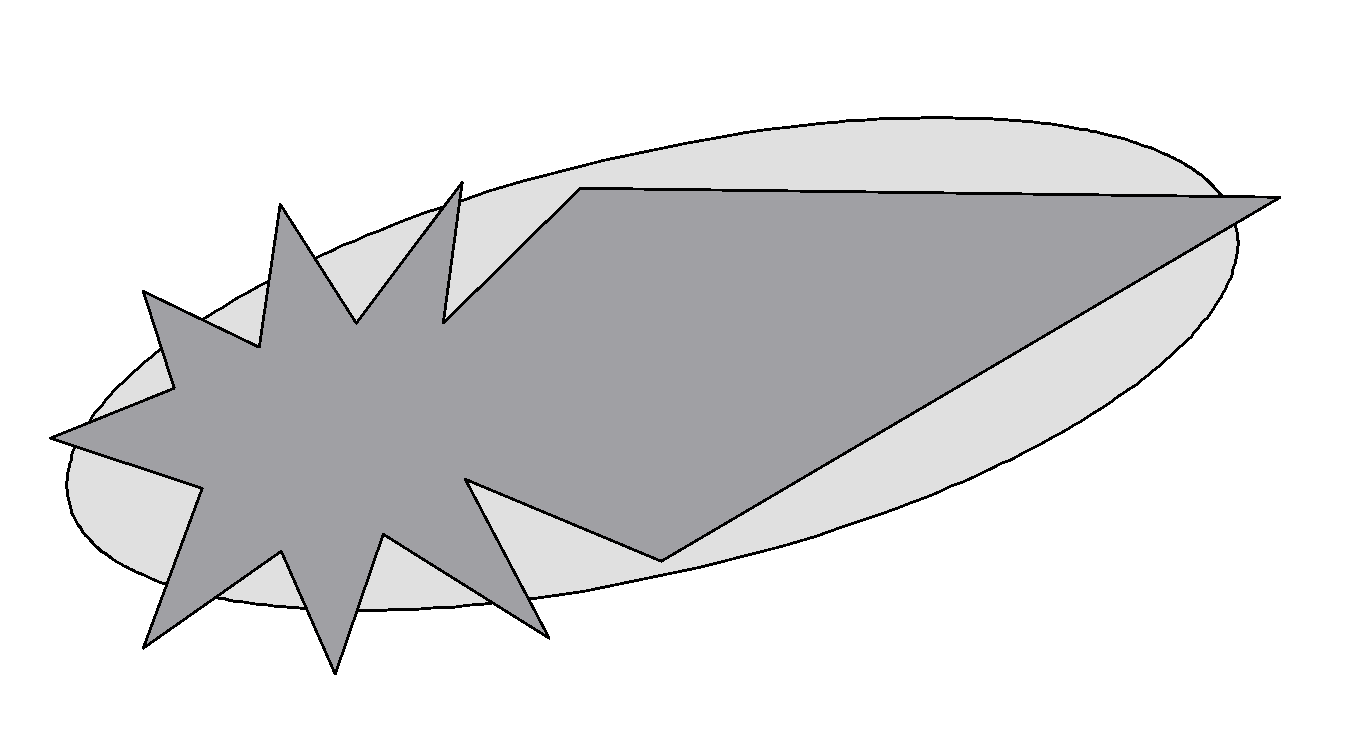


The above can then be condensed into a single line as follows:



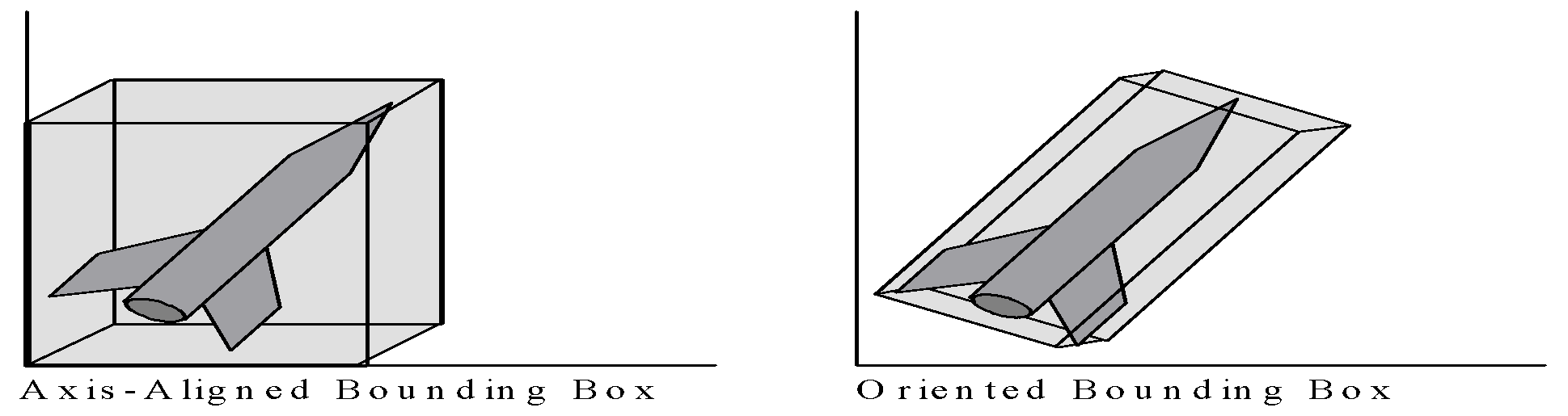
Simplified Geometry

* Approximate complex objects with simpler geometry, like this ellipsoid



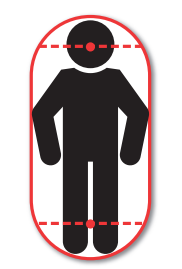
## Axis-Aligned Bounding Box (**AABB**)

* Box edges aligned to **world axes**
* **Recalculate bounds** when object changes orientation
* Collision checks are simplier
* The other type is “**Oriented Bounding Box**”, which rotates with object rotation.



Bounding Capsule

* In 2D, a **capsule** can be thought of as an AABB with two **semicircles** (half circles) attached to the   
  top and bottom.
* If we expand the capsule to 3D, it becomes a **cylinder** with two hemispheres attached to the top and bottom.
* Capsules are also a popular form of collision representation for **humanoid characters**
* **الشخصيات القريبة من شكل الانسان**



**Animation basics[[9]](#footnote-8)**

# What is Animation?

Animation is the art of **bringing life** to an otherwise inanimate objects, or illustrated / 3D generated characters.

It is created by projecting sequenced images quickly, one after another, to create the

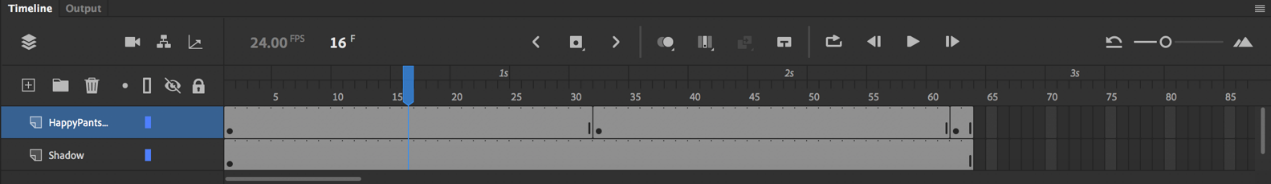
# Animation Vocabulary

Now that we covered the basic idea of what animation is, it's important to understand the vocabulary and specialized terms used in the animation and filmmaking industry.

**These terms will come up often, so let's take a few minutes to go over them:**

## Timeline

The timeline is the part of the animation software that represents the **animation's progress over time**.



Depending on the software, we might use the timeline to make changes to the **timing** of the animation, as well as the **position** of the elements.

## Frame Rate

The frame rate of an animation is the number of individual images (or frames) that are being displayed over the span of one second. It is a setting you can adjust in the animation software.

Animation is usually done in **24 frames per second** (FPS).

## Working on One's & Two's

Working on One's or Two's is a term used in hand drawn animation.

**Working on one's** would mean doing a new drawing over every single frame of the animation.

**Working on twos** means holding each drawing for two frames, so one second of animation at 24 frames per second would only be 12 drawings, not 24.

In 2D animation working on two's looks fine in most instances, and there are even cases where drawings can be held longer. In 3D, though, working on one's is the standard.

## Shots & Scenes

Normally in live action filmmaking, the term '**shot**' refers to the images between camera edits, while a **scene** is all the shots and dialogue that take place at a particular location for a continuous block of time.

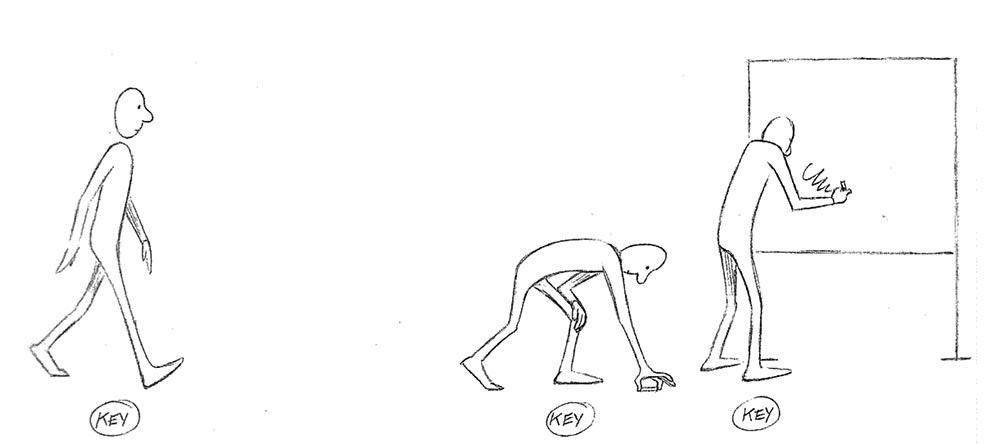
In animation, however, we often use the term 'scene' and 'shot' interchangeably. When we talk about a scene/shot, we often refer to **one specific continuous piece of animation** in between camera cuts.

## Keyframes | Breakdowns | Inbetweens

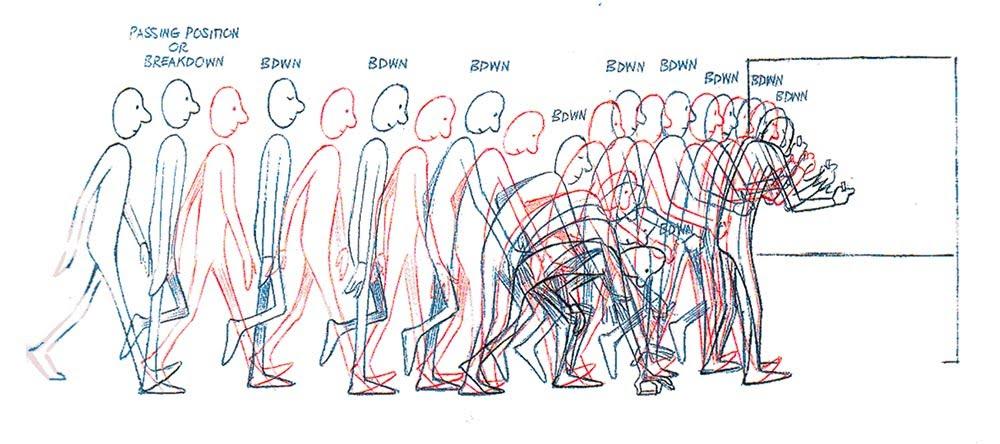
Key frames, breakdowns and in-betweens are important terms, but they mean slightly different things depending on the type of animation.

In hand drawn animation, **keyframes** (or just keys) are the major important poses that define the scene. **Breakdowns** come between keys and define what the motion from key to key will be. **In-betweens** are all the frames that come in between to smooth out the motion.

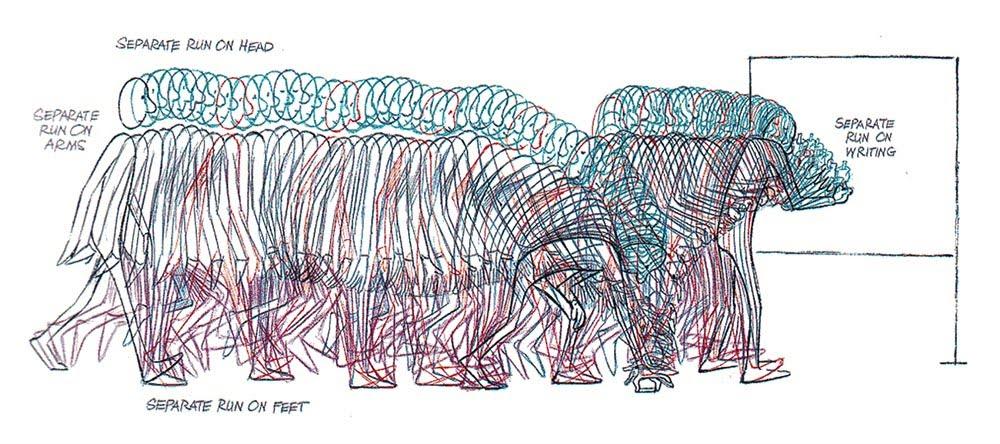
In 3D, a **keyframe** is any position on the timeline where the animator has defined the position of the character. **In-betweens** are all the frames that the computer interprets or automatically generates to move the character from key to key.



Keyframes



Breakdowns



In-betweens

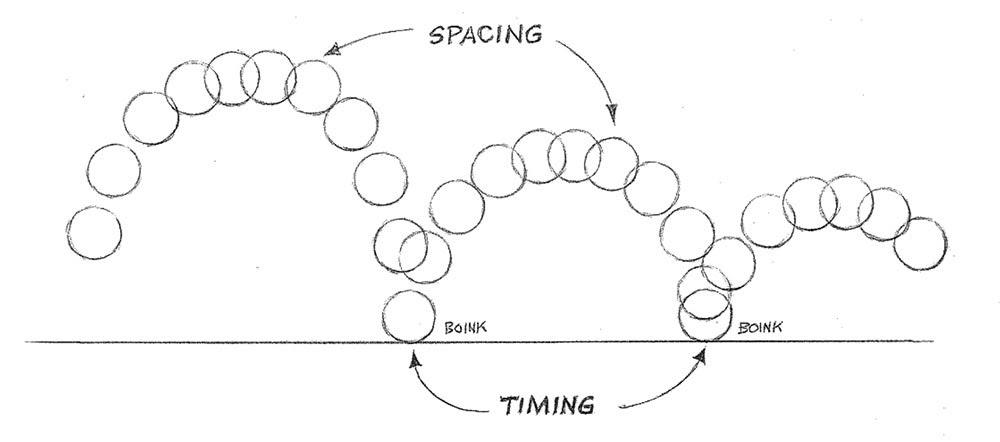
*Images taken from The Animator's Survival Kit / Richard Williams*

## **Timing | Spacing | Easing**

Timing, spacing and easing are closely related terms.

**Timing** means the total number of frames that will be used for a movement. **Spacing** is the amount of change that comes between each frame. Decreasing the spacing, makes an object slower, while increasing the spacing makes it look faster.

In digital animation, **easing** is how spacing is controlled, usually through a motion graph on the timeline.



## Onion Skinning

When animating, it's very useful to be able to see more than one frame at a time.

In paper animation this is done by having multiple drawings on a **light table**, but in modern animation programs there's often a feature called onion skinning. It lets you see semi-transparent representations of the frames behind or ahead of the current frame you're working on.



## Compositing

Compositing is the process of putting all the individual pieces of a scene together to create the **final visual output**.

You might have a background, multiple characters, and some scenery all being developed separately. Compositing is how all those pieces get put together into a single scene.

# The 12 Principles of Animation

The foundation of any animation education is the **12 principles of animation**.

The 12 principles were a set of core concepts that were developed in the 1930's by animators at **Walt Disney Studios** as they were transitioning from doing shorts to feature films. It was a gradual process of discovery and refinement as the animators tried to **push their work to a new higher standard.**

These 12 principles were first compiled by the legendary animators **Frank Thomas** and **Ollie Johnston** in 1981 in their book *The Illusion of Life.*

Creating the illusion of life is what the principles are all about. They help us create characters that look like they have weight, personality, and exist in a real world with real physics at work.

**Even though they were developed by 2D animators, they still apply to 3D and any other type of animation.**

## Squash & Stretch

Squash and stretch describe how an object changes shape in response to forces acting on it.

**Squash** is when the object is compressed by an impact of an opposing force. **Stretch** is when an object is distended by something pulling on it, or by moving quickly.

## Anticipation

Anticipation is a smaller movement that comes before a major one, and signals that the major movement is about to happen.

## Staging

Staging is the presentation of a shot in a way that makes the content of the shot as **clear** as possible, and the narrative function of the shot as **strong** as possible.

## Straight ahead vs. Pose-to-pose

Straight-ahead and pose-to-pose are different approaches to animating.

**Straight-ahead** means creating each new frame in sequence from beginning to end. **Pose-to-pose** means creating the key poses for each action first, and then filling in the in-between poses.

## Follow Through & Overlapping Action

Follow-through and overlapping action refers to the tendency of different parts of a body to move at different speeds.

This includes the concept of **drag**, which is when one part of the body lags behind when a motion starts.

## Slow in & Slow out

Slow-in and slow-out refer to the tendency of objects to gradually **accelerate** (and then **decelerate**) when moving from one position to another.

These are sometimes referred to as **ease-in** and **ease-out**, or simply **easing**.

## Arcs

The principle of arcs come from the observation that living things don't move in straight lines, but rather in **curved motions**.

Creating graceful, clear arcs often elevates the animation and reveals the experience level of the animator.

## Secondary Action

Secondary action refers to **smaller movements** (or gestures) that **support** the primary actions of a character.

These actions make the shot clearer by emphasizing the **attitude** or **motivation** behind the movement.

## Timing

Timing is controlling the **speed** of an action through the number of frames used to represent it.

It is one of the **most fundamental** of the 12 principles and takes years to master.

## Exaggeration

Exaggeration means representing a subject in a **heightened** or more **extreme** way, rather than strictly realistic, in order to push your animation further.

## Solid Drawing

Solid drawing means posing characters in a way that creates a sense of **volume**, **weight** and **balance**.

Drawing for animation requires being able to draw the characters from any angle or pose, with three-dimensionality in mind.

## Appeal

Appeal is a broad term for any qualities of a character's design that makes them inherently **compelling** to watch.

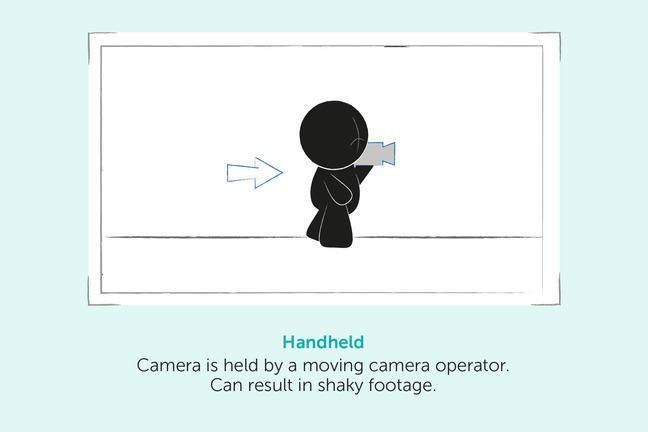
This includes the design of the character, as well as how the character is animated.

# Camera movement[[10]](#footnote-9)

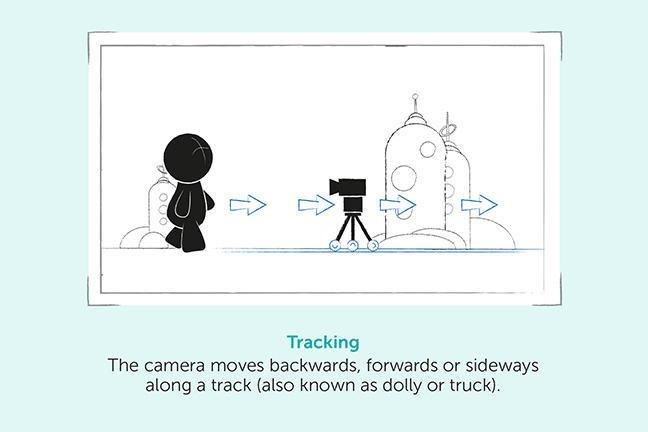
Camera movement is one of the most useful tools in a filmmaker’s armoury. It can be used to reveal information to the audience, or move closer to the subject, providing clearer visual information.

There are a variety of ways of moving the camera:

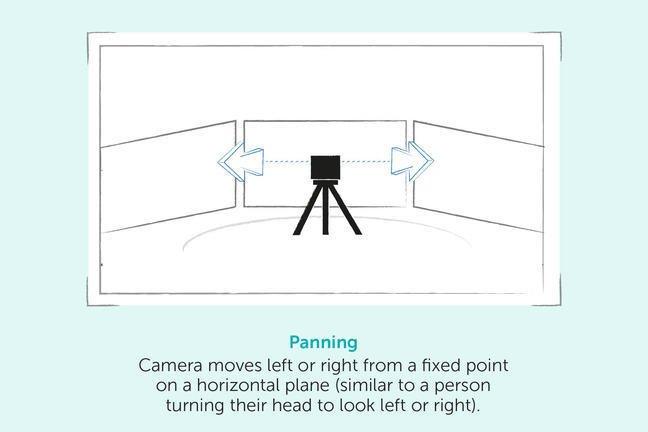
## Handheld



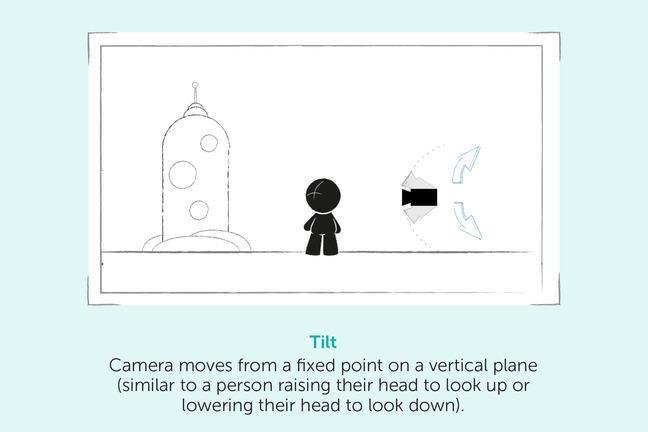
## Tracking



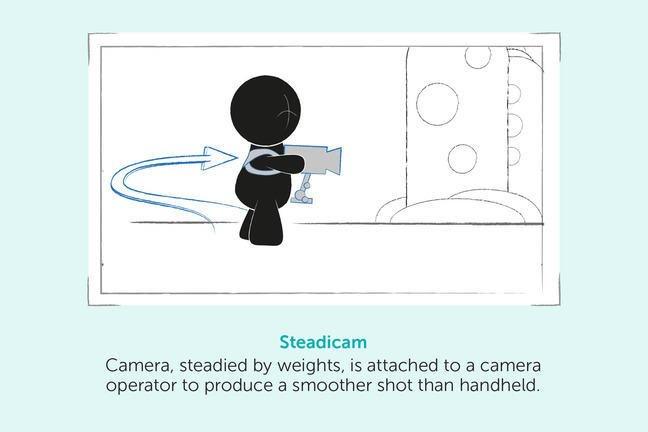
## Panning



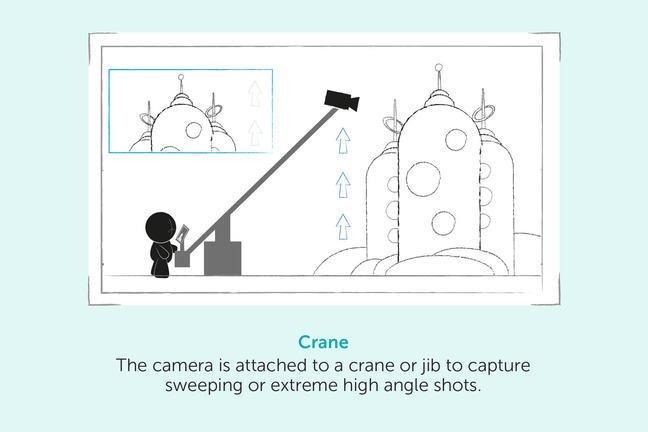
## Tilt



## Steadicam



## Crane



# Case study: Physics Joints in Unity 2D[[11]](#footnote-10)

The cross-platform game engine Unity has powerful support for creating 2D and 3D games. It’s a great choice for aspiring game developers, since it works for most mobile, desktop and console platforms, and even better, it’s free to use for lower-revenue developers and studios.

One of the key components of Unity are *physics joints*, which let you create various connections between objects in Unity. Using joints, you can describe a connection between two objects, which means you can simulate the physics of almost any multi-part object you can think of, including doors, sliding platforms, chains or even wrecking balls

the demo can be downloaded from here: <https://koenig-media.raywenderlich.com/uploads/2015/05/2D_Joints_Starter.zip>

and the main tutorial can be read here:

<https://www.raywenderlich.com/1766-physics-joints-in-unity-2d>

1. * References
     + <http://blog.wolfire.com/2009/07/linear-algebra-for-game-developers-part-1/>
     + <http://blog.wolfire.com/2009/07/linear-algebra-for-game-developers-part-2/>
     + <http://blog.wolfire.com/2010/07/Linear-algebra-for-game-developers-part-3>

   [↑](#footnote-ref-0)
2. <https://gamedevelopertips.com/vector-in-game-development/> [↑](#footnote-ref-1)
3. <https://blog.nobel-joergensen.com/2010/10/22/spherical-coordinates-in-unity/> [↑](#footnote-ref-2)
4. <https://vvvv.org/blog/polar-spherical-and-geographic-coordinates> [↑](#footnote-ref-3)
5. <https://vvvv.org/blog/polar-spherical-and-geographic-coordinates> [↑](#footnote-ref-4)
6. <http://archive.gamedev.net/archive/reference/programming/features/curvessurfaces/> [↑](#footnote-ref-5)
7. Based on Textbook 2: “Game Programming Algorithms and Techniques- A Platform-Agnostic Approach”, Sanjay Madhav,2014, ch1 [↑](#footnote-ref-6)
8. <https://spicyyoghurt.com/tutorials/html5-javascript-game-development/collision-detection-physics>

   [↑](#footnote-ref-7)
9. <https://www.bloopanimation.com/animation-for-beginners/> [↑](#footnote-ref-8)
10. <https://www.futurelearn.com/info/courses/filmmaking-animation-classroom/0/steps/23225> [↑](#footnote-ref-9)
11. <https://www.raywenderlich.com/1766-physics-joints-in-unity-2d> , also it has demo: <https://koenig-media.raywenderlich.com/uploads/2015/05/2D_Joints_Starter.zip> [↑](#footnote-ref-10)