Mansoura University
Faculty of Computers and Information
First Semester- 2022-2023

COMPUTER GRAPHICS

Grade: 2ND YEAR (GENERAL –BIO-AI)

Course Instructors :

Assoc. Prof. Dr-Haitham El-Ghareeb

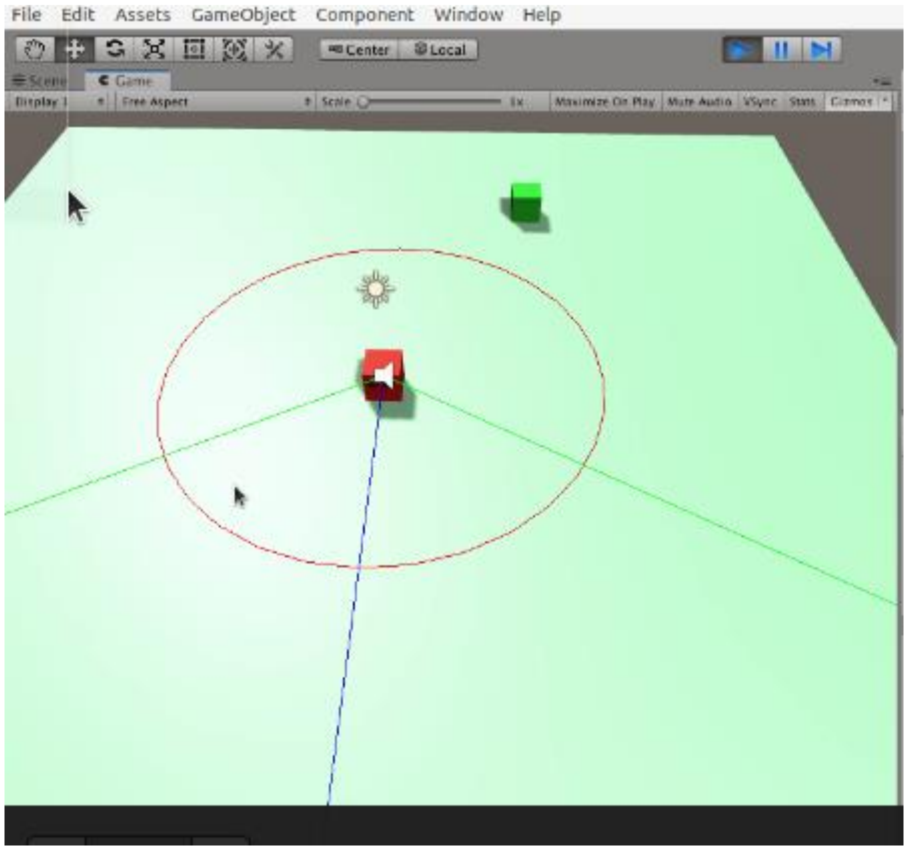
Dr-Nabila Hamed,

waleed mohamed

Chapter 06- Math and Animation for game Developers

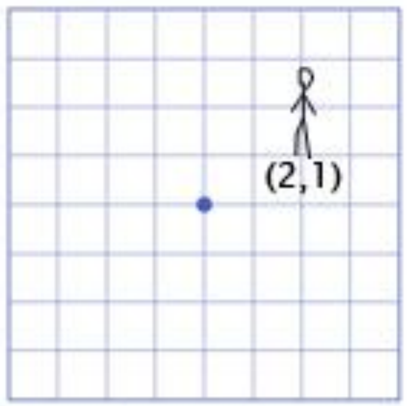
Lecture Demo Application in unity

- Source code and explanation videos are available at Google Drive Link

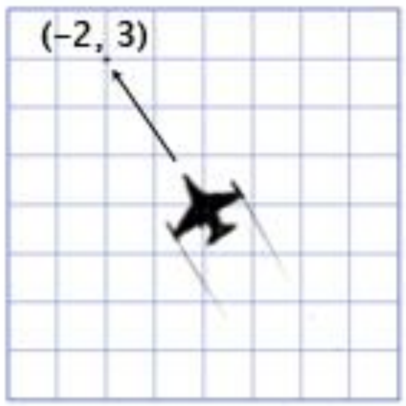


Vector

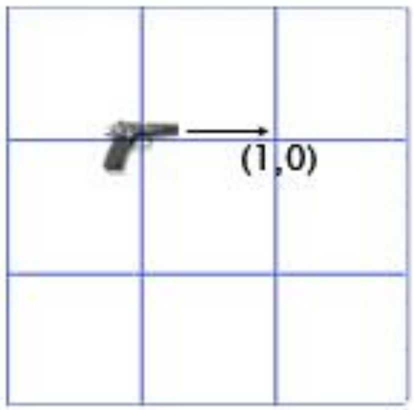
- In games, vectors are used to store:
  - **positions**,
  - **directions**,
  - and **velocities**.
- Here are some 2-Dimensional examples:
  - The **position** vector indicates that the man is standing two meters east of the origin, and one meter north.
  - The **velocity** vector shows that in one minute, the plane moves three kilometers up, and two to the left.
  - The **direction** vector tells us that the gun is pointing to the right.



Position



Velocity



Direction

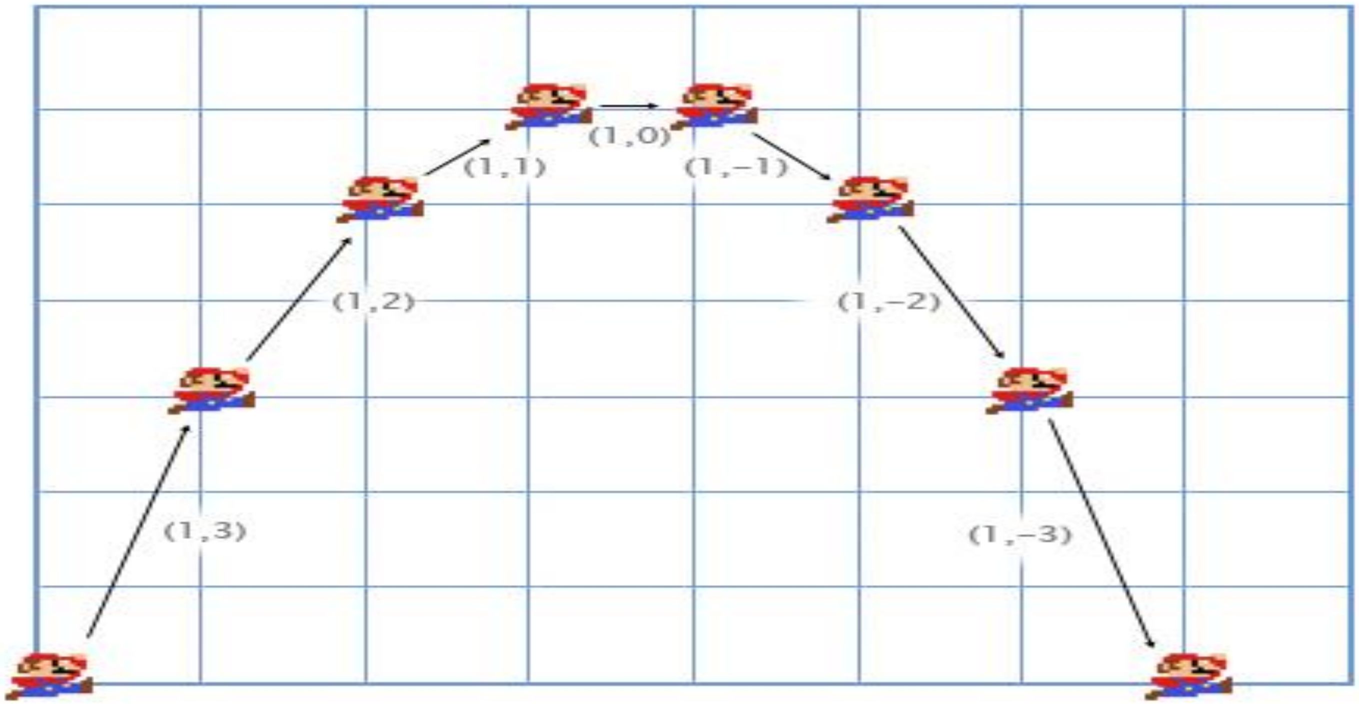
Vector addition

- 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)
- 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**
- **(العجلة)التغير في سرعة الحركة**.

Vector addition Example

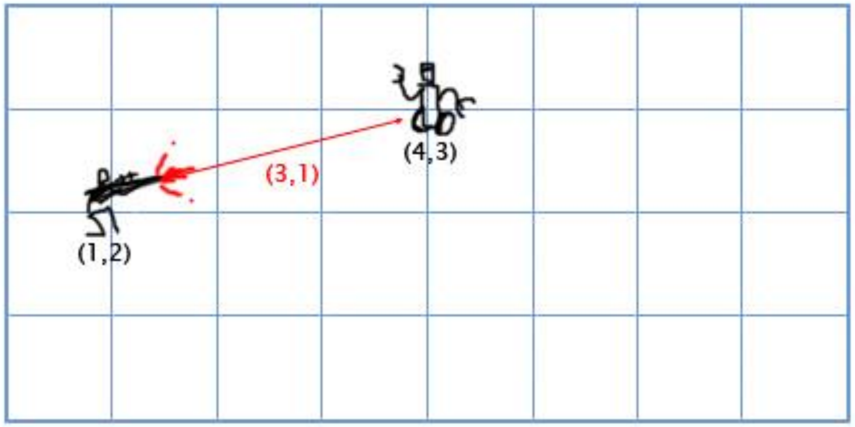
- Let's consider the example of Mario jumping:
  - He starts at position (0,0). As he starts the jump, his velocity is (1,3) -- he is moving upwards quickly, but also to the right.
  - His **acceleration** throughout is (0,-1), because **gravity** is pulling him **downwards**. Here is what his jump looks like in **7 frames**. The black text specifies his velocity for each frame.



Vector subtraction

- 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).

Vector **Length (magnitude)**

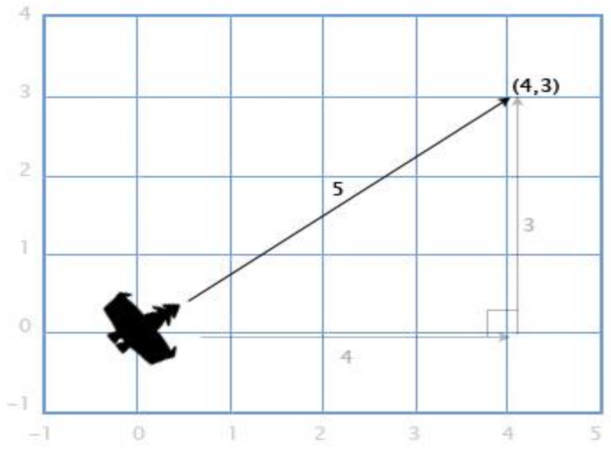
- If we have a ship with velocity vector V (4,3), we might also want to know how fast it is going, in order to calculate how much fuel it should use.
- To do that, we need to find the **length** (or **magnitude**) of vector V. The length of a vector is often written using || for short, so the **length of V is |V|.**
- We can think of V as a right triangle with sides 4 and 3, and use the Pythagorean theorem نظرية فيثاغورث to find the **hypotenuse:** $\mathbf{x^2 + y^2 = h^2}$.
- That is, the length of a vector H with components (x,y) is $\mathbf{sqrt(x^2+y^2)}$

الجذر التربيعي

. So, to calculate the speed of our ship, we just use:

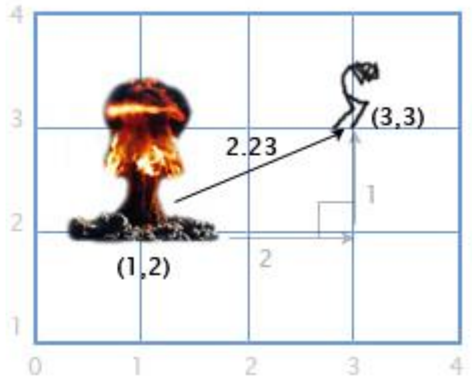
$$|V| = sqrt(4^2+3^2) = sqrt(25) = 5$$

- This works with 3D vectors as well -- the length of a vector with components (x,y,z) is $sqrt(x^2+y^2+z^2)$.



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(2^2+1^2) = 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 programming a lot easier.
- A vector with a length of 1 is called "normalized". So how do we normalize a vector (set its length to 1)? **we divide each component by the vector's length**.
- Example: 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:

$$(^3/_5)^2 + (^4/_5)^2 = {}^9/_{25} + {}^{16}/_{25} = {}^{25}/_{25} = 1$$

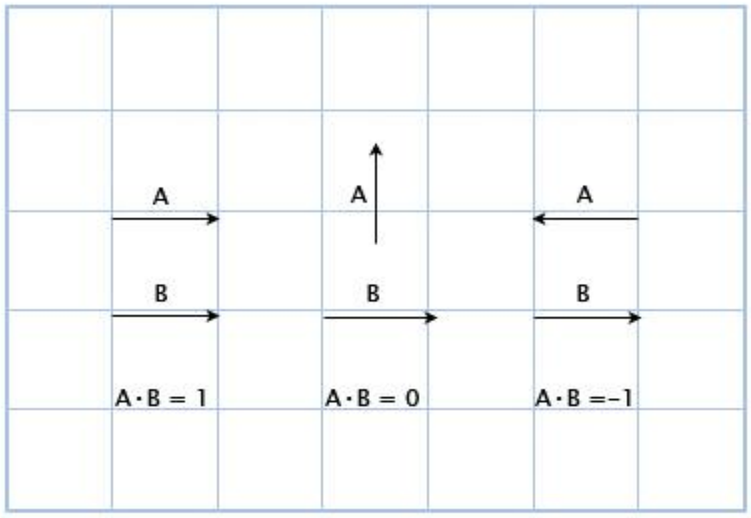
Dot product

- To get the dot product of two vectors, we multiply the components, and then add them together.
- $(a_1,a_2)\bullet(b_1,b_2) = a_1b_1 + a_2b_2$
- For example, $(3,2)\bullet(1,4) = 3*1 + 2*4 = 11$.
- *This seems kind of useless at first, but lets look at a few examples in next slides*

Example 1: Dot Product

- Example 1:
  1. Here, we can see that when the vectors are pointing the **same direction** , the dot product is **positive**.
  2. When they are **perpendicular** , the dot product is **zero**, متعامدان
  3. 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!

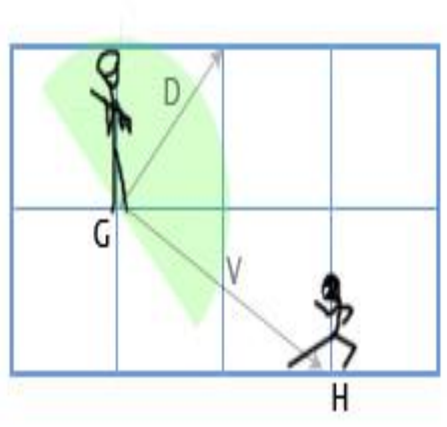


Example 2: Dot Product

- Example 2 : 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!



Using Dot Product to get angle

- 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:

  $$AB = |A||B|\cos\theta$$

- 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:

  $$\theta = \text{acos}([AB]/[|A||B|]).$$

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

  $$\theta = \text{acos}(AB)$$

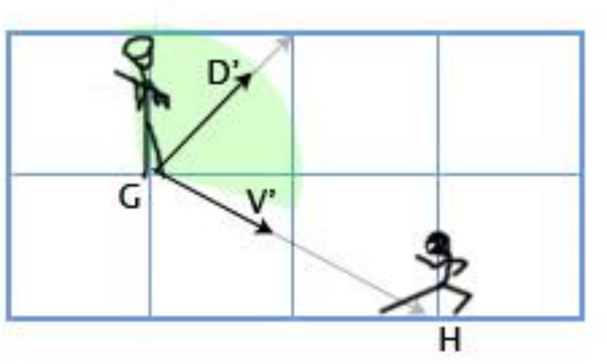
Example 3: Using Dot Product to Get Angle

- Let's revisit the guard scenario above, **<u>except the guard's field of view is only 120</u>**.
  - 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(1^2+1^2) = (1,1)/sqrt(2) = **(0.71,0.71)**
  V' = V/|V| = (2,-1)/sqrt(2^2+$(-1)^2$) = (2,-1)/sqrt(5) **= (0.89,-0.45)**

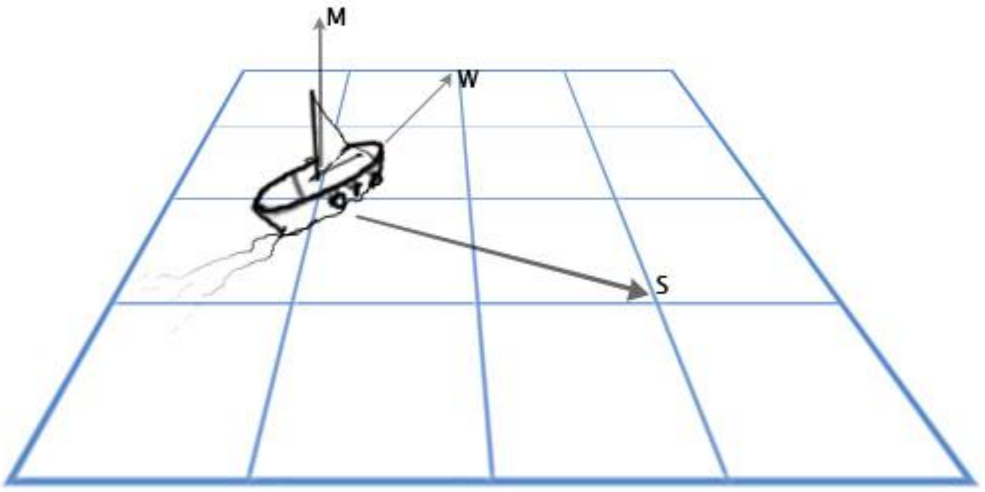
  θ = acos(D'V') = acos(0.71*0.89 + 0.71*(-0.45)) = acos(0.31) = **72**

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



Cross Product

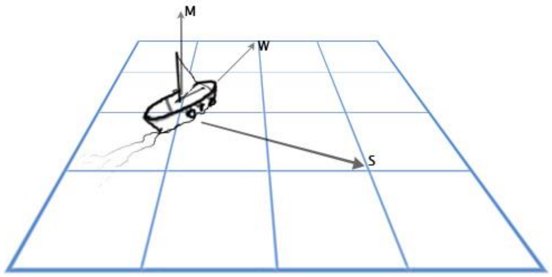
- Example : in a sailing ship, We have a vector for the direction of the mast M, going straight up (0,1,0), and the direction of the north-north-east wind W (1,0,2), and we want to find the direction the sail S should stick out in order to best catch the wind. The sail has to be perpendicular to the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



Cross Product-3 (Continue example)

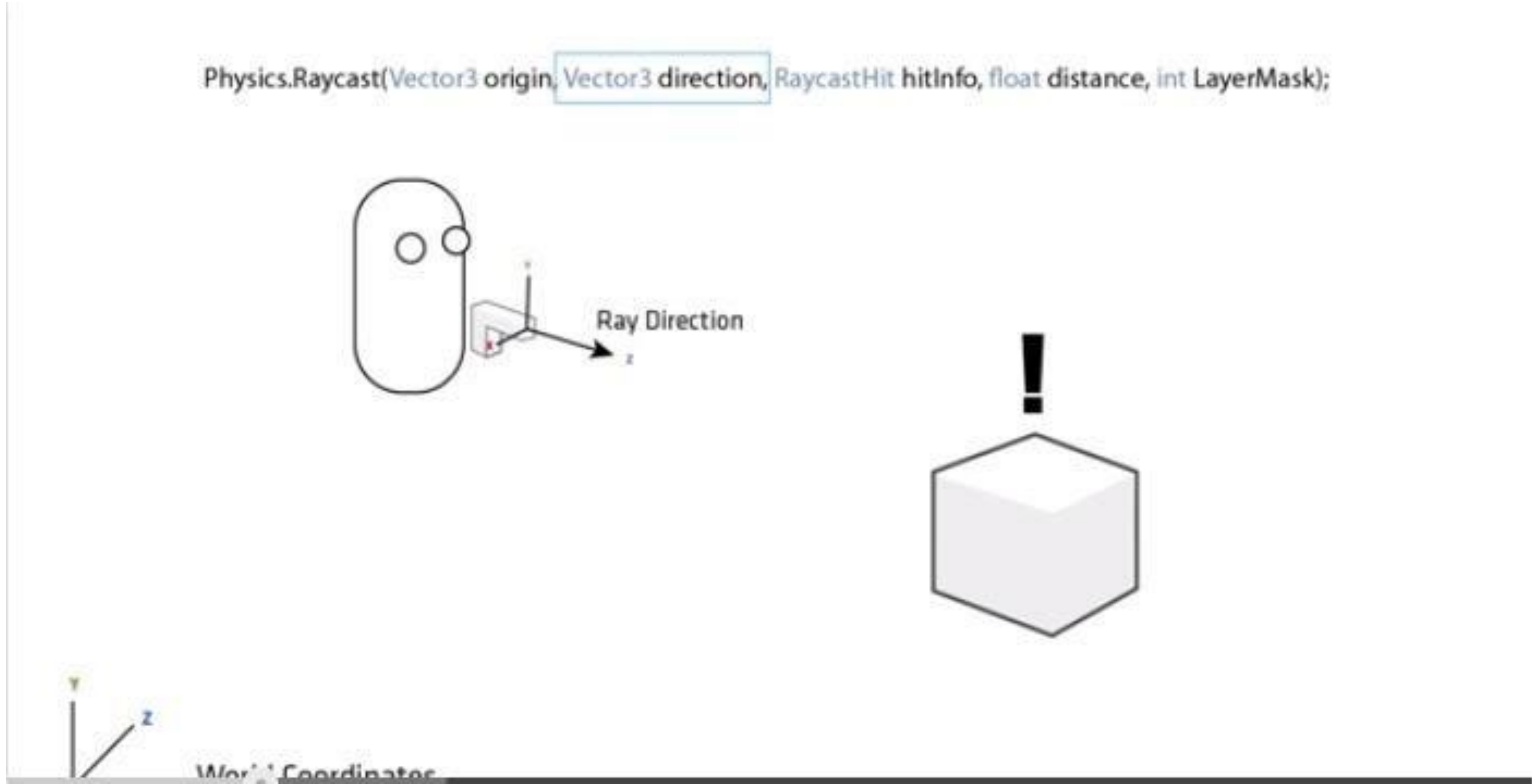
- The cross product of A(a_1,a_2,a_3)) and B(b_1,b_2,b_3)) is:
  (a_2b_3-a_3b_2, a_3b_1-a_1b_3, a_1b_2-a_2b_1)
- So now we can plug in our numbers and solve our problem:

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



Ray casting
شعاع افتراضي لاختبار وجود عوائق –

- 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.



```
Physics.Raycast(Vector3 origin, Vector3 direction, RaycastHit hitInfo, float distance, int LayerMask);
```

LINEAR INTERPOLATION (LERP)

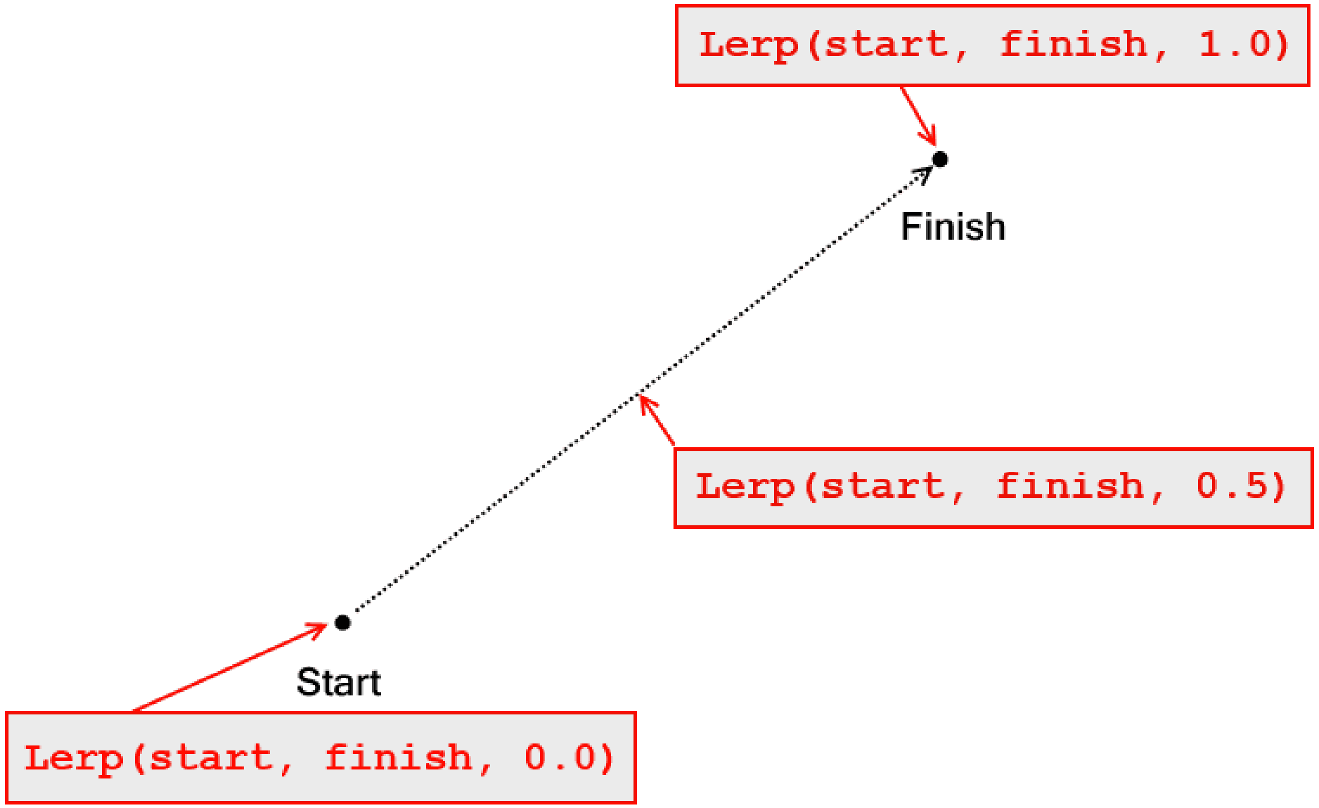
The linear interpolation (LERP)

- 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: المعادلة للاطلاع فقط

$$
\begin{aligned}
\mathbf{L} = \mathrm{LERP}(\mathbf{A},\mathbf{B},\beta) &= (1-\beta)\mathbf{A} + \beta\mathbf{B} \\
&= [(1-\beta)A_x + \beta B_x,\quad (1-\beta)A_y + \beta B_y,\quad (1-\beta)A_z + \beta B_z].
\end{aligned}
$$

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

Example


Lerp(start, finish, 1.0)
Finish
Lerp(start, finish, 0.5)
Start
Lerp(start, finish, 0.0)

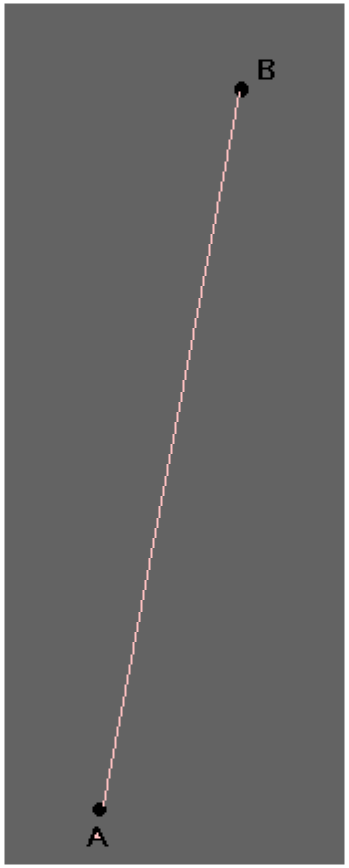
Parametric curves للاطلاع فقط

- You are probably familiar with curves like **$y = x^2$** or **$f(x)=x^2$**. 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)**.

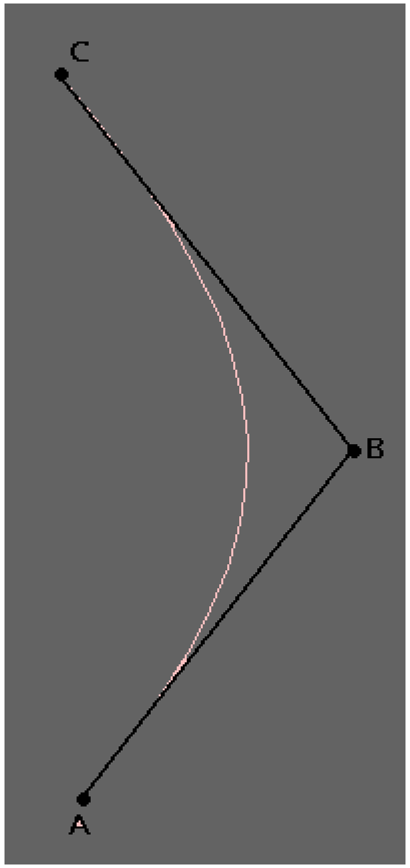
Bezier curves: 1- Simple line للاطلاع فقط

- 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**.



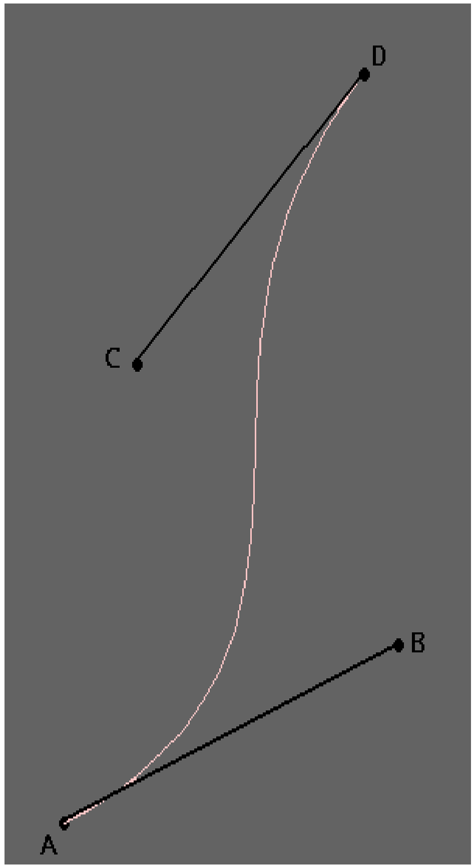
2-Quadratic (curves) للاطلاع فقط

- 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:
- $(a+b)^2 = a^2 + 2a \cdot b + b^2$
- Knowing that **b=1-a** we see that $a^2 + 2 \cdot a \cdot b + b^2$ is still equal to one, since **(a+b) = 1**, and $1^2 = 1$.
- 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.
- $X(a) = Ax \cdot a^2 + Bx \cdot 2 \cdot a \cdot b + Cx \cdot b^2$
- $Y(a) = Ay \cdot a^2 + By \cdot 2 \cdot a \cdot b + Cy \cdot b^2$
- $Z(a) = Az \cdot a^2 + Bz \cdot 2 \cdot a \cdot b + Cz \cdot b^2$
- Still, if you set **a = 0.5**, then $a^2 = 0.25$, $2 \cdot a \cdot b = 0.5$ and $b^2 = 0.25$ 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 $a^2 = 1.0$, $a \cdot b = 0.0$ and $b^2 = 0.0$ 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**.



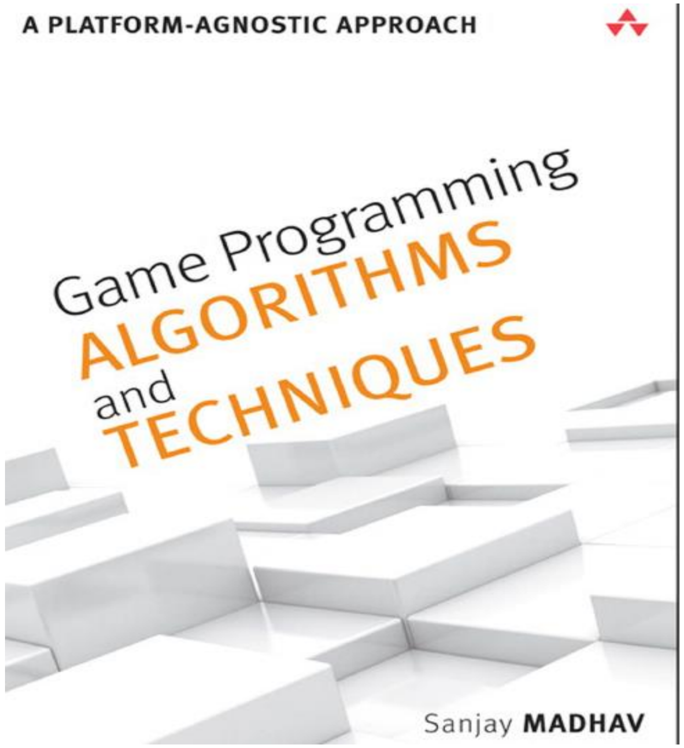
3-Cubicللاطلاع فقط

- 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.
- **$(a+b)^3 = a^3 + 3 \cdot a^2 \cdot b + 3 \cdot a \cdot b^2 + b^3$**
- Now we need four control points **A**, **B**, **C** and **D**, where **A** and **D** are the end points.
- **$X(a) = Ax \cdot a^3 + Bx \cdot 3 \cdot a^2 \cdot b + Cx \cdot 3 \cdot a \cdot b^2 + Dx \cdot b^3$**
- **$Y(a) = Ay \cdot a^3 + By \cdot 3 \cdot a^2 \cdot b + Cy \cdot 3 \cdot a \cdot b^2 + Dy \cdot b^3$**
- **$Z(a) = Az \cdot a^3 + Bz \cdot 3 \cdot a^2 \cdot b + Cz \cdot 3 \cdot a \cdot b^2 + Dz \cdot b^3$**



Basics of Game programming

- Based on Textbook 2: “Game Programming Algorithms and Techniques- A Platform-Agnostic Approach”, Sanjay Madhav,2014, ch1

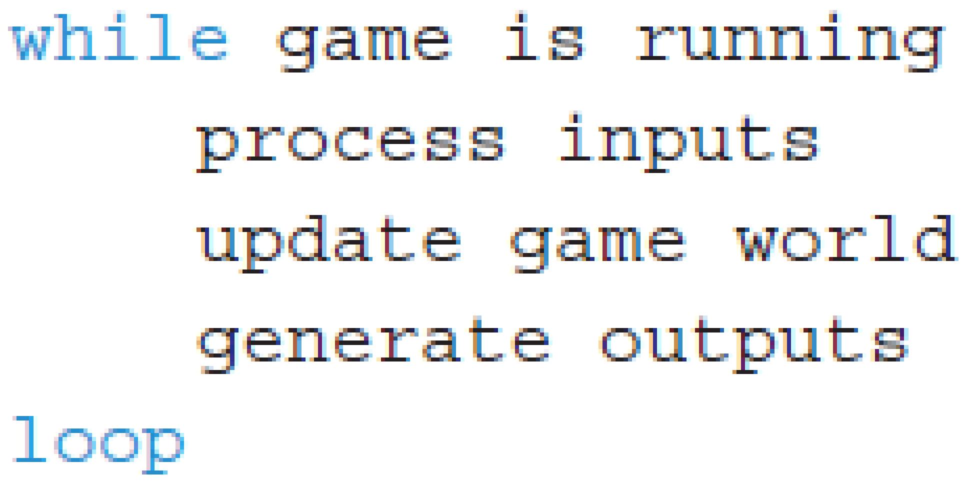


The Game Loop

- 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.

Traditional Game Loop

- 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:

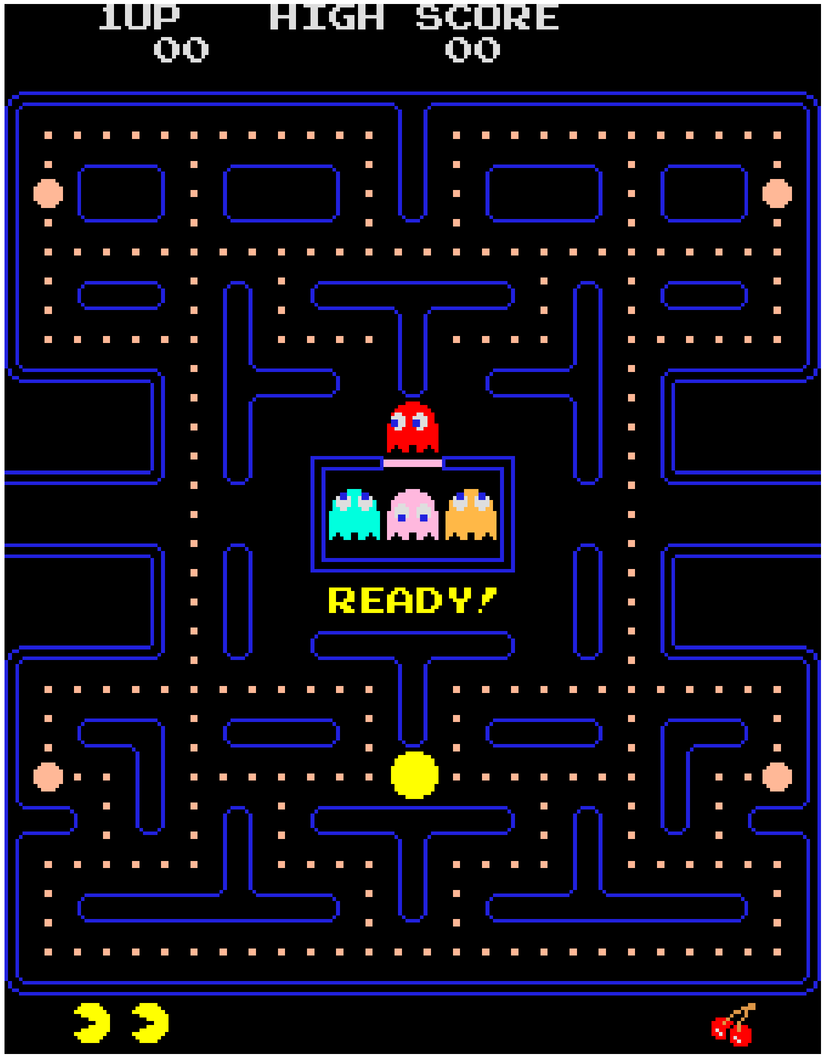


```
while game is running
    process inputs
    update game world
    generate outputs
loop
```

Game Loop (continue)

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
   - Others…

Example: pacman game



Example: Theoretical *Pac-Man* Game Loop

```
while player.lives > 0
    // Process Inputs
    JoystickData j = grab raw data from joystick

    // Update Game World
    update player.position based on j
    foreach Ghost g in world
        if player collides with g
            kill either player or g
        else
            update AI for g based on player.position
        end
    loop

    // Pac-Man eats any pellets
    ...

    // Generate Outputs
    draw graphics
    update audio
loop
```

Multithreaded Game Loops

- Some mobile and independent Games still use a variant of the traditional game loop.
- But most Large Games do not.
- That's because newer hardware features CPUs that have **multiple cores**.
- This means the CPU is physically capable of running multiple lines of execution, or **threads** , at the same time.

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.

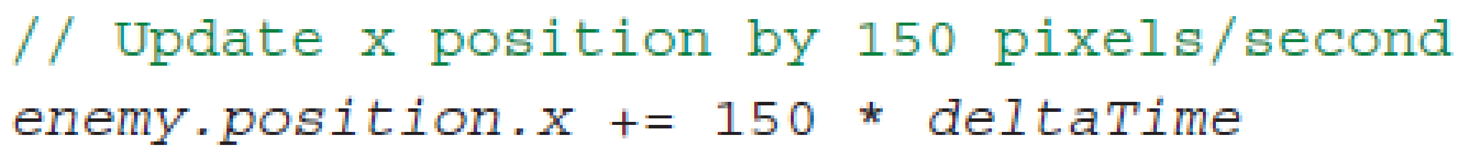
Logic as a Function of Delta Time

- 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).

```
// Update x position by 5 pixels
enemy.position.x += 5
```

Solution: concept of **delta time**

- concept of **delta time** : *the amount of elapsed game time since the last frame.*
- جزء من الثانية
- 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



```
// Update x position by 150 pixels/second
enemy.position.x += 150 * deltaTime
```

2D GRAPHICS

Why 2D games are famous?

- Suitable to **Mobile** and **web** Platforms
- <u>**Developers**</u> are drawn to 2D because the typical **budget** and team can be much **smaller**.
- <u>**Gamers**</u> are drawn toward 2D because of the **simplicity** of the games.
- Many techniques are common between 2D and 3D games

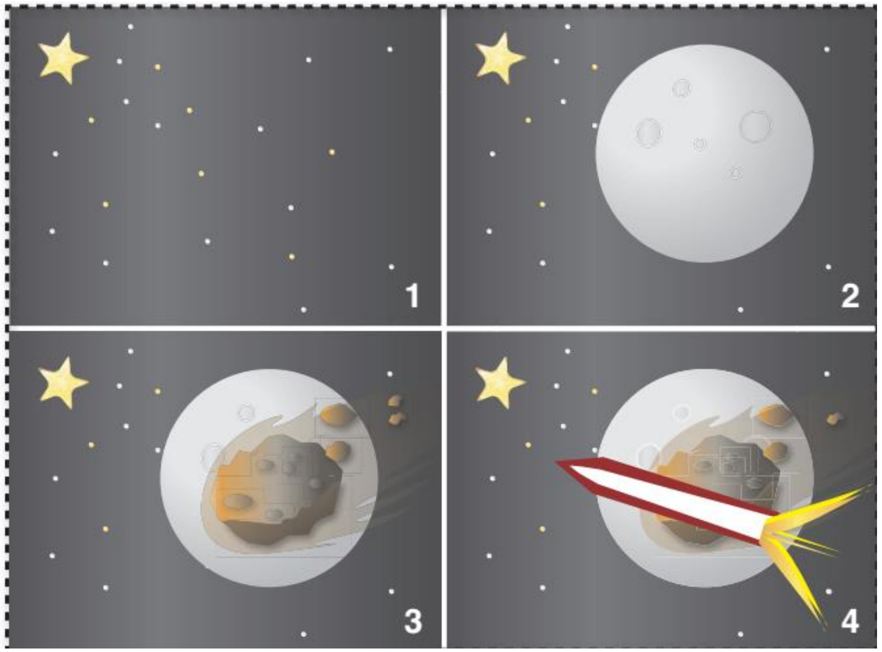
Sprites

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

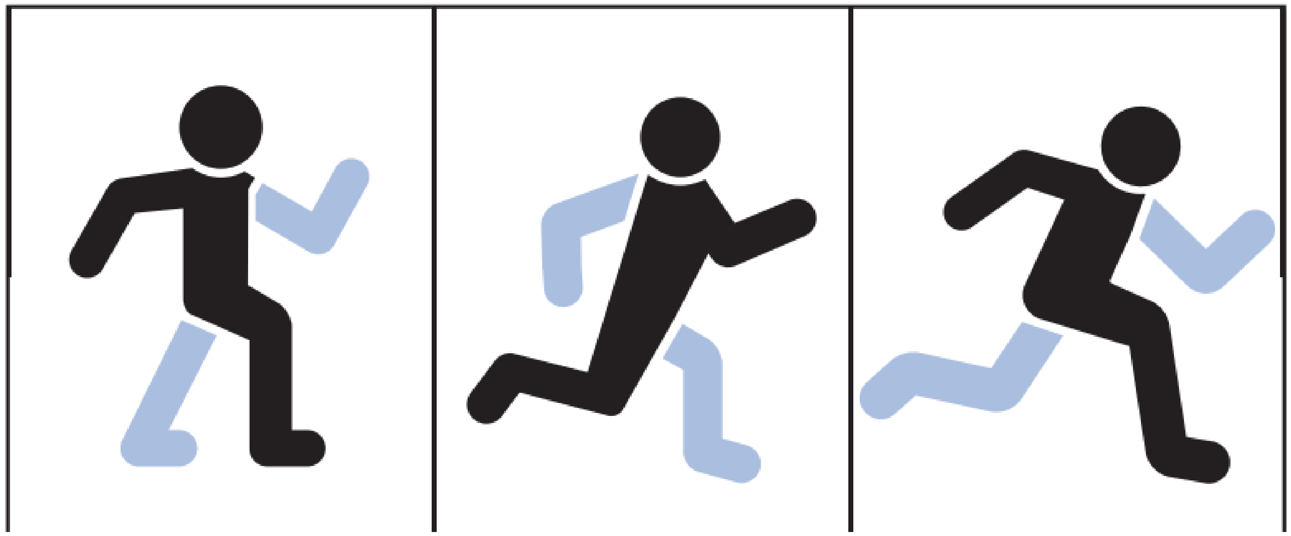
Drawing Sprites

- 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 <u>**sorted from back to front (based on Z-Order/Depth value)**</u>



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

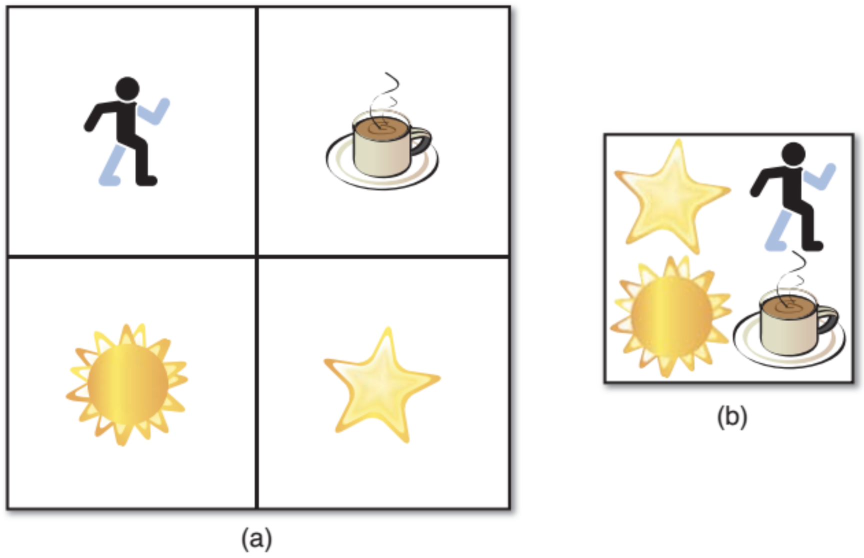


Sprite Sheets

- 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** .

Sprite Sheets (2)

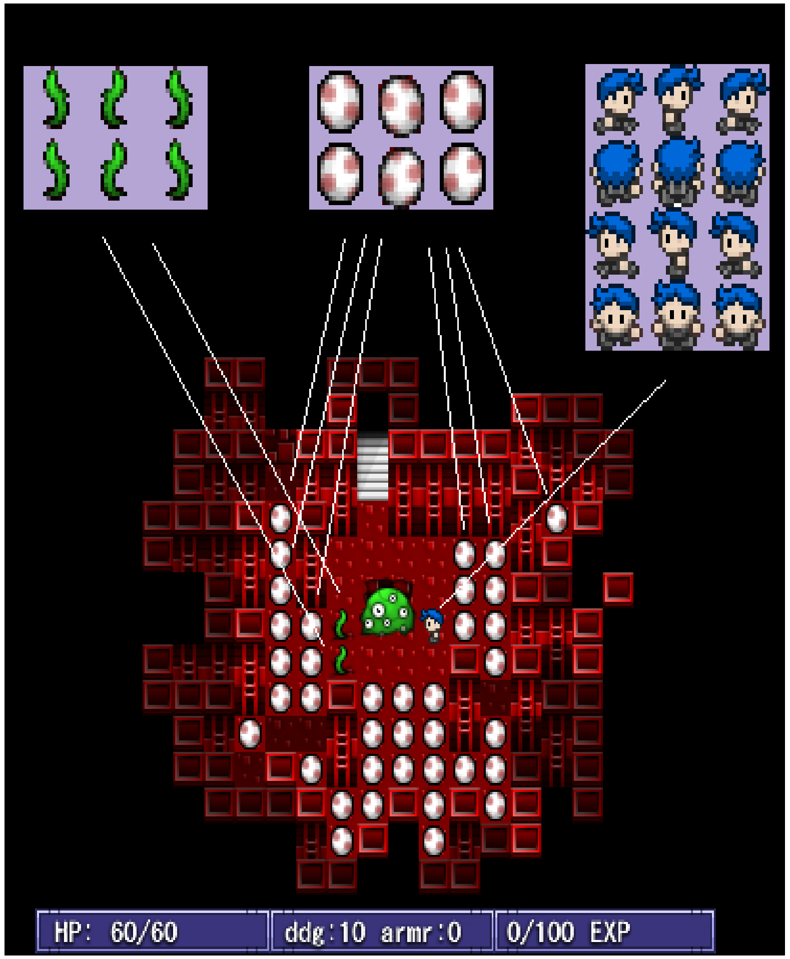
- In a sprite sheet, sprites are grouped **closely** and **overlap the unused space تقاطع**.
- This means that when the sprite sheet is opened, a bit of work is necessary to **reconstruct the correct images in memory**.
- This **reduces** total **installation size** of the game.



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…



Relative Paths

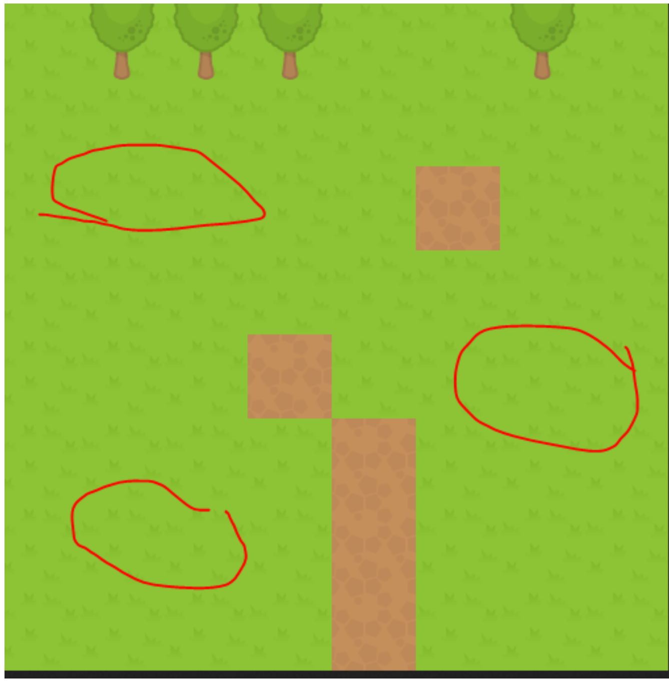
- For All Resource Files:
  - Don’t use absolute paths
  - “/gpfs/main/home/<login>/course/cs1971/tac/resources/spritesheet.png” is bad
  - “resources/spritesheet.png” is good
  - Absolute filepaths won’t work when we/your classmates try to test your game

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.

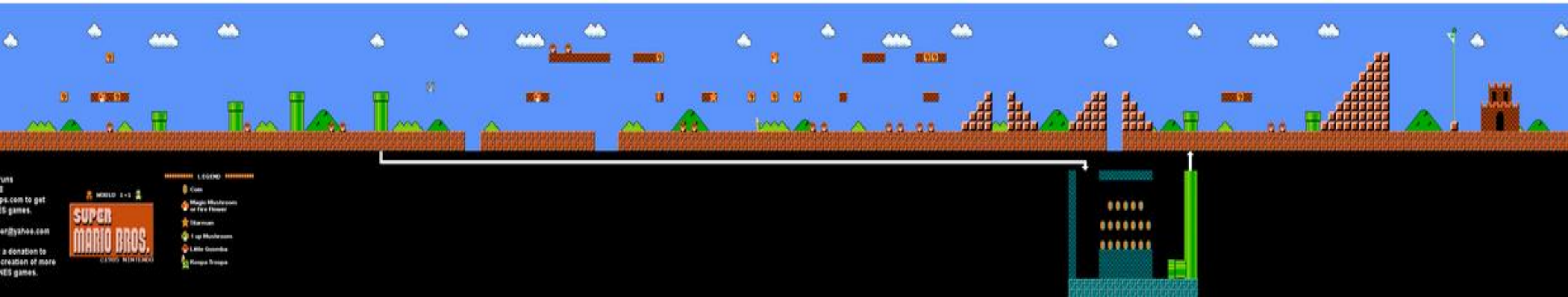
- **Alternative** to creating large background images for games.

- Useful for **repeating patterns**



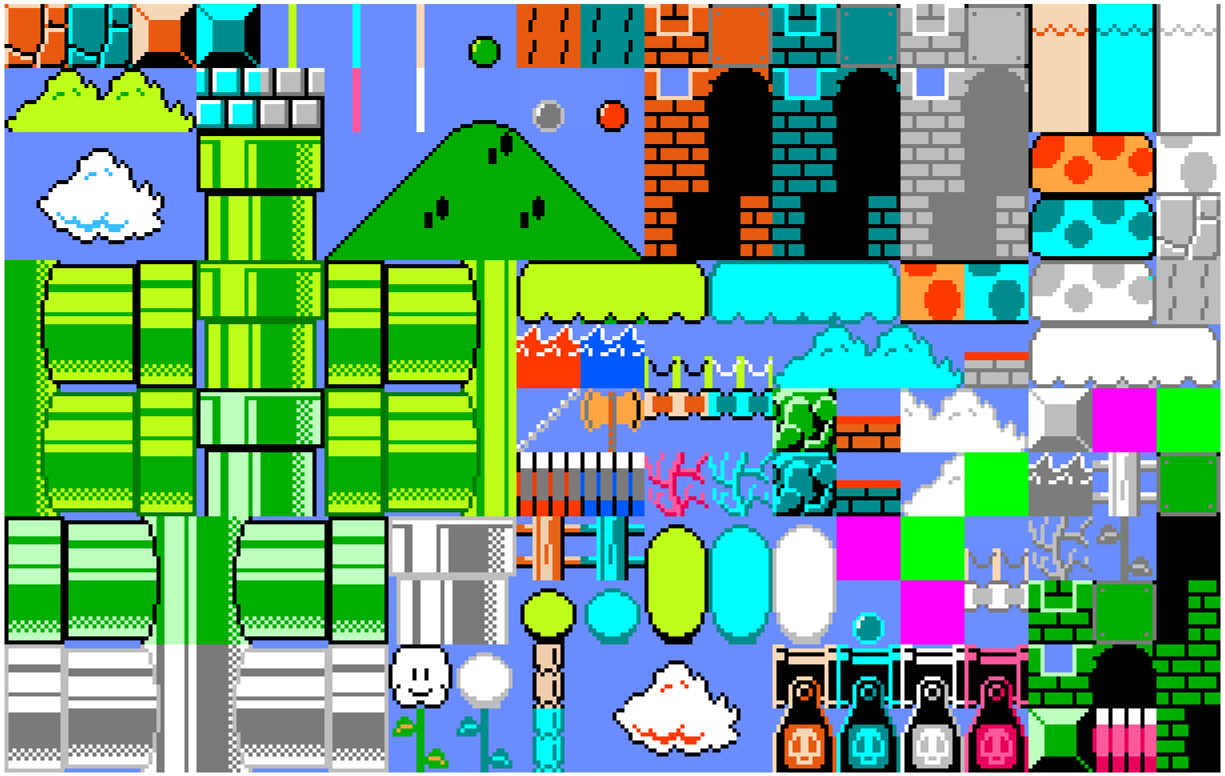
Example

- 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?



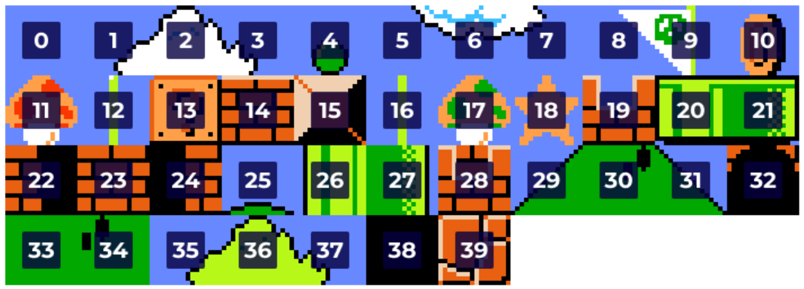
Example-continue

- 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*.

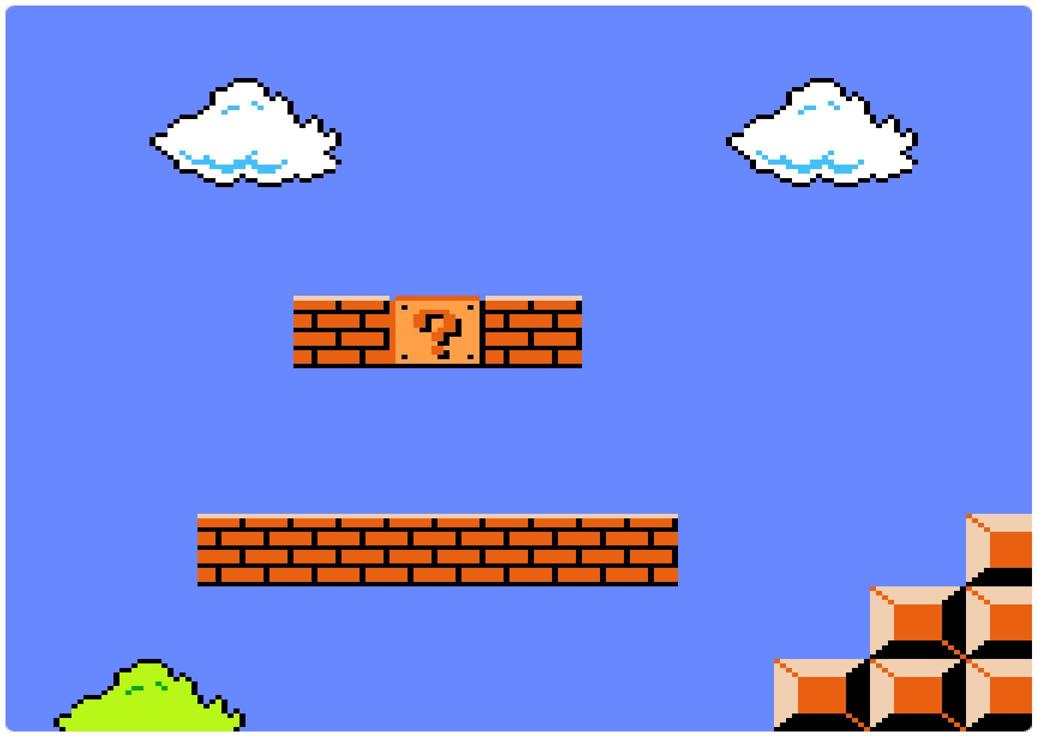


Example-continue 2

- We can represent tilemap as 2D matrix in code



```
// Load a map from a 2D array of tile indices
const level = [
  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],
  [  0,   1,   2,   3,   0,   0,   0,   1,   2,   3,   0 ],
  [  0,   5,   6,   7,   0,   0,   0,   5,   6,   7,   0 ],
  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],
  [  0,   0,   0,  14,  13,  14,   0,   0,   0,   0,   0 ],
  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],
  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],
  [  0,   0,  14,  14,  14,  14,  14,   0,   0,   0,  15 ],
  [  0,   0,   0,   0,   0,   0,   0,   0,   0,  15,  15 ],
  [ 35,  36,  37,   0,   0,   0,   0,   0,  15,  15,  15 ],
  [ 39,  39,  39,  39,  39,  39,  39,  39,  39,  39,  39 ]
];
```



Part2- Some Techniques for Game Development

1- 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 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.

https://spicyyoghurt.com/tutorials/html5-javascript-game-development/collision-detection-physics

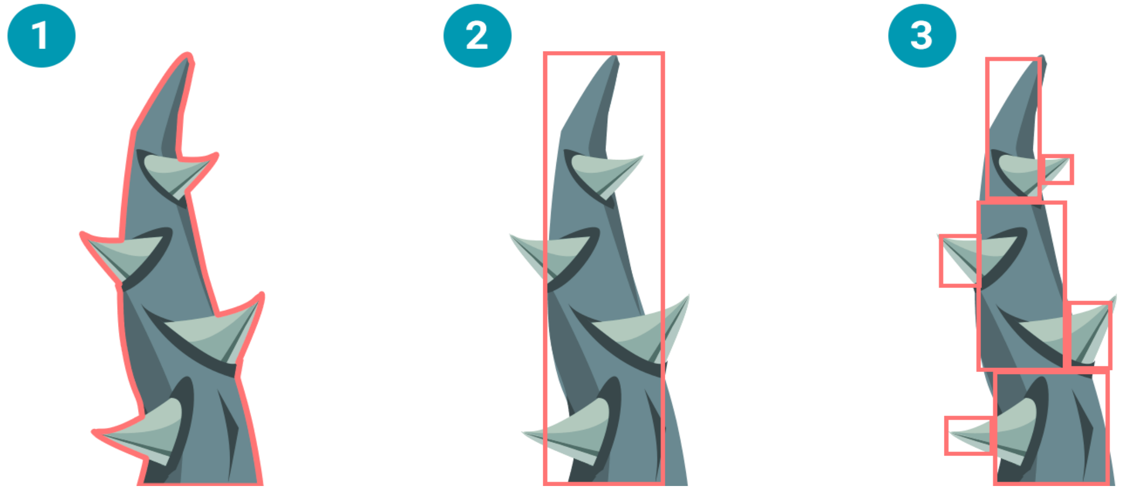
Types of hitboxes

- 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.



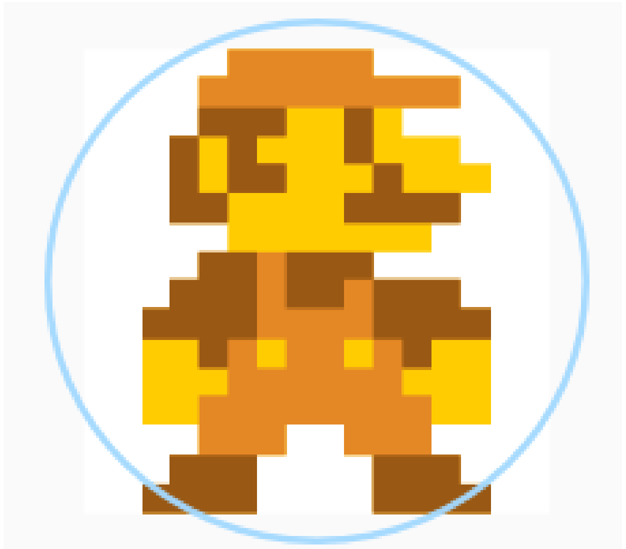
https://spicyyoghurt.com/tutorials/html5-javascript-game-development/collision-detection-physics

Common Hitboxes

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

1-1 Bounding sphere/circle test

- he 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.

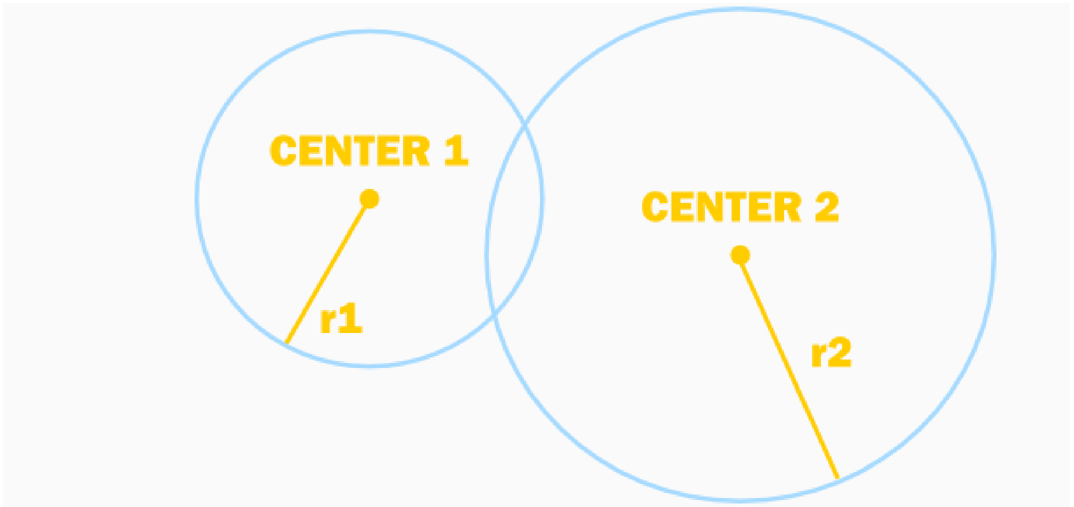


Sprite with circular collision bounds.

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



Circle-circle collision.

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.

Sprite with rectangular collision bounds.

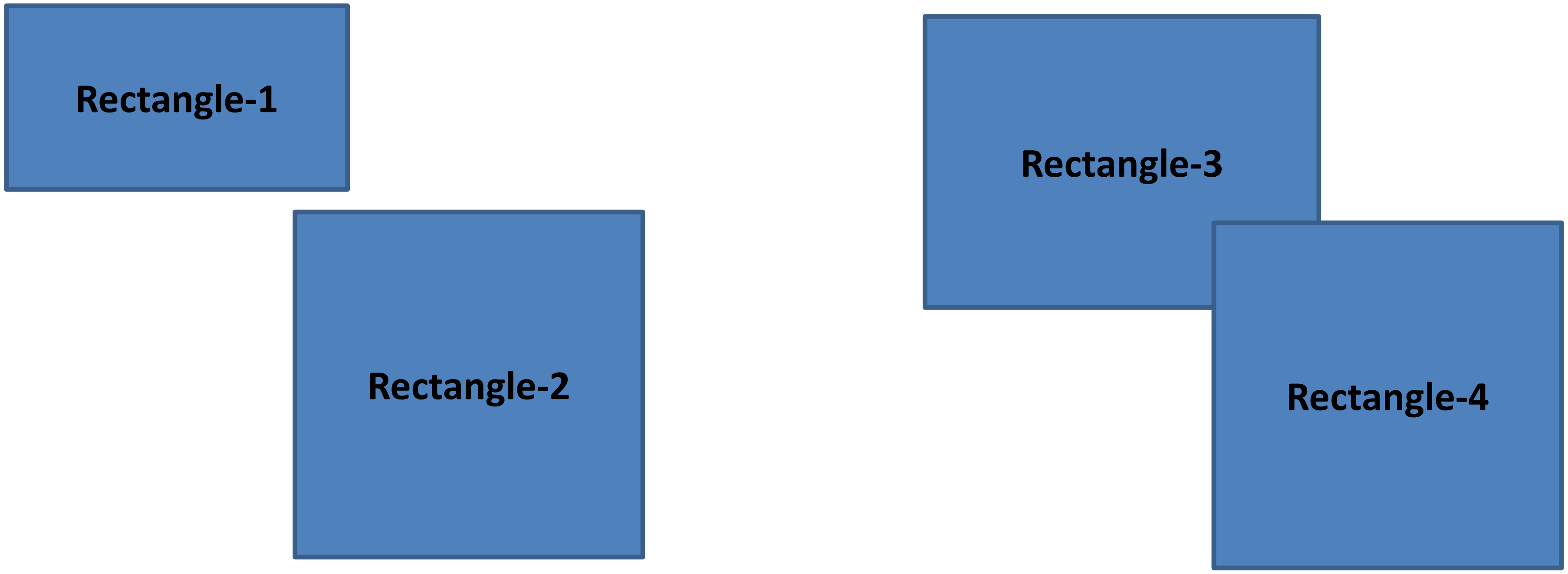
Steps of Box Testing

- 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.

Steps of Box Testing-2

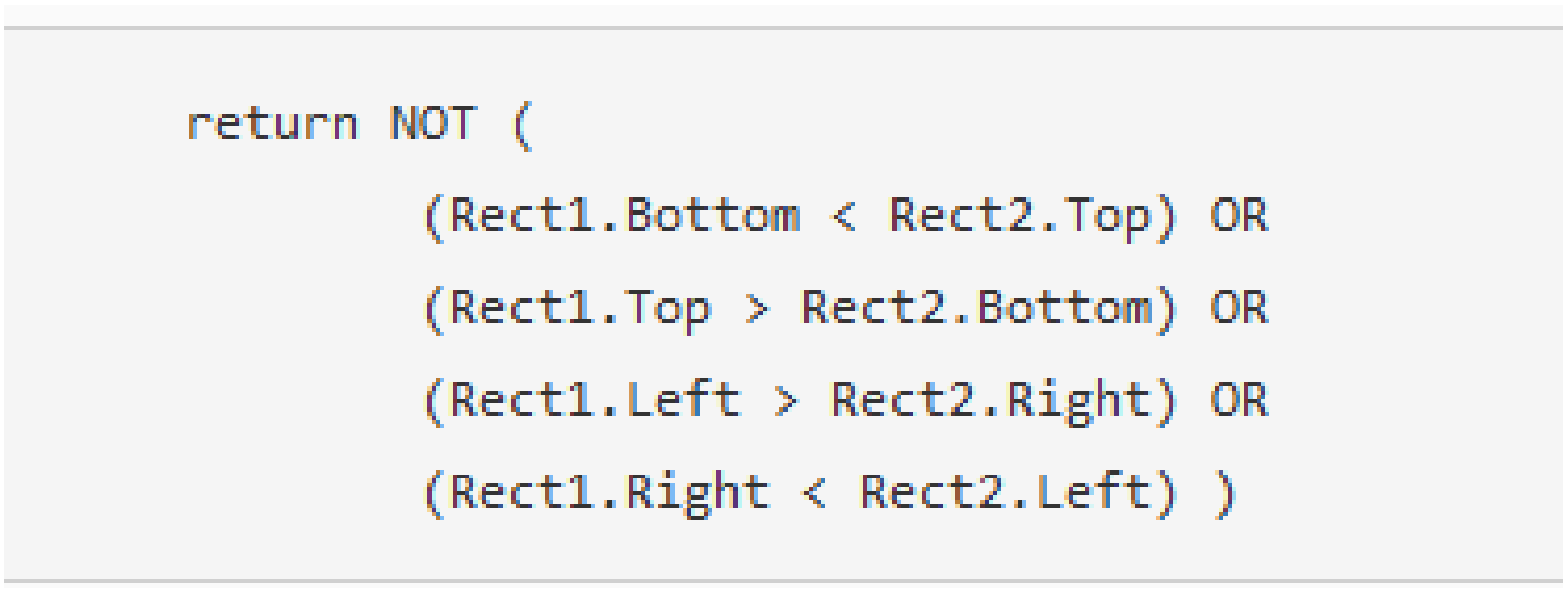
Rectangles will be defined by their left-, top-, bottom- and right-edges. To determine whether two rectangles **do not intersect, one simply has to check for *any* of the following conditions:**

- 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.



Steps of Box Testing-3

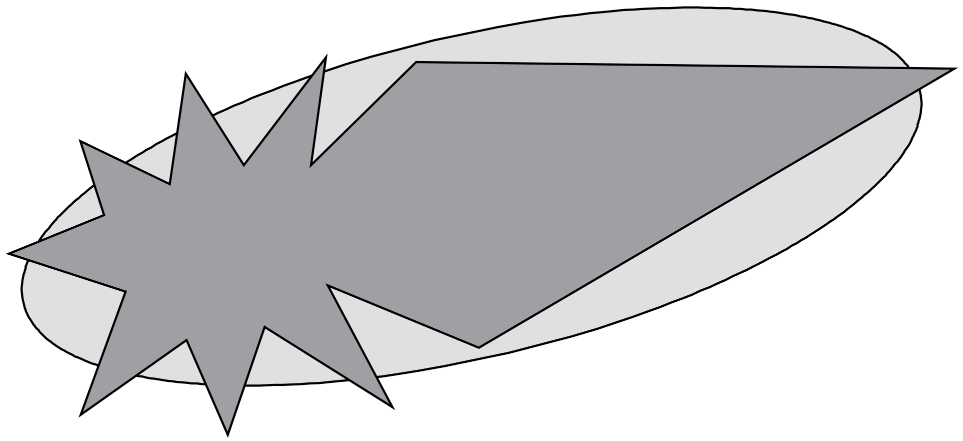
- The above can then be condensed into a single line as follows.



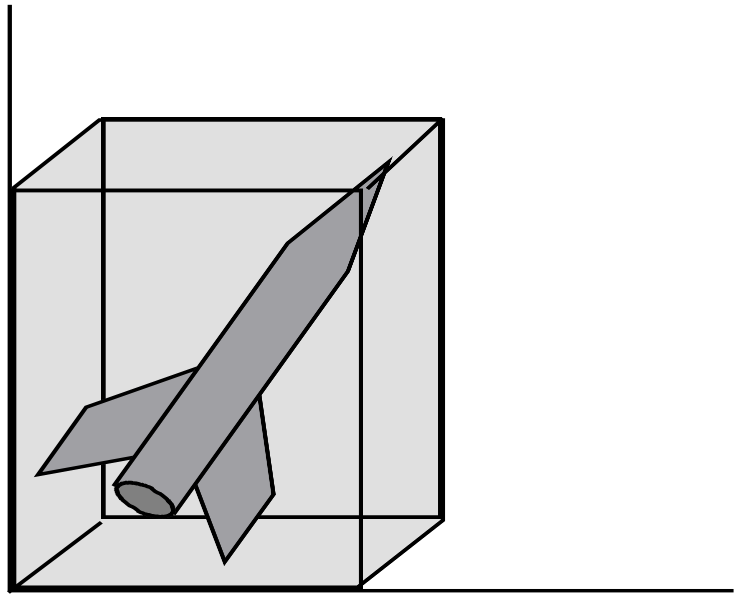
```
return NOT (
        (Rect1.Bottom < Rect2.Top) OR
        (Rect1.Top > Rect2.Bottom) OR
        (Rect1.Left > Rect2.Right) OR
        (Rect1.Right < Rect2.Left) )
```

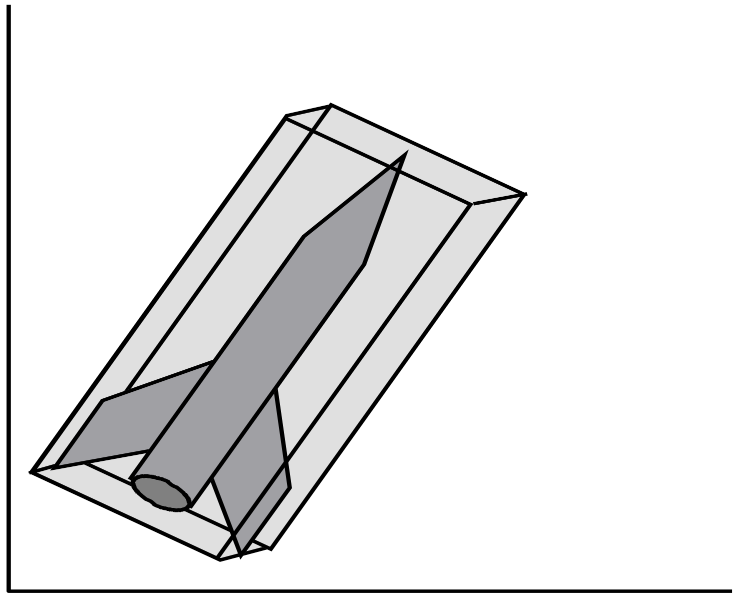
3- Dealing with Complexity: Simplified Geometry

- Approximate complex objects with simpler geometry, like this ellipsoid



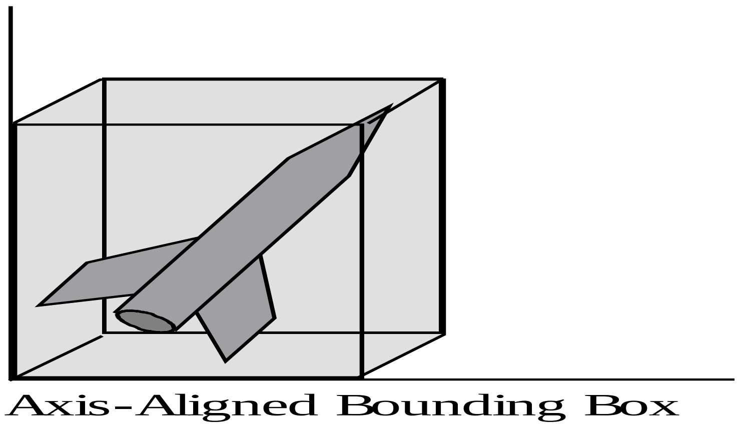
Dealing with Complexity: Box Bounding Volumes

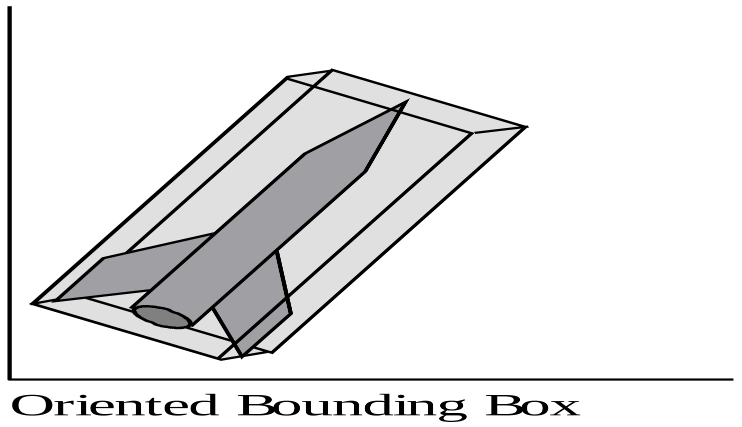

Axis-Aligned Bounding Box


Oriented Bounding Box

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.



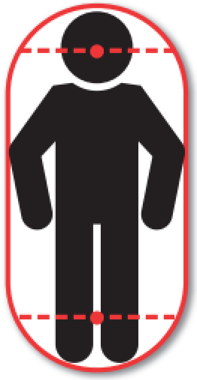


Bounding Capsule

- In 2D, a **capsule** can be thought of as an AABB with two **semicircles** (half circles) attached to the
  top and bottom.

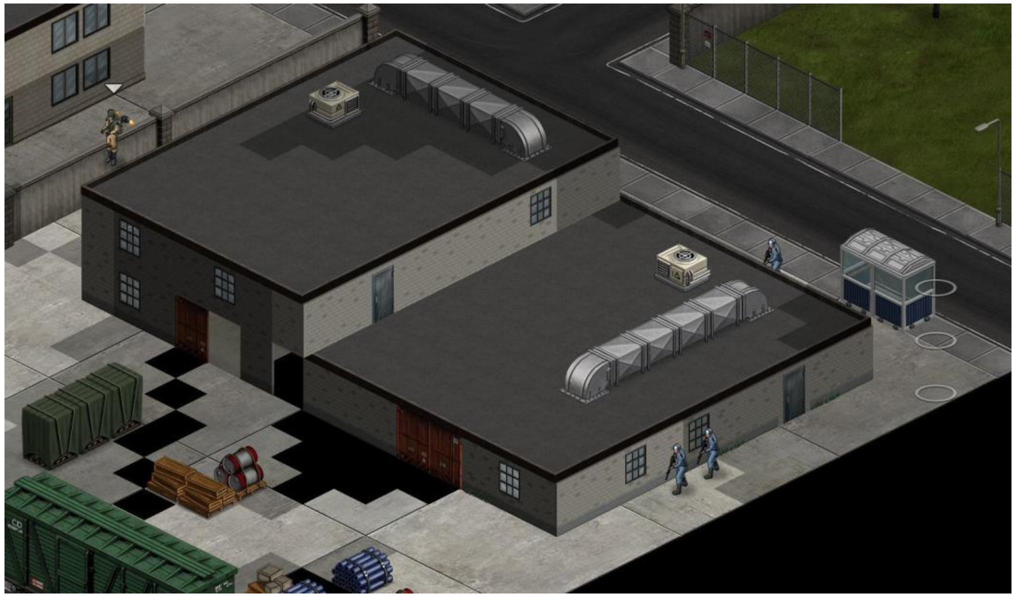
- If we expand the capsule to 3D, it becomes a **cylinder** with two hemispheres attached to the top and bottom.

- Capsules are also a popular form of collision representation for **humanoid characters**
- الشخصيات القريبة من شكل الانسان



2.5D (*Pseudo 3d البعد الثالث الوهمي*)

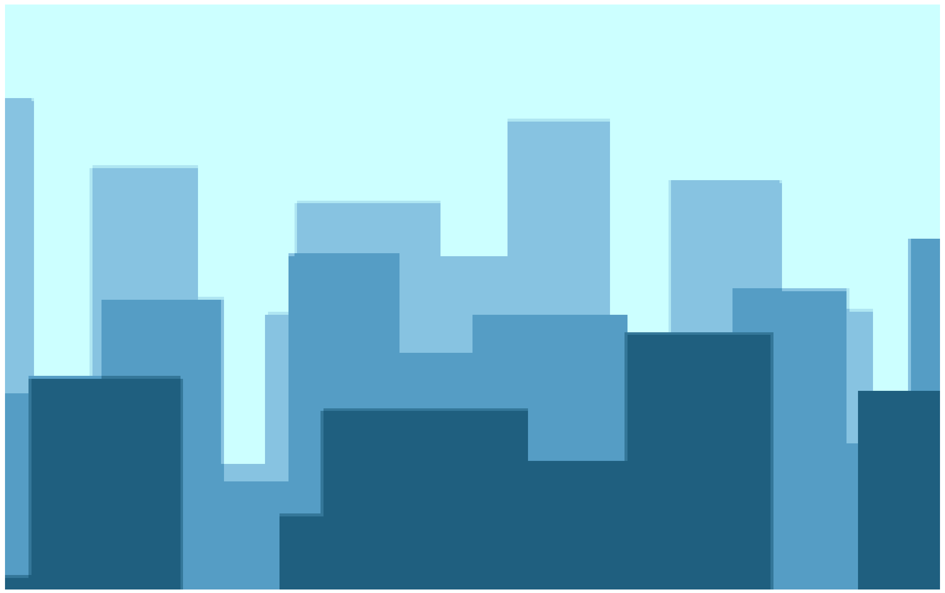
- **2.5D** (**two-and-a-half dimensional**) perspective refers to gameplay or movement in a video game or virtual reality environment that is restricted to a two-dimensional (2D) plane with little to no access to a third dimension in a space that otherwise *appears* to be three-dimensional and is often simulated and rendered in a 3D digital environment.
محاولة محاكاة البعد الثالث في الالعاب ثنائية الابعاد



https://en.wikipedia.org/wiki/2.5D

Parallax scrolling

- Parallaxing refers to when a collection of 2D sprites or layers of sprites are made to move independently of each other and/or the background to create a sense of added depth
- This depth cue is created by relative motion of layers.

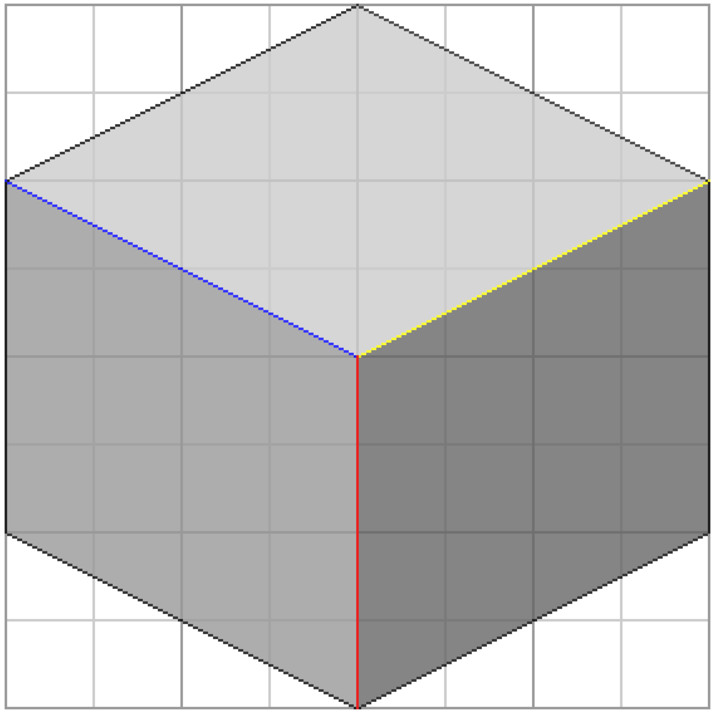


Graphical Projection

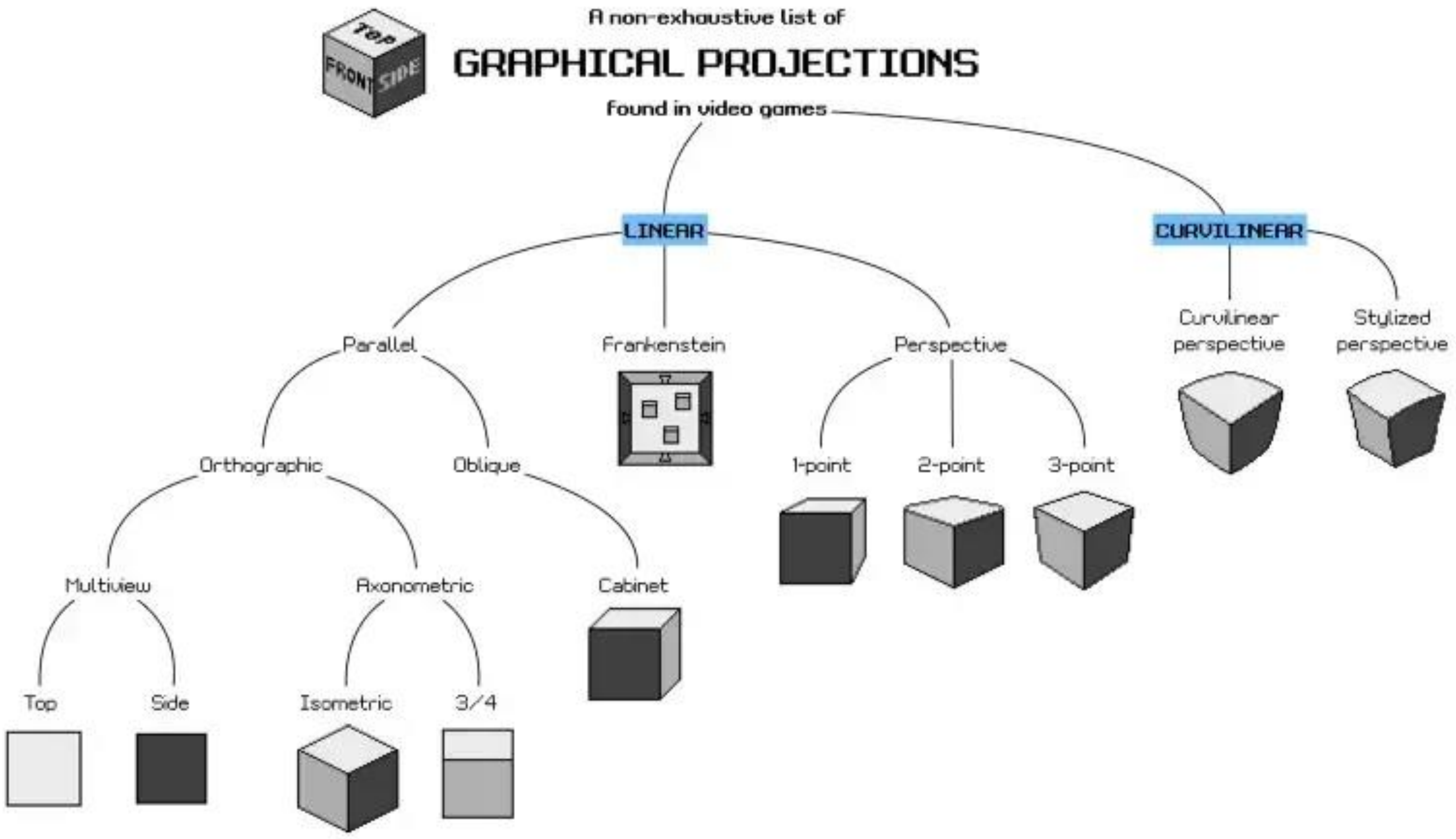
- how to draw three-dimensional objects on a two-dimensional surface.
- Used in 2.5D games to simulate 3D enviroment

Isometric Projection

- The word ‘isometric’ comes from Greek origin, meaning 'equal measure’.
- In an Isometric view, the scale of all three dimensions — x, y and z — is identical, and the angle between each axis is 120 degrees. A cube projected isometrically will retain its proportions, meaning that its width, height and length will be of identical size. Also, all three faces of the cube will contain the same surface area.



More Graphical Projections
الرسمة للاطلاع فقط



https://medium.com/retronator-magazine/game-developers-guide-to-graphical-projections-with-video-game-examples-part-1-introduction-aa3d051c137d

Unity 2D tutorials

Unity 2D

https://www.youtube.com/playlist?list=PLGmYIROty-5YhzMWqgAIXlPIRU2_tcz0l

Unity 3D

https://www.youtube.com/playlist?list=PLGmYIROty-5bpzKQNK3mRMi4pmh_LinV4
and
https://www.youtube.com/playlist?list=PLhPa7T8XPi0I7XkxrTEYyyU_YR1sePyTE

References

- https://etu-cnm.uqat.ca/20181-art1211/racj22/TP3/2-5d.html
- https://www.significant-bits.com/a-laymans-guide-to-projection-in-videogames/
- https://en.wikipedia.org/wiki/2.5D
- https://www.vectornator.io/blog/isometric-illustration/