Mansoura University
Faculty of Computers and Information
First Semester- 2020-2021

COMPUTER GRAPHICS

Grade: 2ND YEAR (GENERAL –BIO)

Course Instructors :

Assoc. Prof. Dr-Haitham El-Ghareeb
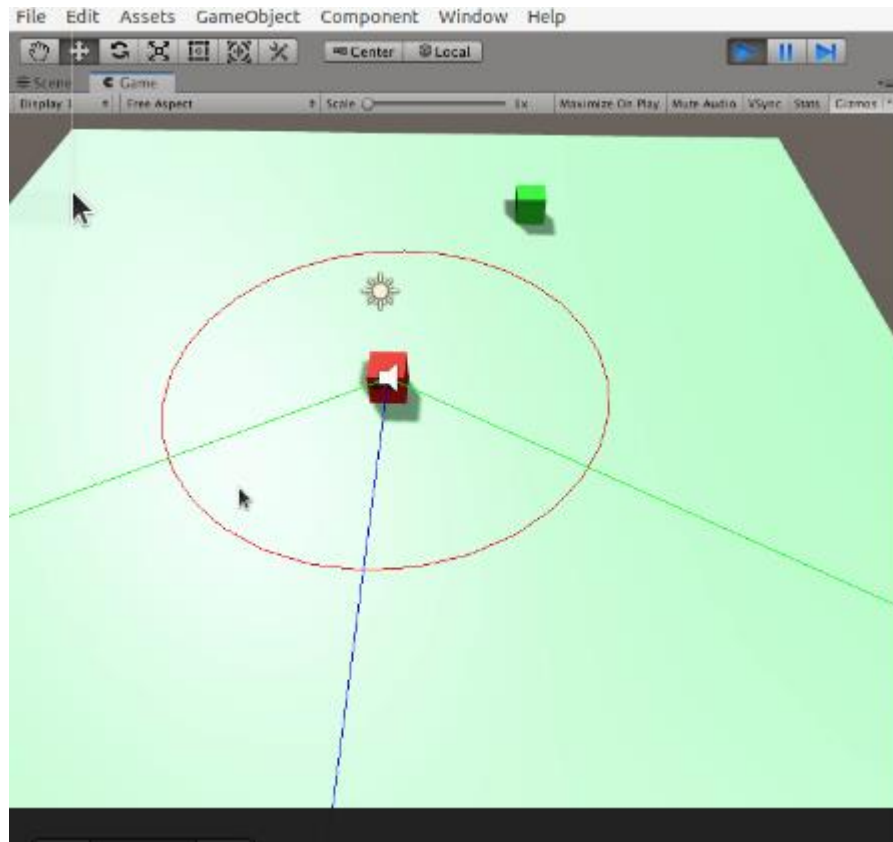
Dr-Ibrahim El-Hassanoni

waleed mohamed

Chapter 06- Math and Animation for game Developers
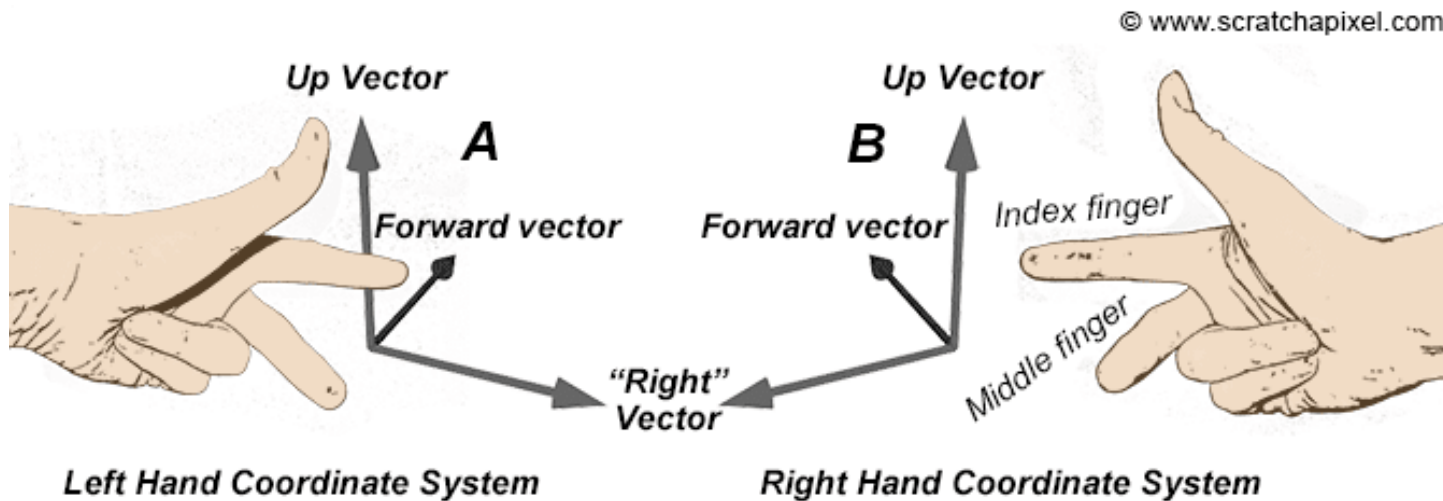
Lecture Demo Application in unity

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

Orientation of vectors

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.

© www.scratchapixel.com

Left Hand Coordinate System — A — Up Vector, Forward vector, "Right" Vector

Right Hand Coordinate System — B — Up Vector, Forward vector, Index finger, Middle finger
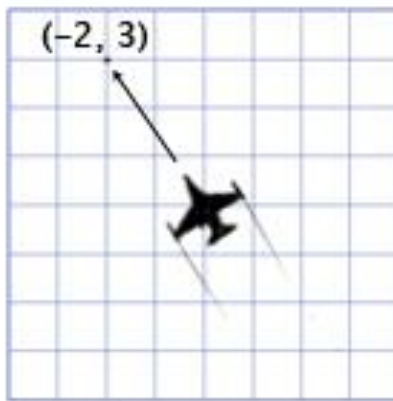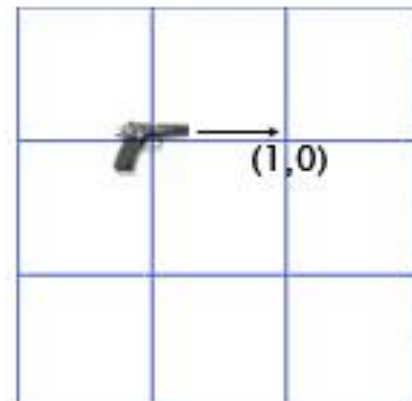
Vector

- 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.


(2,1)

Position

(−2, 3)

Velocity

(1,0)
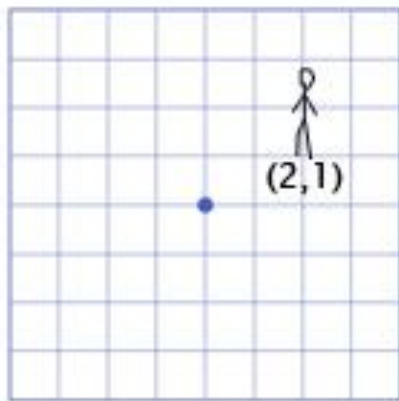
Direction

- 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**

- **التغير في سرعة الحركة(العجلة)**.

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.
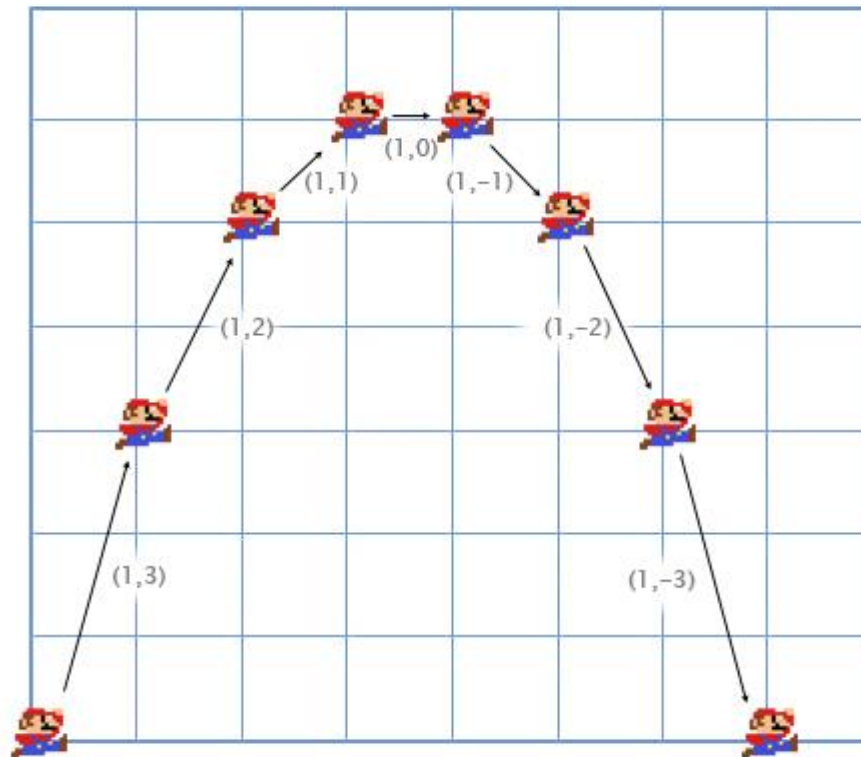
Vector subtraction

- 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).
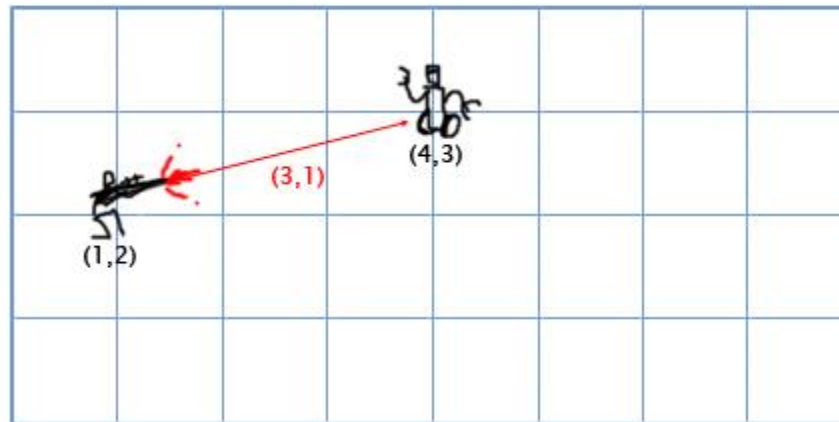
Scalar-vector multiplication

- 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: $x^2 + y^2 = h^2$.**

- That is, the length of a vector H with components (x,y) is **$sqrt(x^2+y^2)$**

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 $sqrt(x^2+y^2+z^2)$.
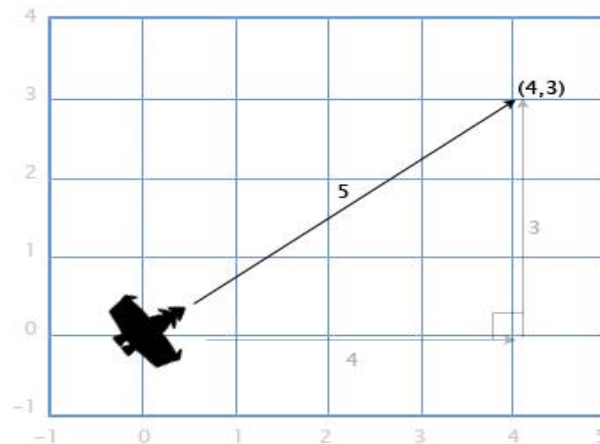
Distance

- 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
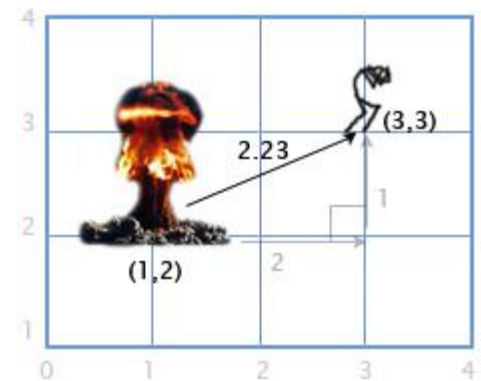
Normalization

- 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 $(\frac{3}{5}, \frac{4}{5})$. Now we can use the pythagorean theorem to prove that it has length 1: $(\frac{3}{5})^2 + (\frac{4}{5})^2 = \frac{9}{25} + \frac{16}{25} = \frac{25}{25} = 1$
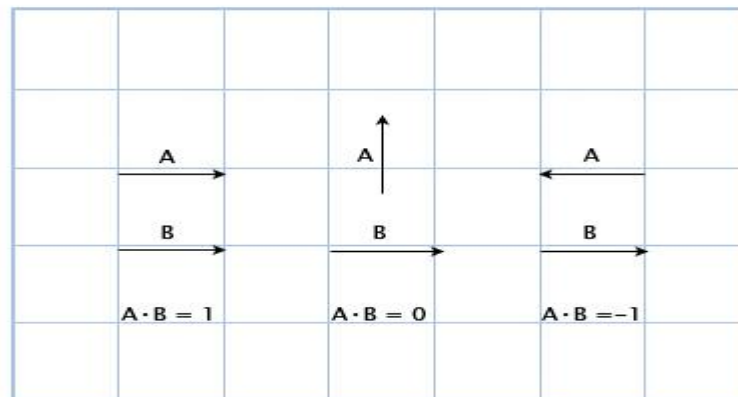
Dot product

- To get the dot product of two vectors, we multiply the components, and then add them together.

- $(a_1, a_2) \cdot (b_1, b_2) = a_1 b_1 + a_2 b_2$

- For example, $(3,2) \cdot (1,4) = 3*1 + 2*4 = 11$.

- *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:

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

- Since 1 is **positive**, the hero is in the guard's field of view!
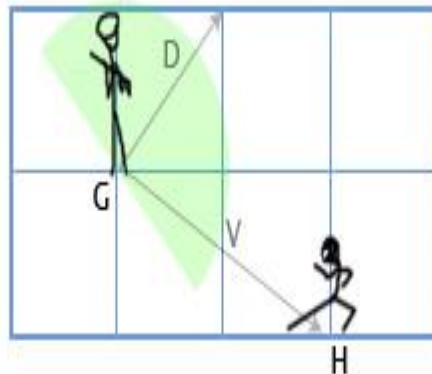
Using Dot Product to get angle

- 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:

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

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

$$\theta = acos(AB)$$

Example 3: Using Dot Product to Get Angle

- 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(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!
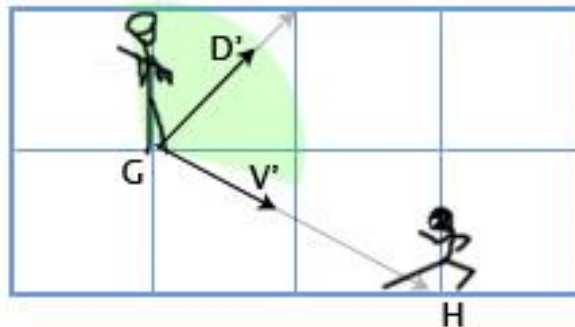
Cross Product (special 2d case)

- 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** مع عقارب الساعة, just **flip** بدلthe 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 of right facing vector** to get: (-1,2).
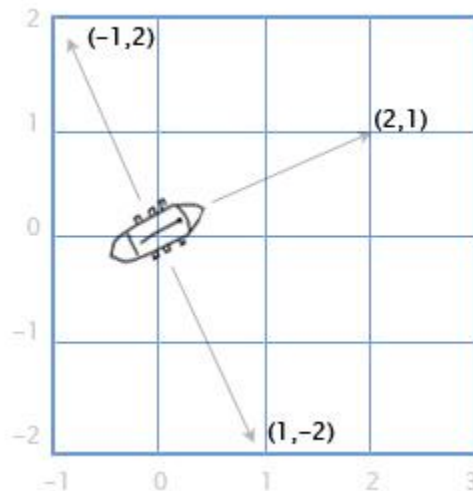
Cross Product-2

- 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.
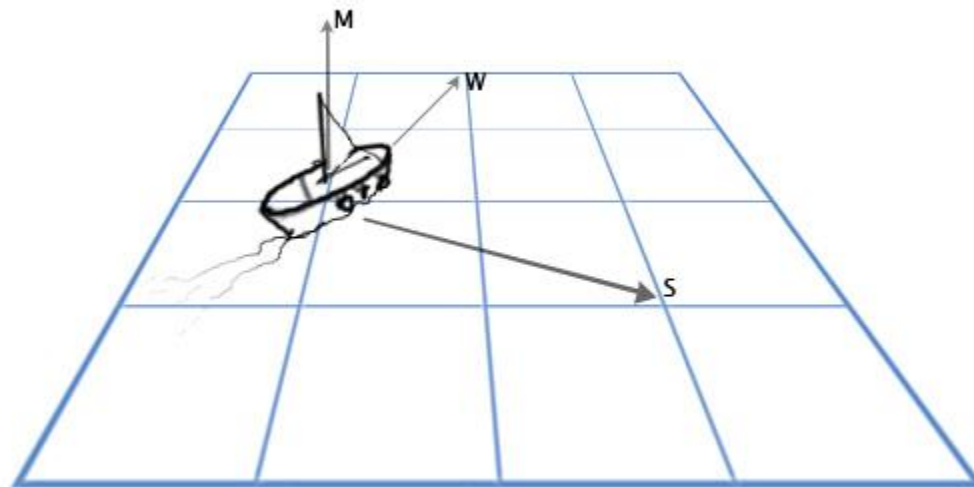
Cross Product-3 (Continue example)

- 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:

$S = MxW = (0,1,0)x(1,0,2) = ([1*2-0*0], [0*1-0*2], [0*0-1*1]) = \textbf{(2,0,-1)}$
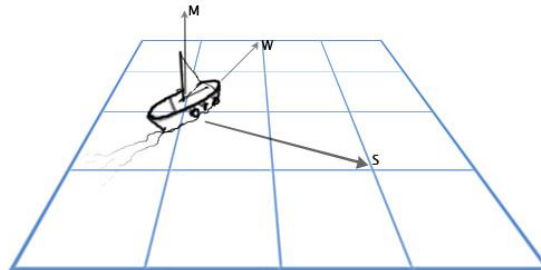
LINEAR INTERPOLATION (LERP)

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:

$$L = \text{LERP}(A, B, \beta) = (1 - \beta)A + \beta B$$
$$= [(1 - \beta)A_x + \beta B_x, \quad (1 - \beta)A_y + \beta B_y, \quad (1 - \beta)A_z + \beta B_z].$$

- 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
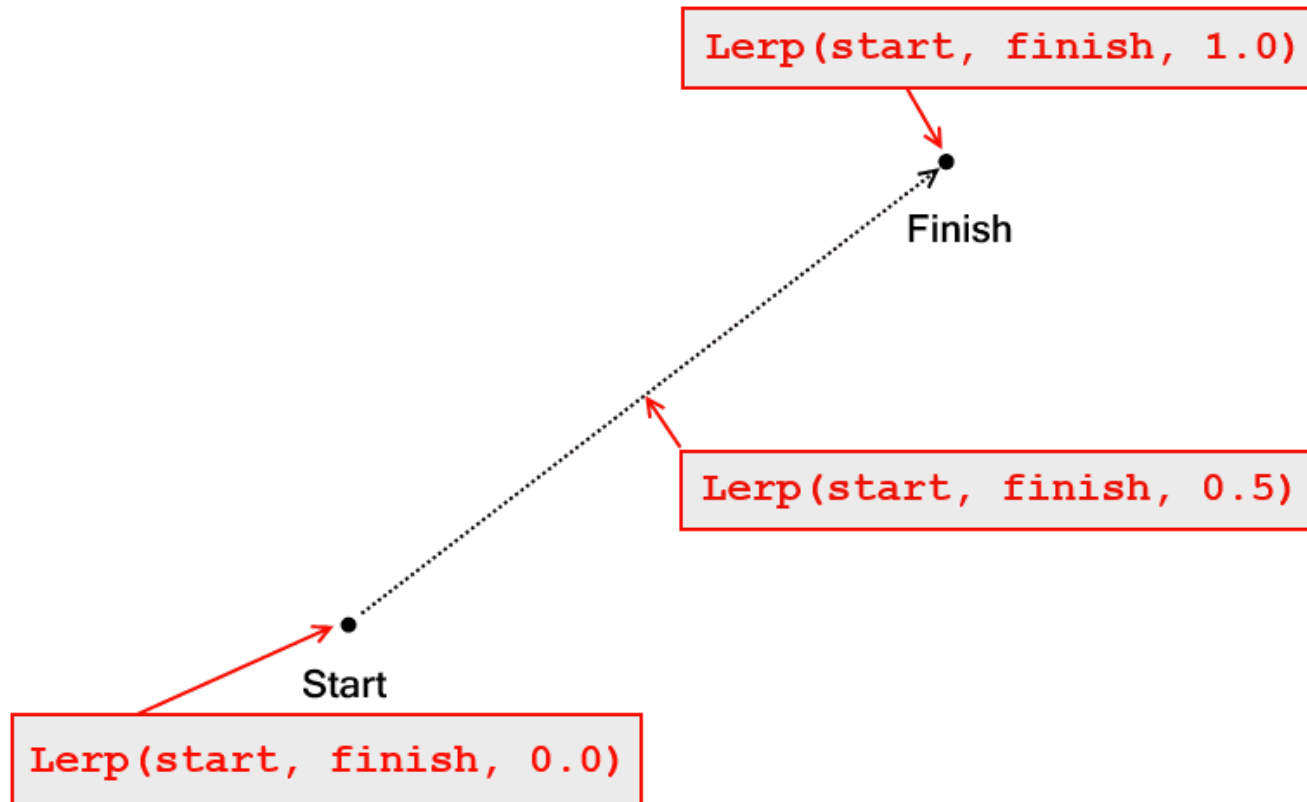
Model space (world and view coordinates)

- 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:
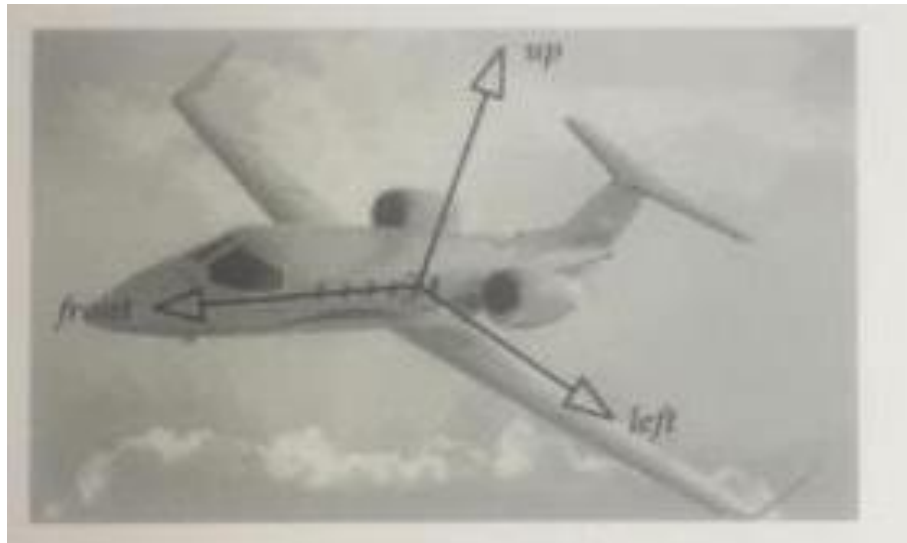
World coordinate

- 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.
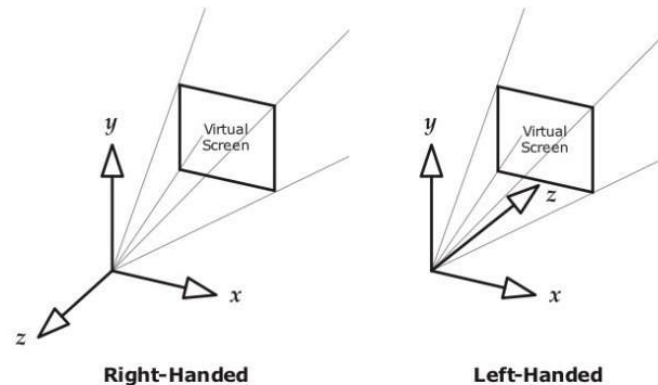

Right-Handed　　　　**Left-Handed**

Figure 4.19. Left- and right-handed examples of view space, also known as camera space.

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.
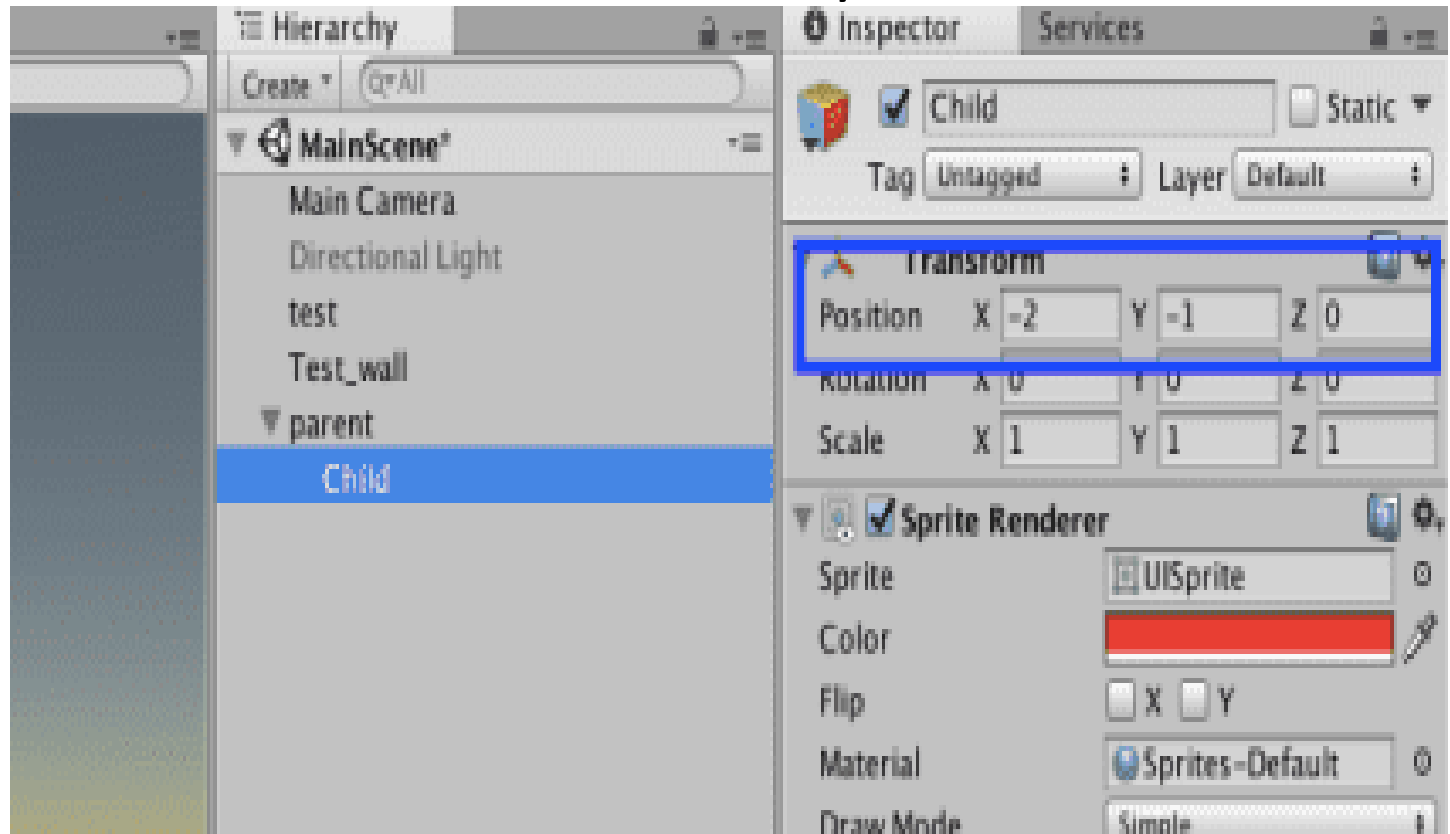
Raycasting

- 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.

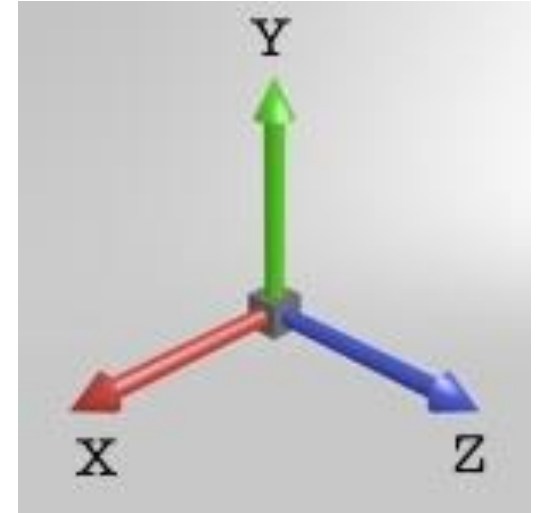- Let's take this example from the Unity Tutorials:

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

Ray Direction

!
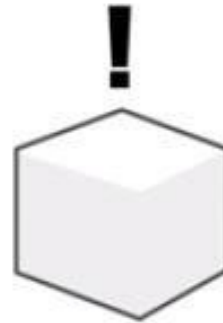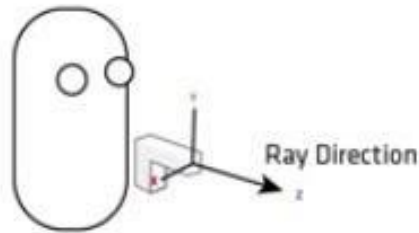
World Coordinates

Cartesian & Spherical coordinates

- A point in the **cartesian** coordinate system can be defined using a vector (x,y,z) from the origin to the point

Cartesian & Spherical coordinates

- Spherical coordinate system :
    - 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**.
    مثل خطوط الطول ودوائر العرض

- To define a point in this system the following is needed:

1. *Radius*: the distance from the origin to the point

2. *Elevation angle*: the angle between the plane (with zenith axis as normal) and the line from the origin to the point

3. *Polar angle*: the rotation around the zenith axis
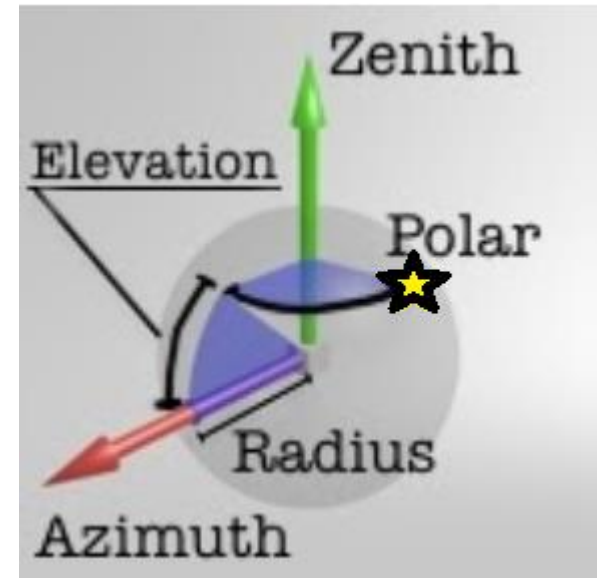
(the star represents point with the measured angle)

Applications of Spherical coordinates

- 1- 3d map visualization

- 2-Chase camera:
  - 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.

Polar, Spherical and Geographic Coordinates

• للاطلاع فقط
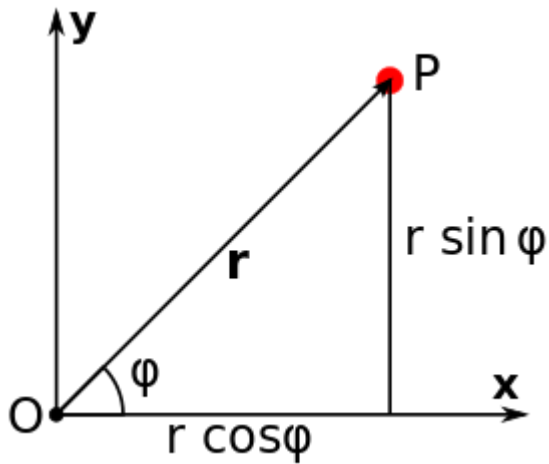
Then the conversion from a cartesian vector (x, y) of a position **P** to polar coordinates (radius, angle) is:
radius = sqrt(x^2 + y^2)
angle = atan2(y, x)
and the way around:
x = radius * cos(angle)
y = radius * sin(angle)

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: 1- 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**.
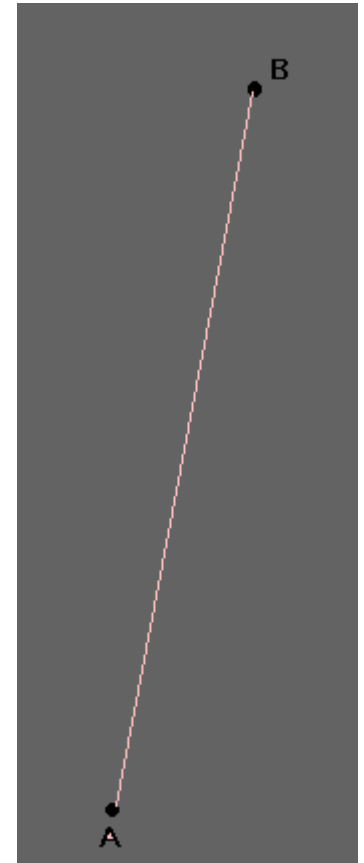
2-Quadratic (curves)

- 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**.
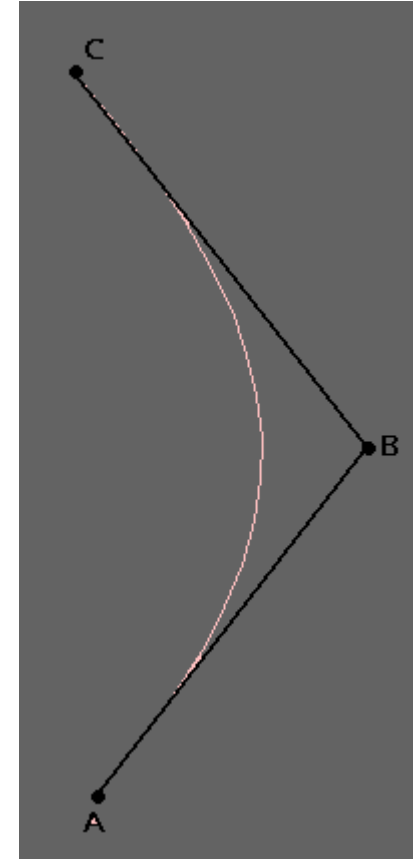
3-Cubic

- 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**
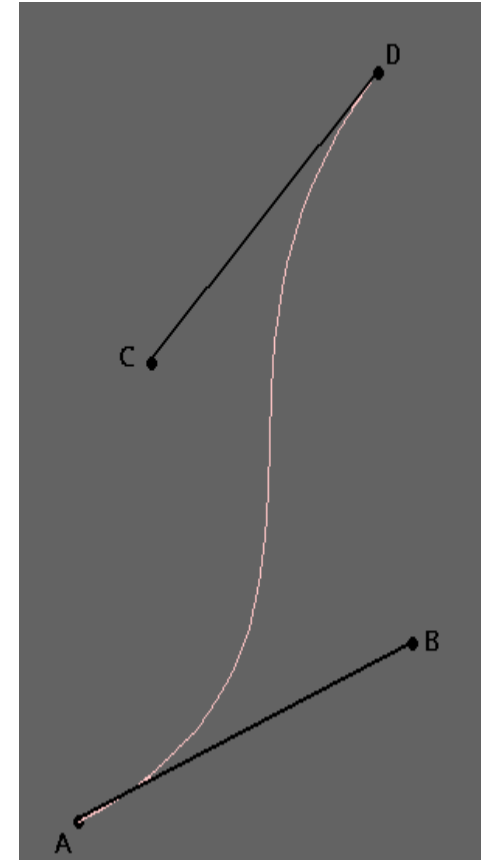
Basics of Game programming

- 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
```

Game Loop with Delta Time

```
while game is running
    realDeltaTime = time since last frame
    gameDeltaTime = realDeltaTime * gameTimeFactor

    // Process inputs
    ...
    update game world with gameDeltaTime

    // Render outputs
    ...
loop
```

- Note: gameTimeFactor سرعة الحركة 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 then 1 for more speed

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>**
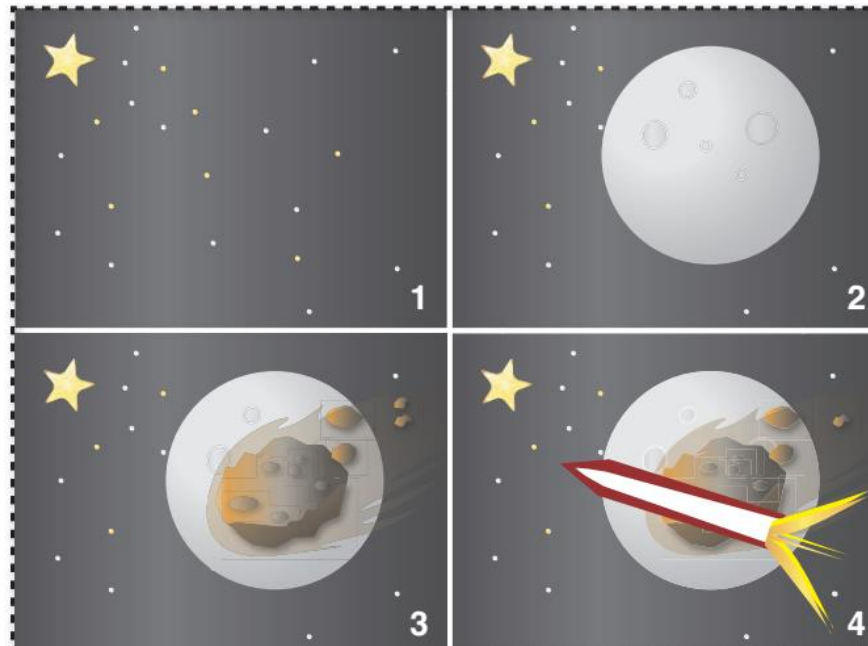
Animating Sprites

- 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
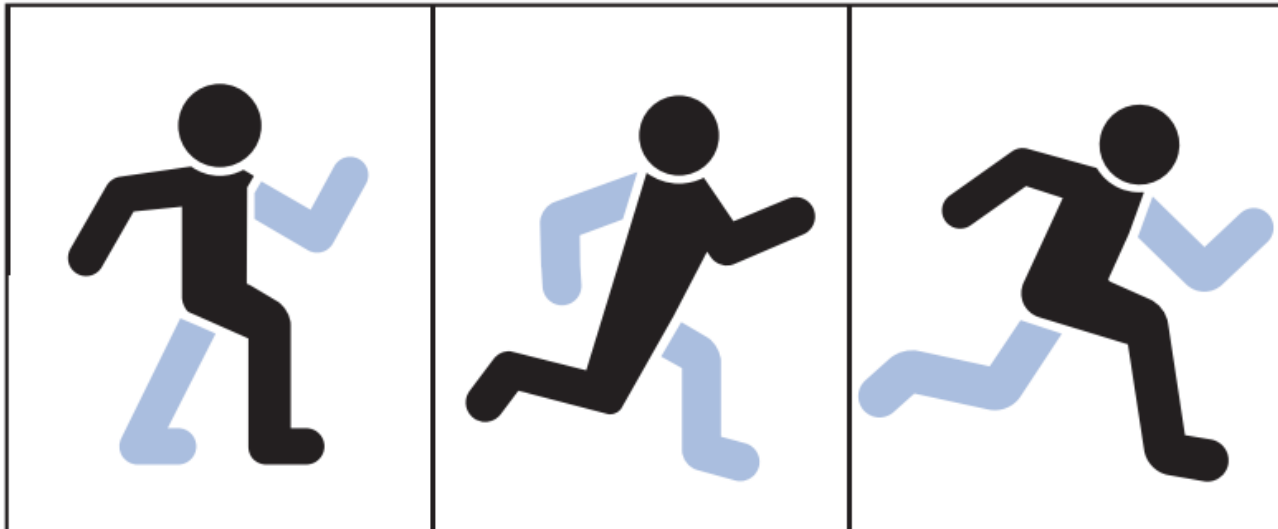
Sprite Sheets

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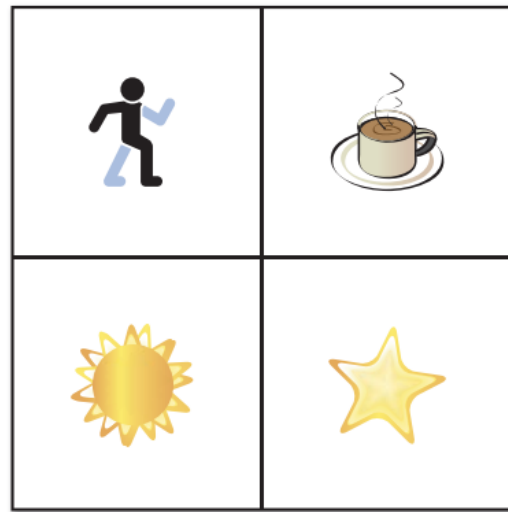
Individual sprites (a), and those sprites packed in a sprite sheet (b)

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…
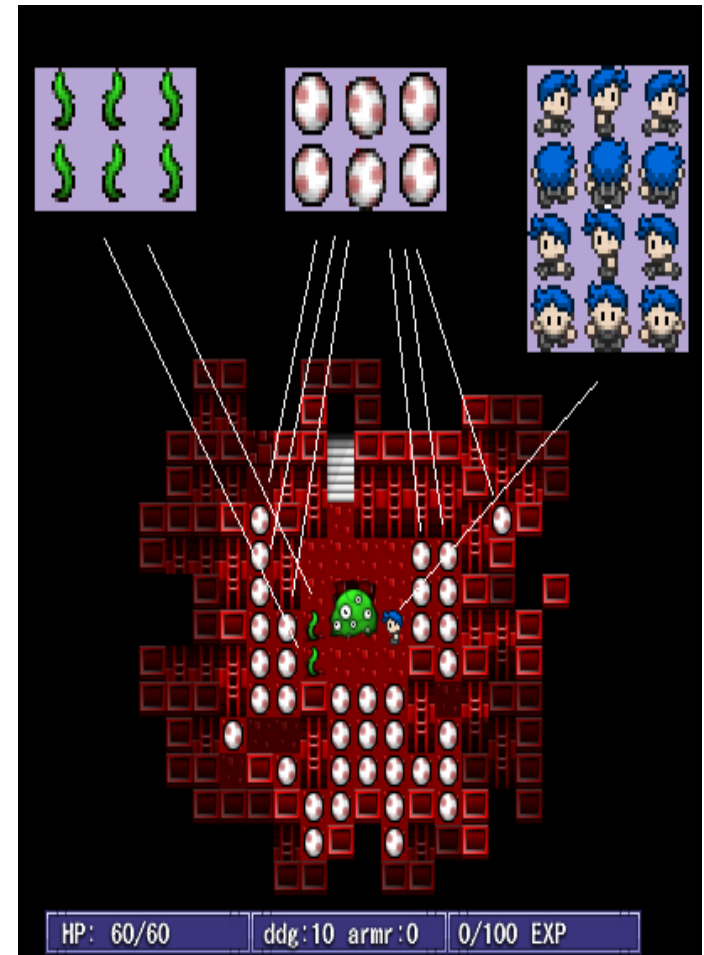
Relative Paths

- 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?
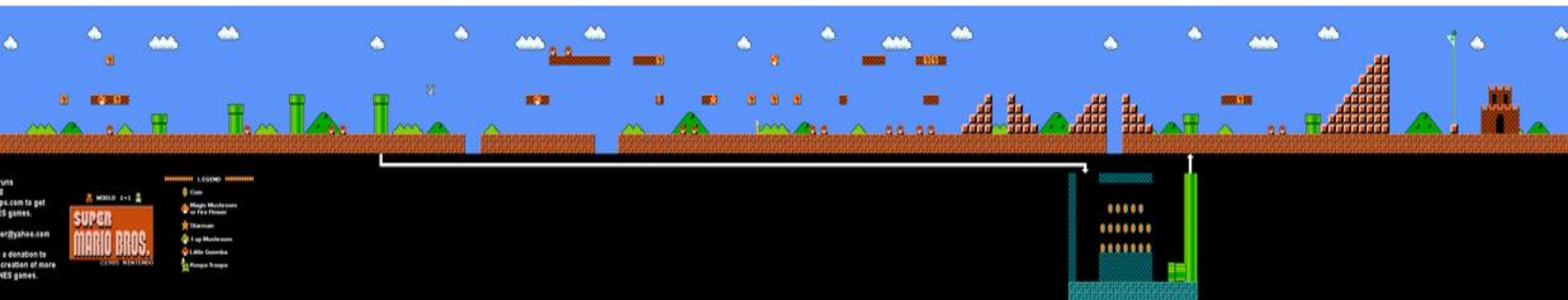
Example-continue

- The tilemap approach defines a set of modular, regularly-sized *tiles* مربعات متساوية الأبعادthat we can use to build our levels. That way, we only need one image, a *tileset*:
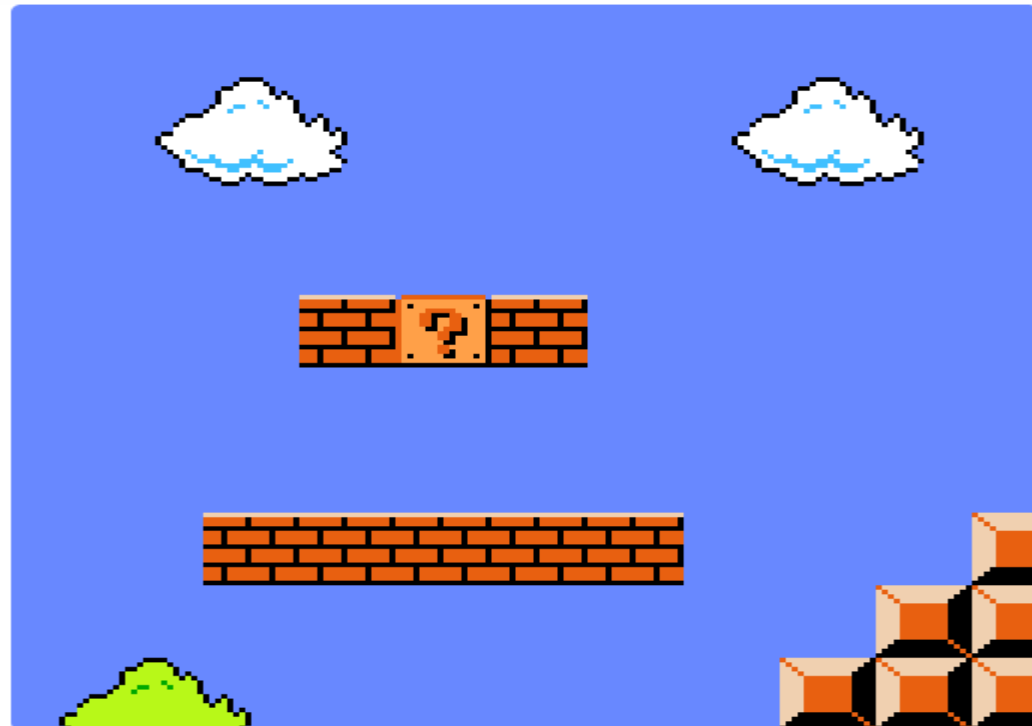
Example-continue 2

- 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.
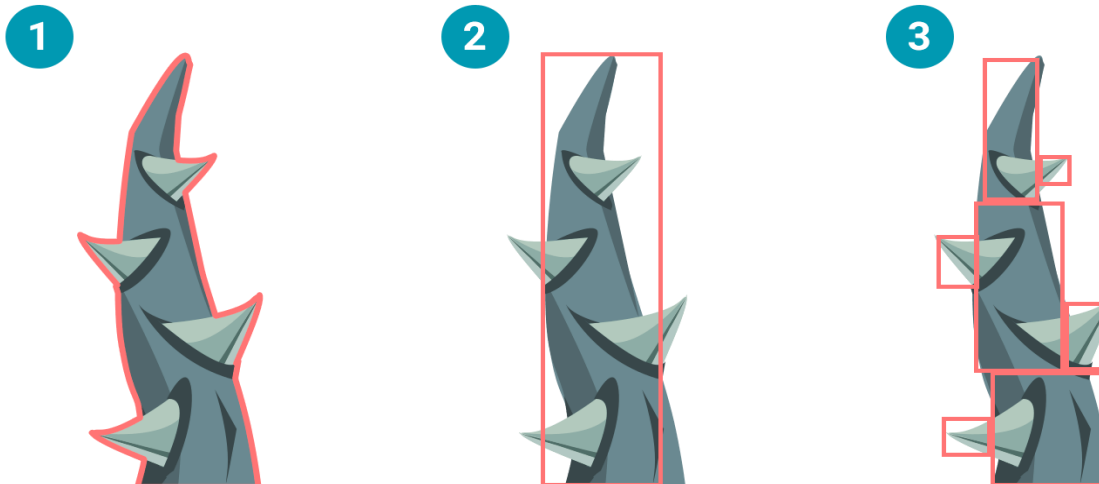
Common Hitboxes

- In 2d there are some common geometric shapes for hitboxes:

  - Circles
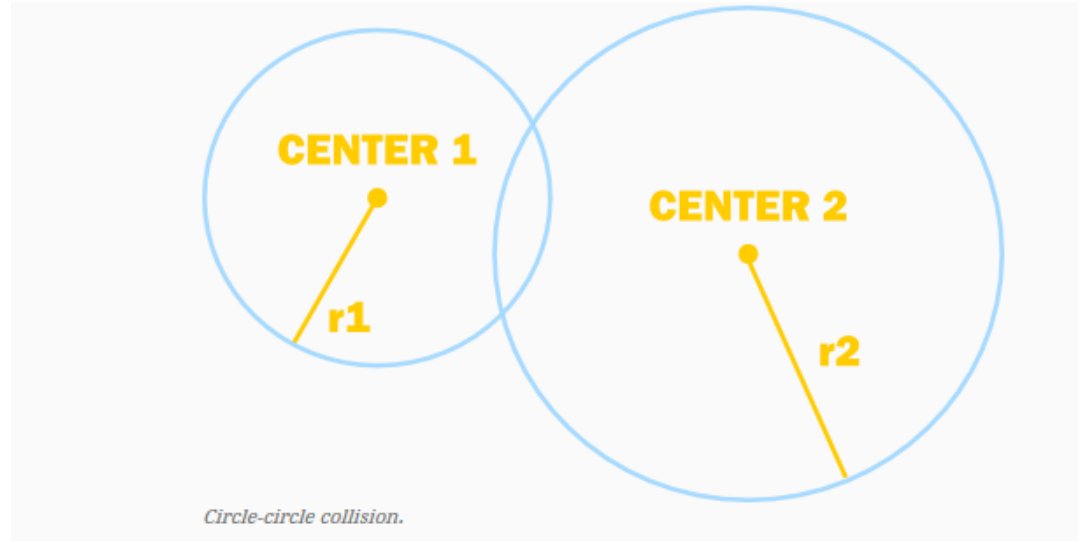
  - Box (rectangle)

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.
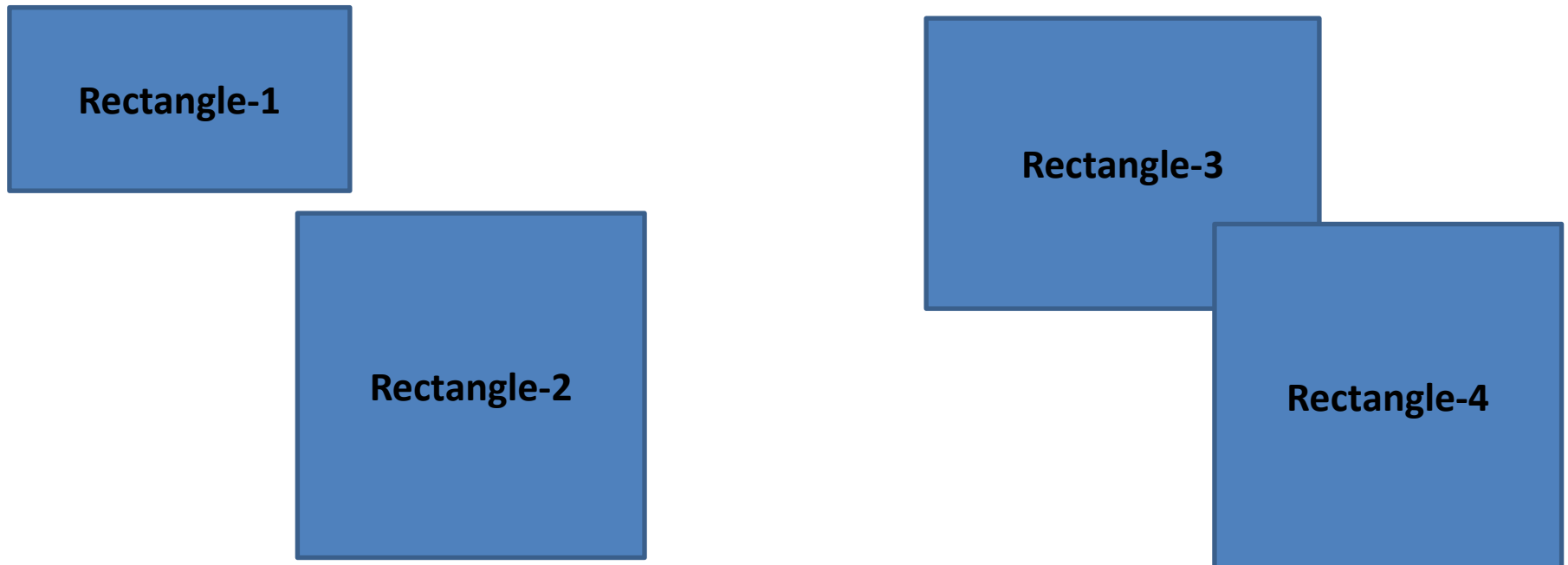
Steps of Box Testing-3

- 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

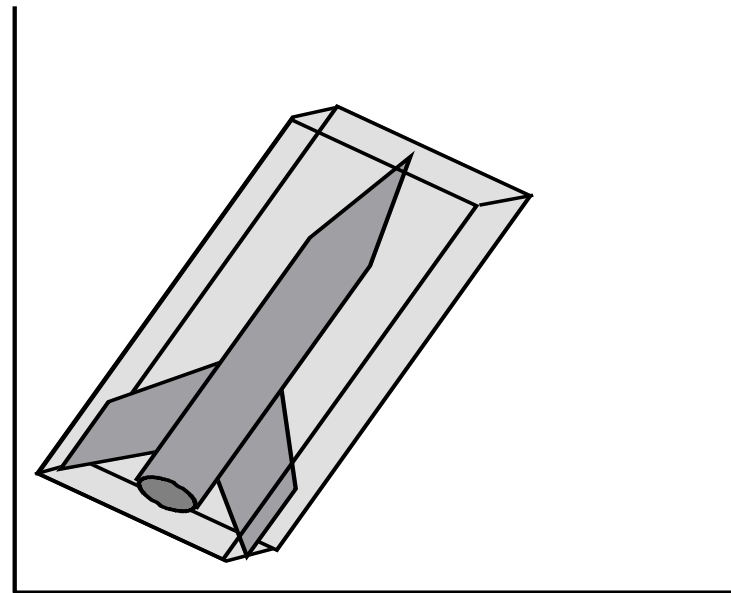- Approximate complex objects with simpler geometry, like this ellipsoid
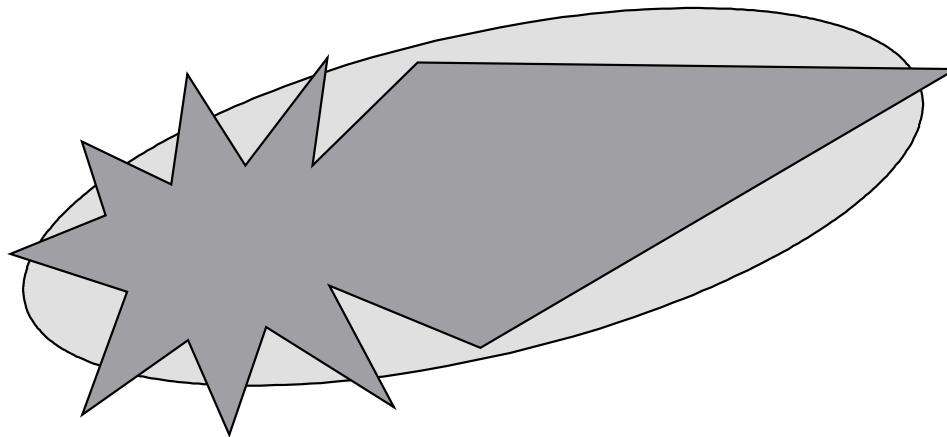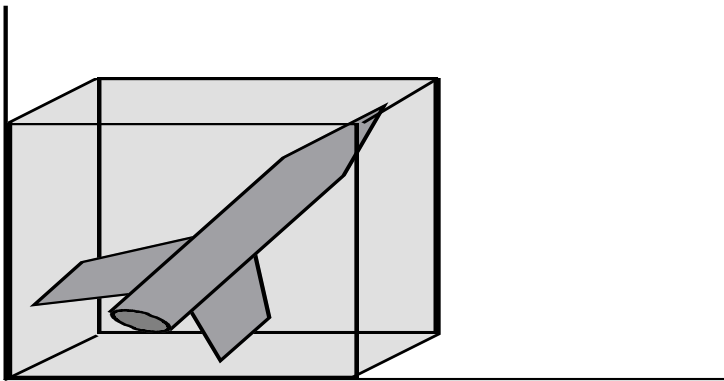
Dealing with Complexity:
Box Bounding Volumes
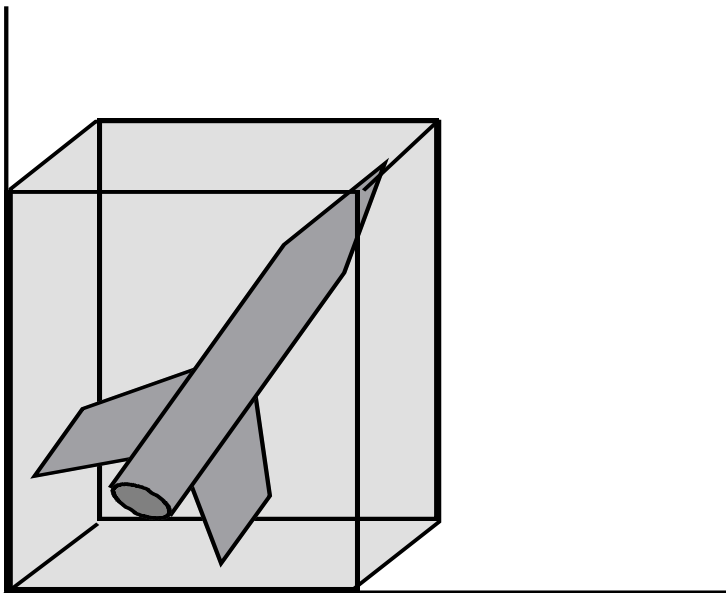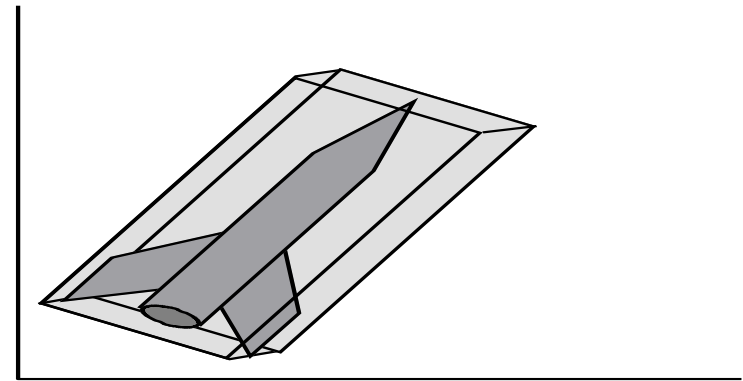

Axis-Aligned Bounding Box

Oriented Bounding Box

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

- Box edges aligned to **world axes**

- **Recalculate bounds** when object changes orientation

- Collision checks are simpler

- The other type is "**Oriented Bounding Box**", which rotates with object rotation.

Axis–Aligned Bounding Box

Oriented Bounding Box

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

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

Animation basics

Animation Vocabulary

1) Timeline

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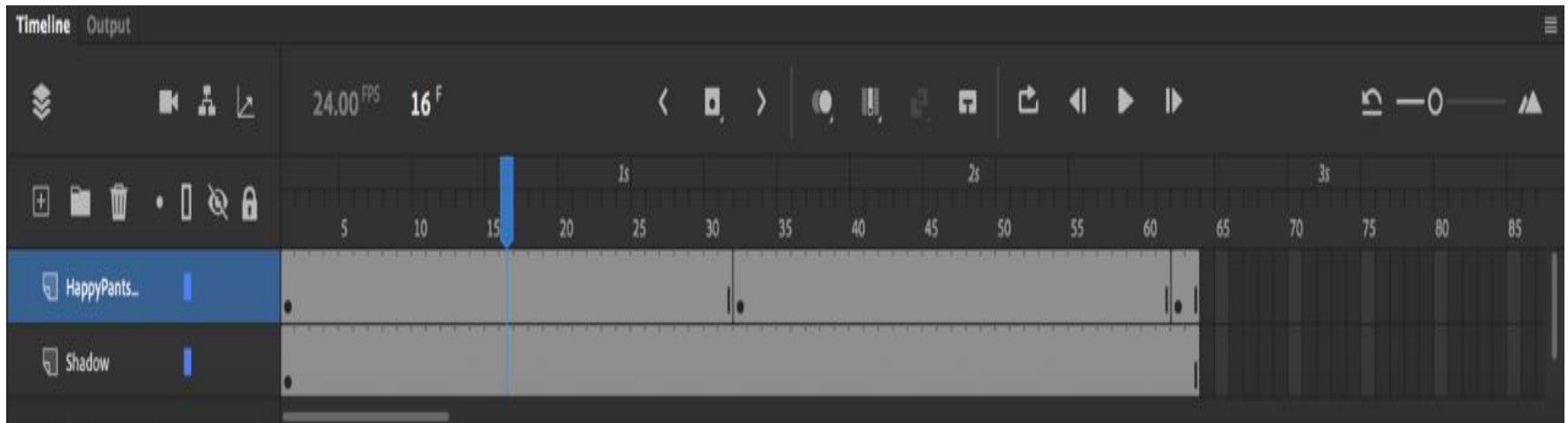
Animation Vocabulary- continue

2) 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).

2) 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.

Animation Vocabulary- continue

- 4) 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.

- 5) 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.

Keyframes

KEY KEY KEY

Breakdowns

PASSING POSITION
OR
BREAKDOWN BDWN BDWN BDWN BDWN BDWN BDWN BDWN BDWN

In-betweens

SEPARATE RUN ON HEAD

SEPARATE
RUN ON
ARMS

SEPARATE
RUN ON
WRITING

SEPARATE RUN ON FEET

Animation Vocabulary- continue

6)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.
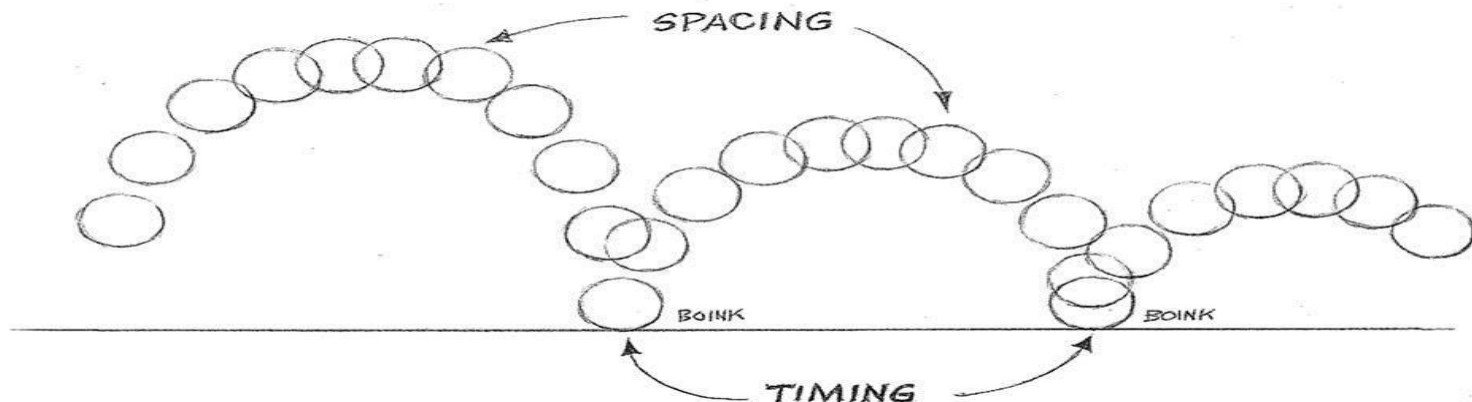
Common Animation Types

1- Hand-drawn Animation

•In this type of animation, a series of drawings is photographed

•Each drawing makes up one frame of the film

•The position of the character or object changes very slightly from frame to frame

•This technique is known as **cell animation** because the first animations were drawn on **thin plastic sheets called celluloid**

2-Stop-Motion Animation

- In Stop-Motion Animation, models are photographed on a set one frame at a time.

- The frames are played in sequence to create movement

- Early stop-motion models were made of modeling **clay**

Stop motion

• Stop-motion animation is used to describe animation created by physically manipulating real-world objects and photographing them one frame of film at a time to create the illusion of movement.

• There are many different types of stop-motion animation, usually named after the medium used to create the animation. Computer software is widely available to create this type of animation.

2-1. Puppetoon
عرائس متحركة

•In puppetoon animation the puppets are rigid and static pieces;

•each is typically used in **a single frame** and then switched with a separate, near-duplicate puppet for the next frame.

2-2 Clay animation or claymation

- A subtype of **stop motion** animation.
- Each animated piece, either character or background, is "**deformable**"—
- قابلة لتغيير الشكل بالضغط عليها

3-Computer Animation

•Computer Animation software can be used to create an animation scene, cartoon, a video game, or even a full-length movie

•Computer Animation reduces the time needed to produce an animated film

•Software can be two-dimensional (2D) or three-dimensional (3D)

4- Motion Capture Animation

• Motion Capture Animation is a 3D representation of a live performance

• It can be done using 3 different technologies:

1. Magnetic
2. Optical
3. Electro-Mechanical

4-1. Magnetic Motion Capture

- Magnetic motion capturing uses **<u>sensors</u>** placed on an actor's body that measure magnetic fields created by a transmitter
- The files are then represented in 3D by a computer

4-2. Optical Motion Capture

- Optical motion capturing uses video cameras to track the motion of light-emitting diodes or reflective markers placed on the **joints** مفاصلof the actor's body

4-3.Electro-mechanical Capture

•Electro-mechanical devices resemble an **exoskeleton**

• بدلة تماثل الهيكل الخارجي للشخصية that is worn by the actor being recorded. The exoskeleton's movements are recorded without any additional sensors
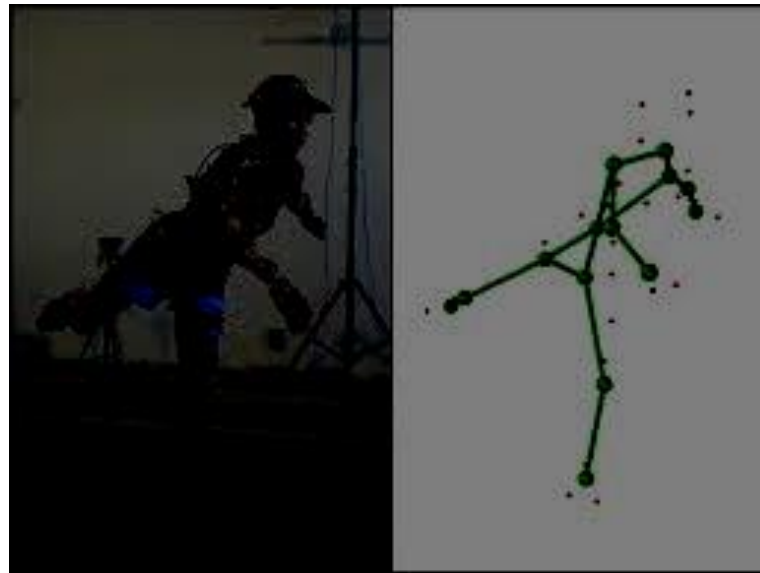
Cutout animation

•is a type of stop-motion animation produced by moving two-dimensional pieces of material such as paper or cloth.

the-flying-animator.com
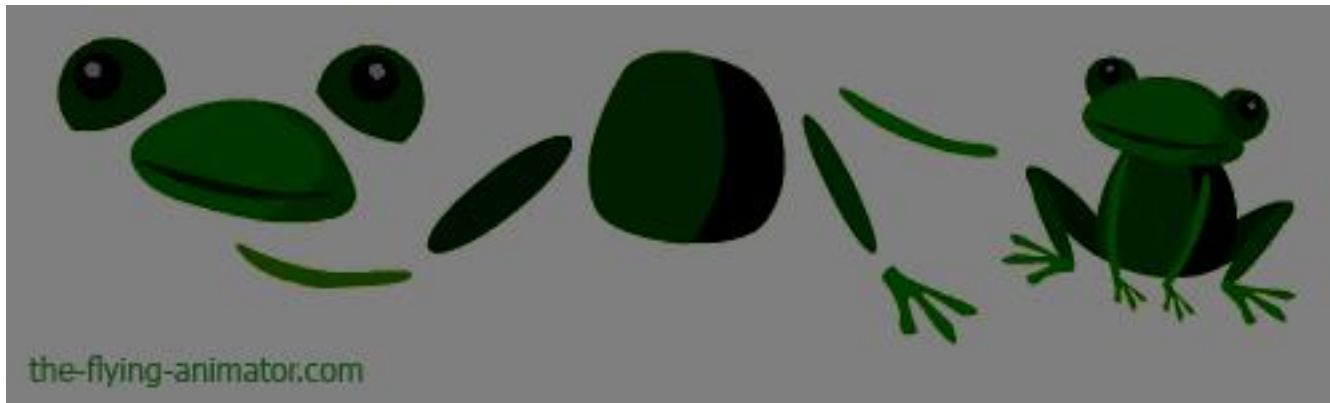
7- Rotoscoping

•where animators trace live-action movement, frame by frame. The source film can be directly copied from actors' outlines

•الحدود الخارجية لصورة الشخصية

•into animated drawings

8-Silhouette animation

• is a variant of cutout animation in which the characters are converted to shadows and only visible as silhouettes

• ظلال .

Part 2-
Animation Principles and visualization techniques

Recommended Textbook (This Lecture)

3D Animation Essentials, Andy Beane ,chapter 4

3-Character Animation

•12 Basic Principles of Animation (*Created at at Walt Disney Animation Studios*):

1. Squash and stretch
2. Anticipation
3. Staging
4. Straight-ahead action and pose-to-pose
5. Follow-through and overlapping action
6. Slow-in and slow-out
7. Arcs
8. Secondary action
9. Timing
10. Exaggeration
11. Solid drawing
12. Appeal

Reference Playlist

- Excellent video playlist ([youtube](#))

- [https://www.youtube.com/playlist?list=PL-bOh8btec4CXd2ya1NmSKpi92U_l6ZJd](#)
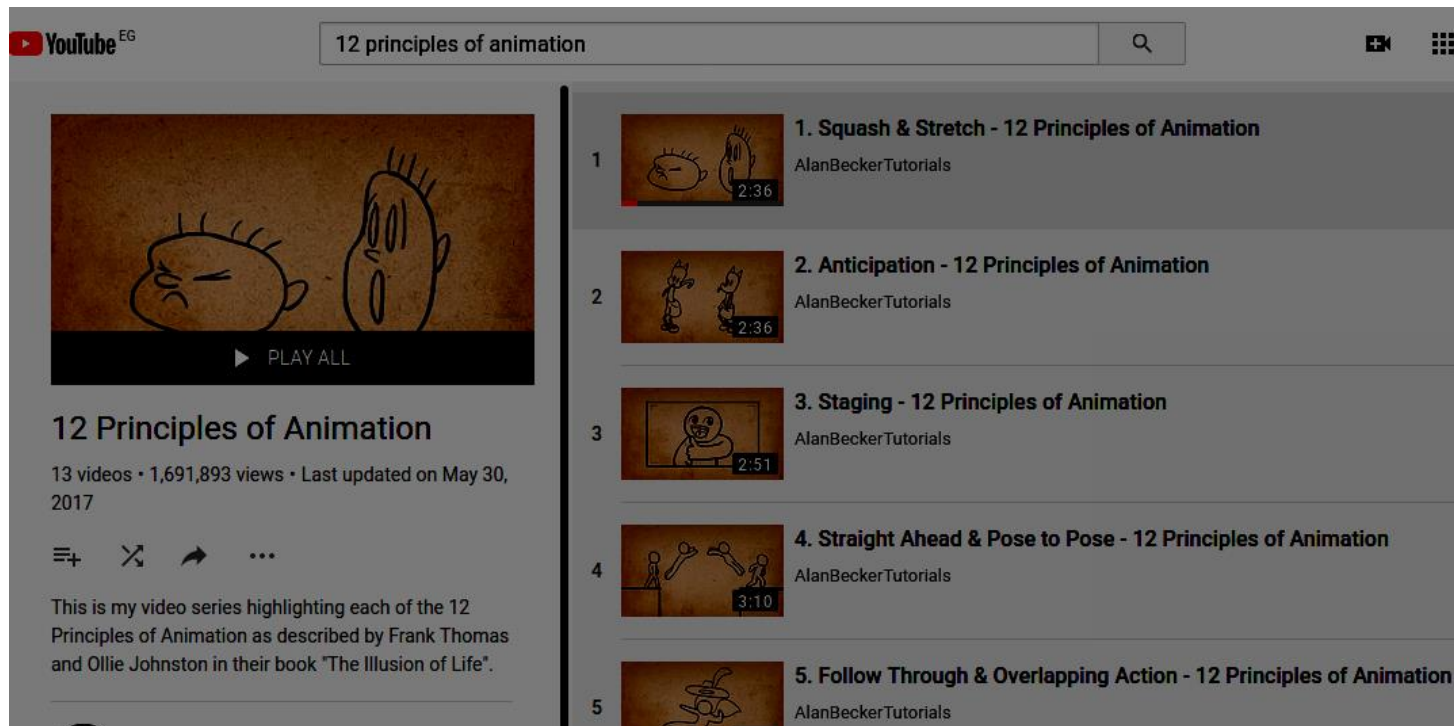
3-1.Squash & stretch
ضغط وتمدد

- A given object possesses some degree of rigidity صلابةand intuitively has some amount of mass كتلة.

- This is reflected in the distortion (squash and stretch) of its shape during an action.

Squash & stretch
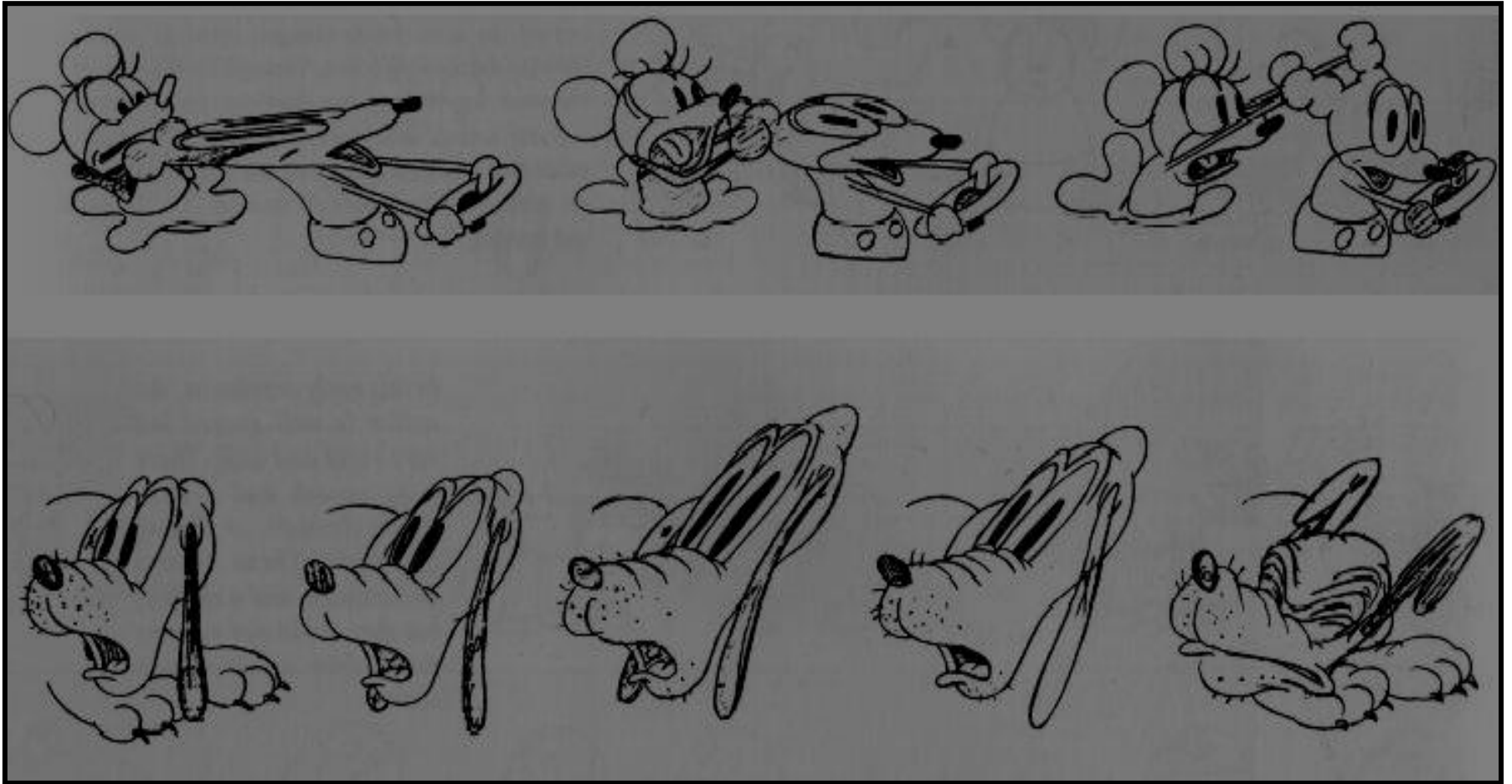
3-2.Anticipation

- The **setup** before the main action.

- It lets the audience know that something is about to happen and gathers the necessary energy.

- Why?

– In animation, a viewer's eye ends to lag behind تأخير by a few frames, so it is often necessary to announce that something is about to happen so your audience doesn't miss it.

- Anticipation are often physical movements in the opposite direction of the intended motion.

– Exception: a growl before an attack or a deep breath before a bold statement.

3-3.Staging (Composition)

•Successful staging will result in the clear presentation of an idea.

•Choose the best scene layouts,T iming and camera angles to deliver that story with appropriate style and mood.

•If a scene has a singular focus, you need to decide whether it should be centered or perhaps *off* to the side a bit.

•Maybe the best staging for a particular shot would be to put its main focus off the screen (Rule of Third

•تقسيم الشاشة لثلاثة أجزاء بالعرض والطول) and reveal the action with sound or dialogue.

Staging

- Poses should be strong, and elements of your scene should work well together visually.

- Typically, you should not hide important scene elements behind less important elements

Staging
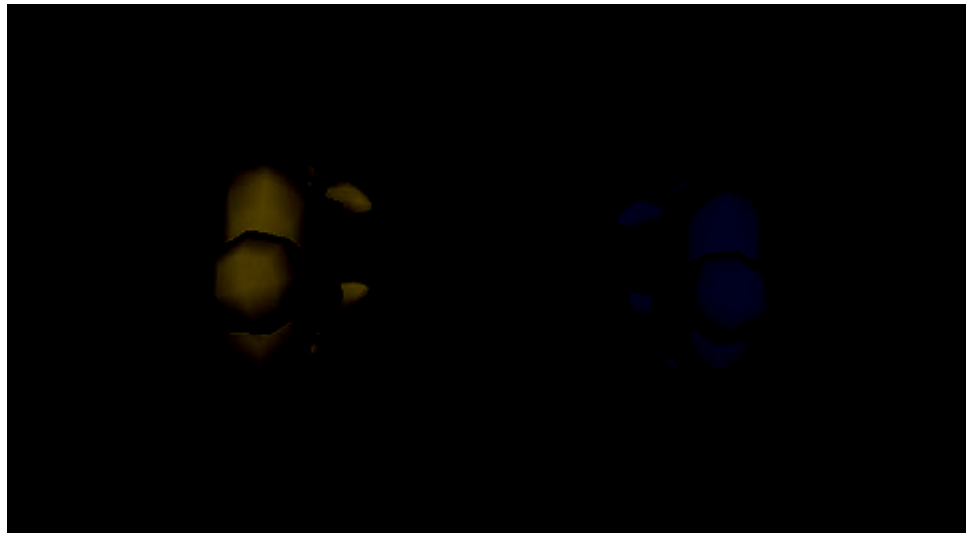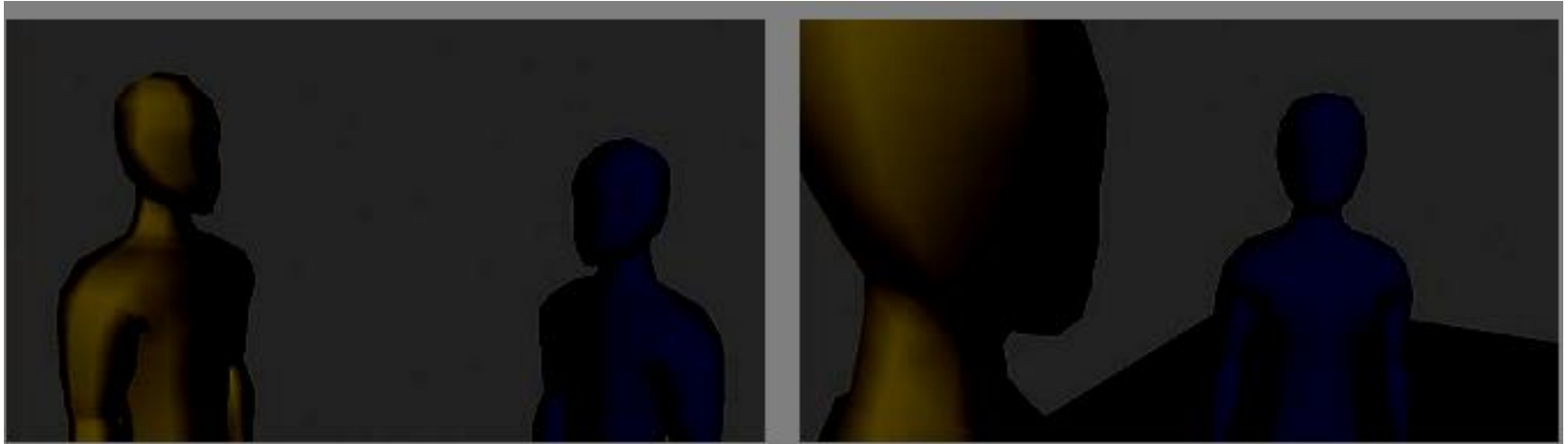
The rule of thirds Example

Image © David Hannon, oil on canvas

3-4.Pose-to-pose & straight ahead action

Straight Ahead

- Involves stepping through the individual frames of your scene and manipulating them sequentially.

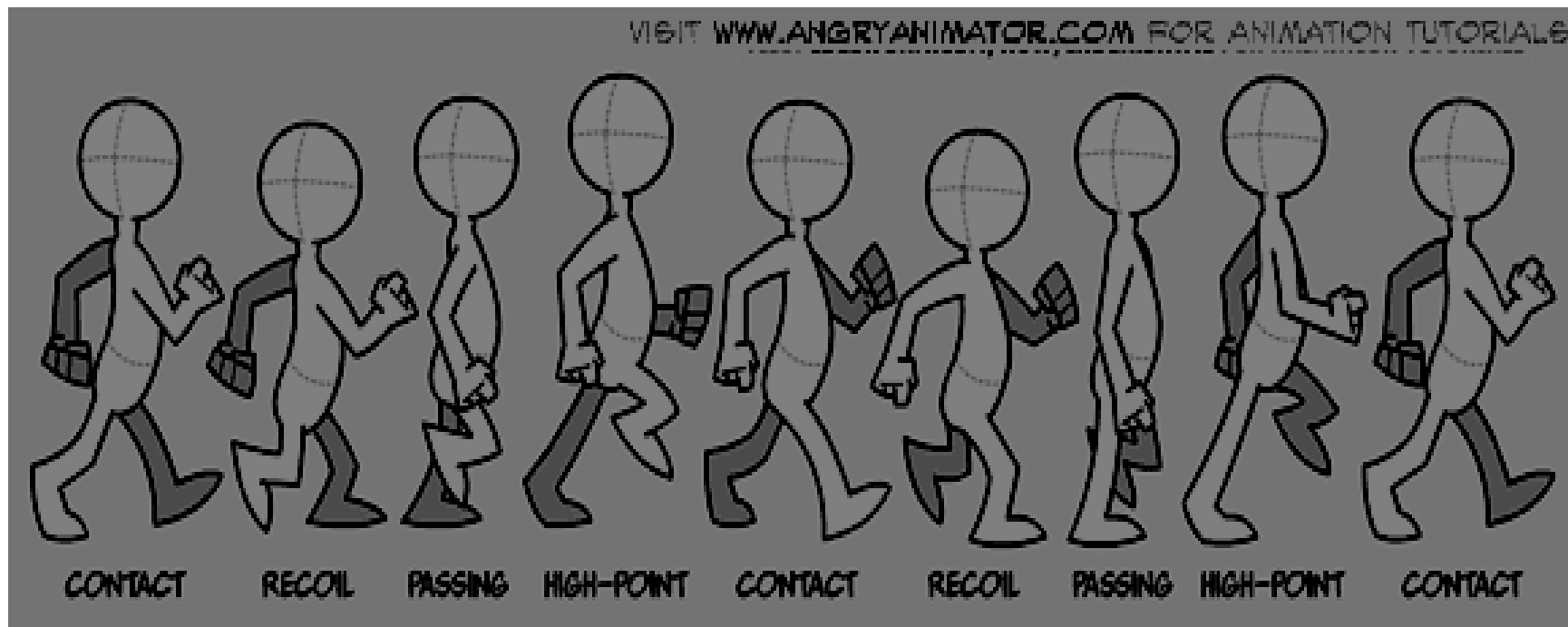- Requires more pre-planning

- Used for stop motion

Pose to pose

- Involves defining the extremes & then filling in the spaces in between.
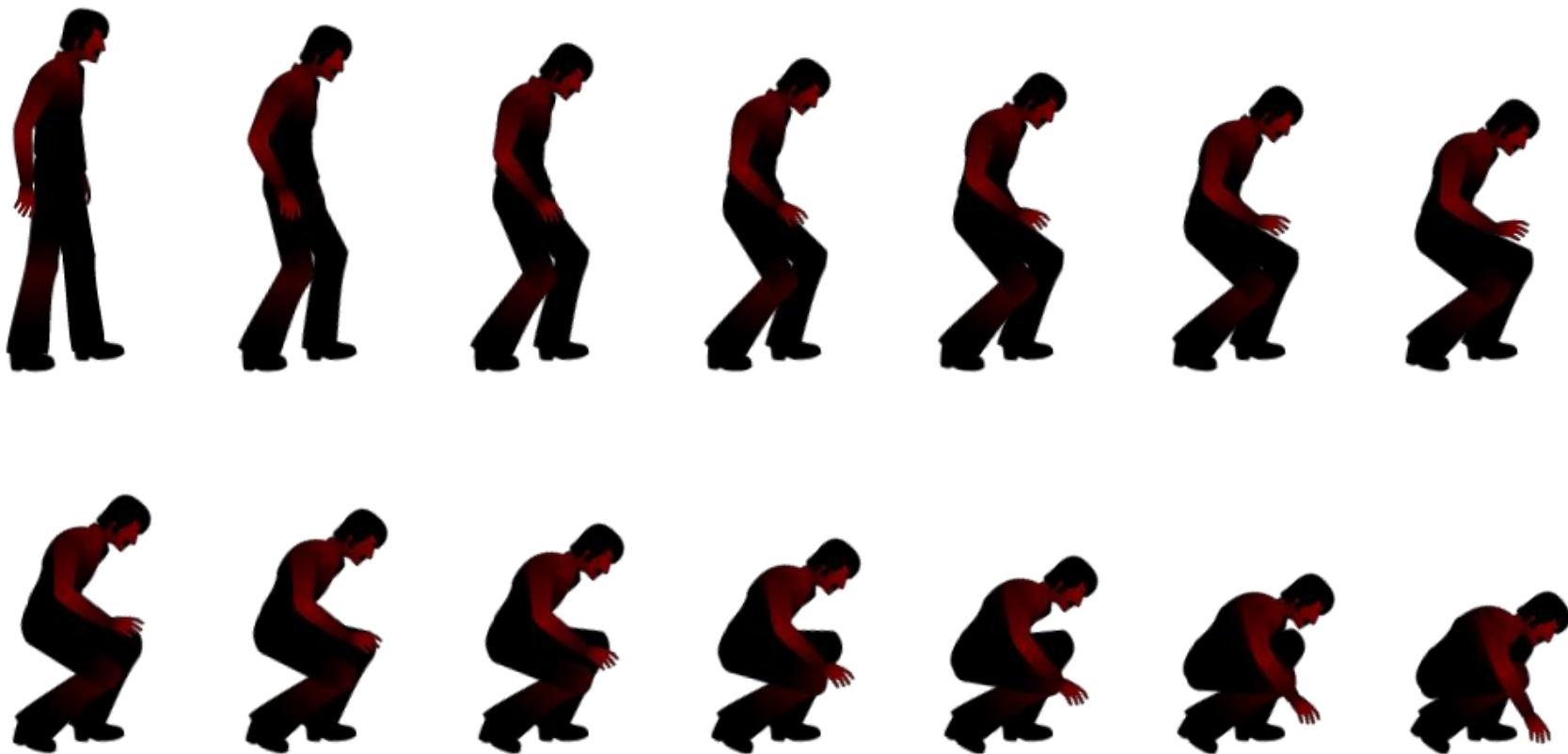
Example for Animation Sheets

- Example 1: Running Horse
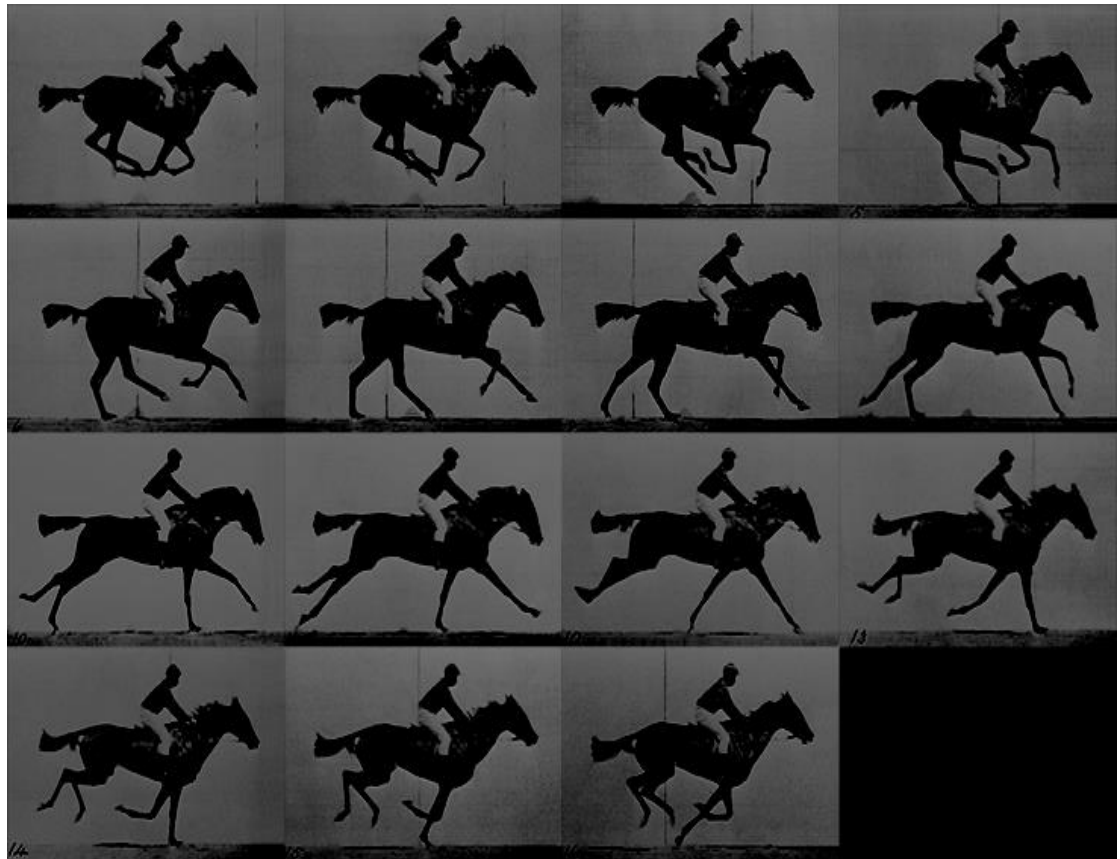
Example 2

Example 3

3-5.Follow-through & overlapping action

تداخل التوقيتات +قصور ذاتي للزوائد

- The extension of a main action.
- Occur when a secondary **appendage**
- زوائد , e.g. a tail or an antenna, is indirectly driven by the primary motion of the body.
- Occur later than the main action as the force dictating the main action takes longer to reach the appendages.
- When an object in motion changes direction, stops accelerating, or stops completely, secondary parts of that object will continue in original direction after the change in the main force.
- Follow-through is often reduced if an appendage has & uses its own muscles.
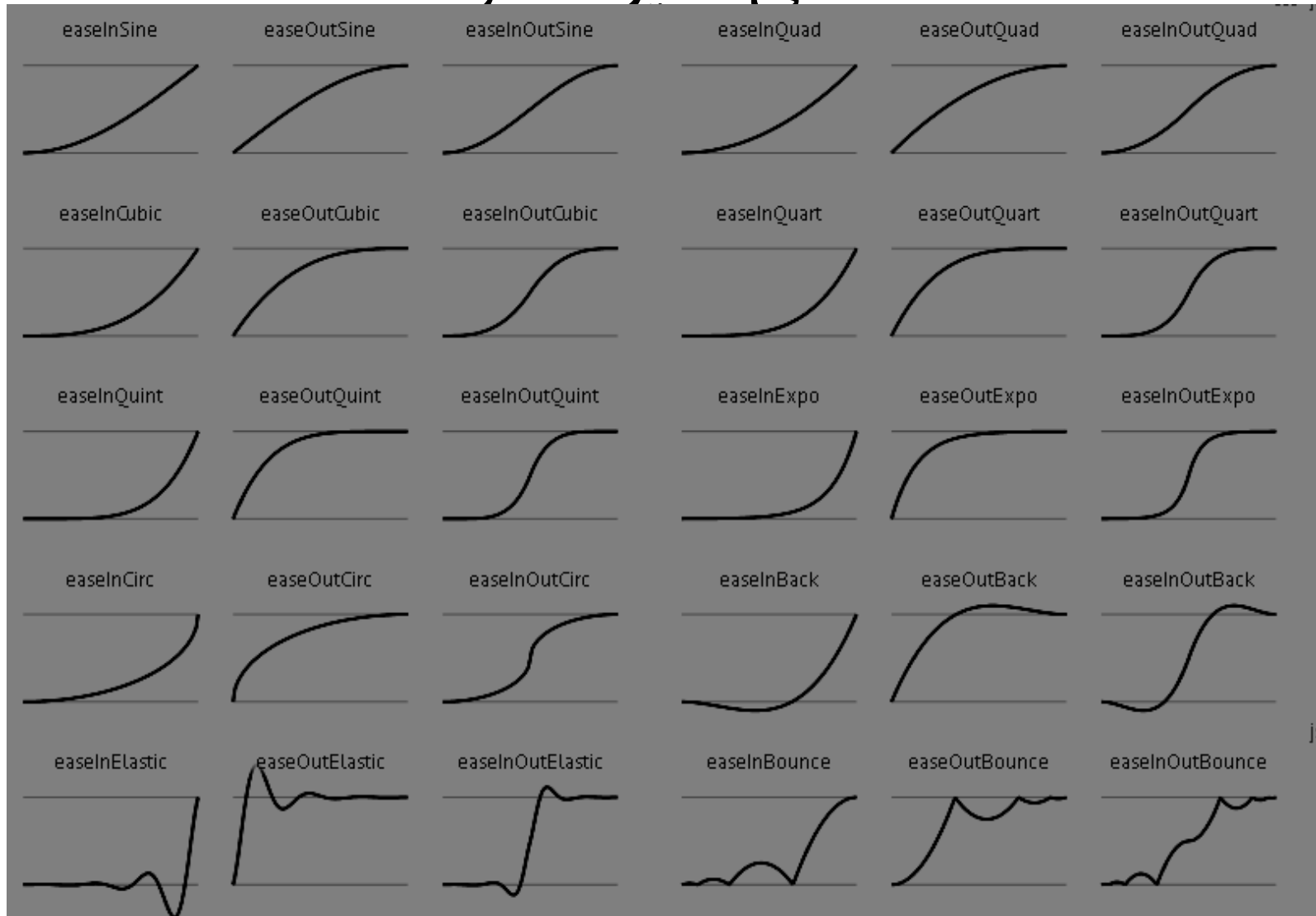
Follow-through & overlapping action
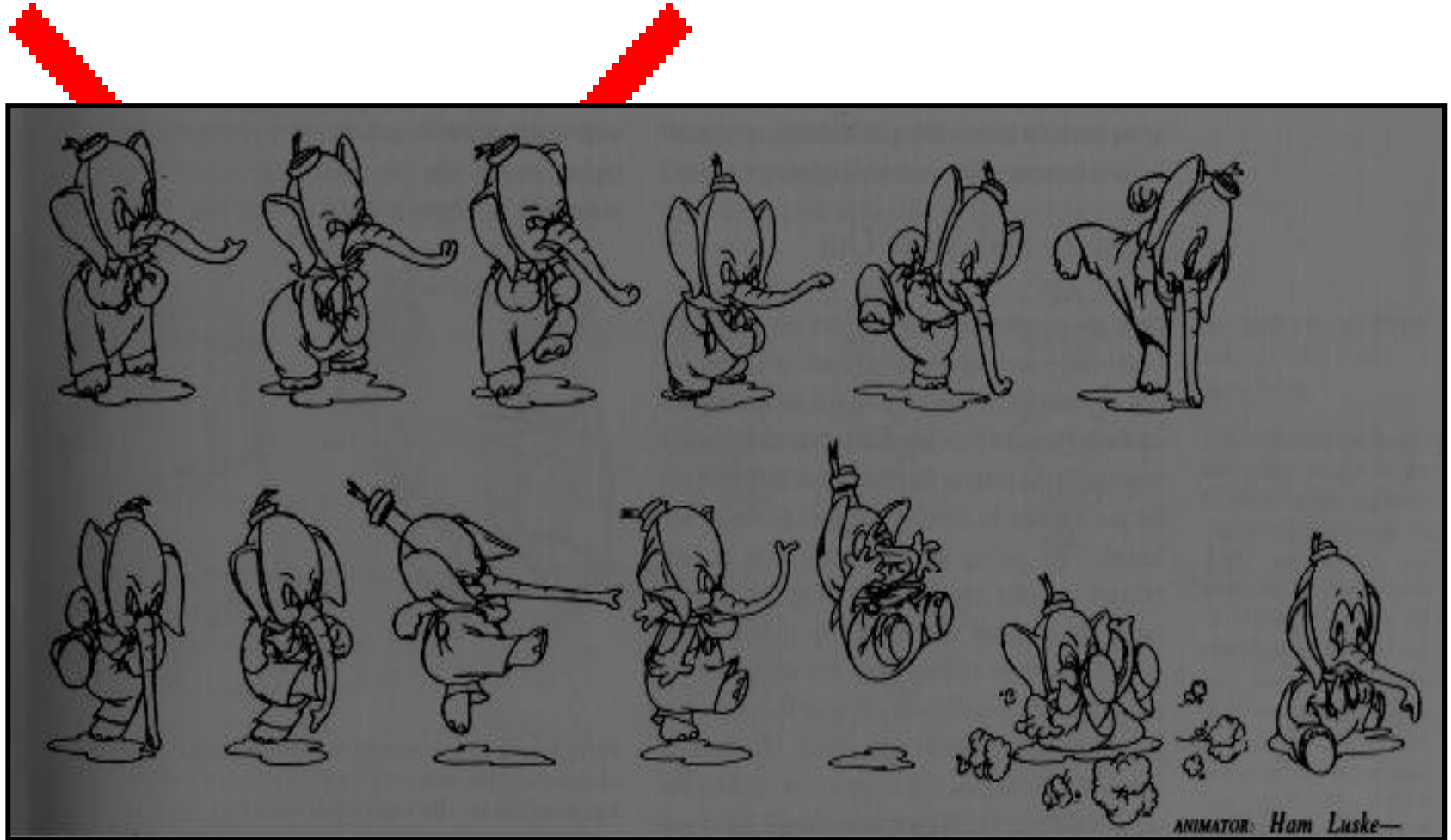
3-6.Slow-in and slow out

•The **acceleration** and **deceleration** of action, i.e. the way that an action begins slowly, moves quickly through the main action, and then slows down as it comes to a stop rather than moving linearly from one position to another.

Tip: Easing Functions
منحنى تغير السرعة

3-7.Arcs
أقواس

•The visual path of action for natural movement.

•Straight-line paths look mechanical.

•Arcs show the most natural route.

•With computer animation the in between frames may follow straight lines - may need to add a curve too it to make it more realistic.

•Using arcs to animate the movements of characters helps achieve a natural look because most living creatures move in curved paths, never in perfectly straight lines.

3-8.Secondary action
فعل هامشي

•Any motion that is secondary to the main action.

•Secondary action often reveal emotional subtleties or hidden thoughts.

•If secondary action pulls the viewer's attention away from the main action, however, it becomes the main action.

3-9. Timing

• Realism of an object depends on how it looks and moves.

• The precise moment and the amount of time that a character spends on an action.

• Timing adds emotion and intention to the character's performance.

• The timing of a response to someone calling us, or a noise; tells a lot about our attitude, confidence and disposition.

Timing

- Not too slow or audience eyes wander.
- Not too fast or action is misunderstood.
- The faster the motion, the more critical it is to make the audience focus on it.

3-10.Exaggeration

- To increase the readability of emotions and actions.
- To effectively tell the story of a particular performance.
- Animation medium don't deliver all of the same information that exists in real life.
- Effective exaggeration isn't always a matter of making a motion larger.
- Significant *decreasing* an action is also a type of exaggeration.
- E.g. completely stopping a character's motion for an unnaturally long period of time can demonstrate a particular emotion, e.g. shock or disgust.
- Exaggeration is especially appropriate in **cartoon-style** animation.

3-11. Solid drawing /modeling & rigging

- Each individual drawing should be a work of art **on its own**.

- Each drawing should look like the character **being presented**.

- Avoid **inconsistency** in terms of character's size & volume

3-12. Appeal
جذب انتباه

•In the context of animation, appeal means that a character or its performance is **visually interesting**.

•The most **subjective** of all the principles.

•Is the presentation of your idea pleasant (or unpleasant) to look at?

•Apply fundamental **aesthetic principles** to your shots—though appeal doesn't necessarily mean attractive.

Part 3
Lighting &
Camera Movements

1-Three-Point Lighting

Most basic type of lighting setup is the **three-point light set:**

Key light

Fill light

Rim/kicker light(s)
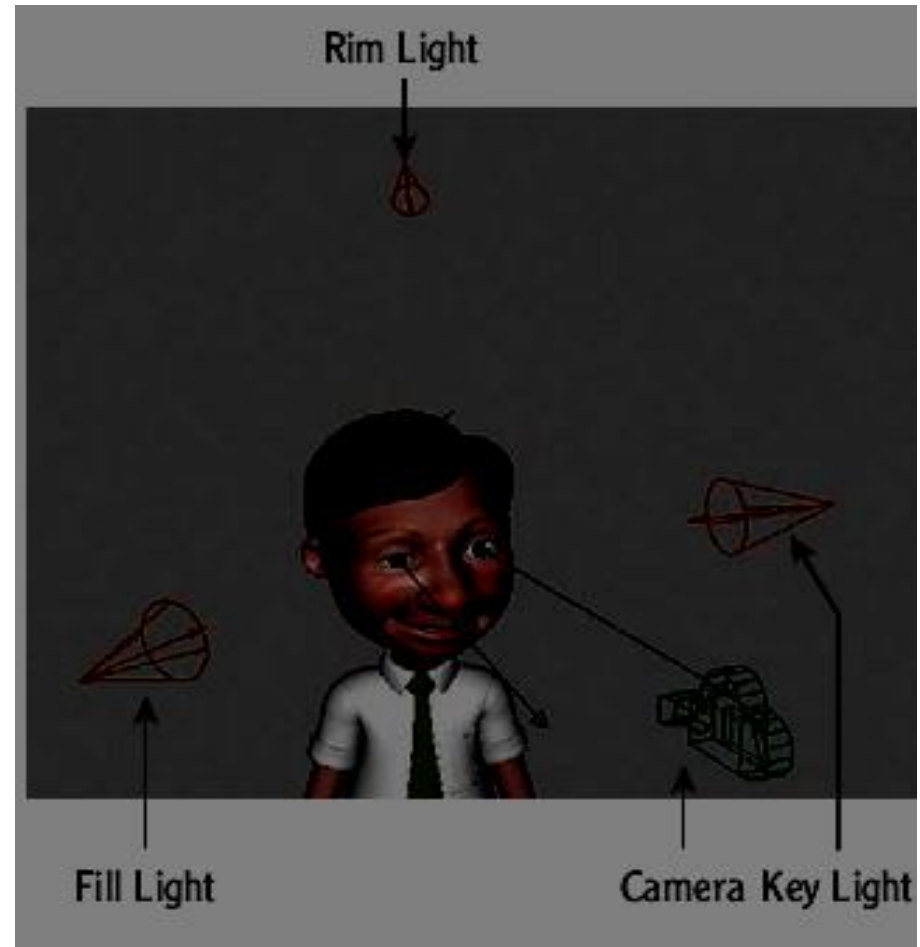
1-1 Key light

Left Or Right to the camera and **above the subject.**
 This is the <u>strongest</u> light in the setup
casts the **majority** of the light
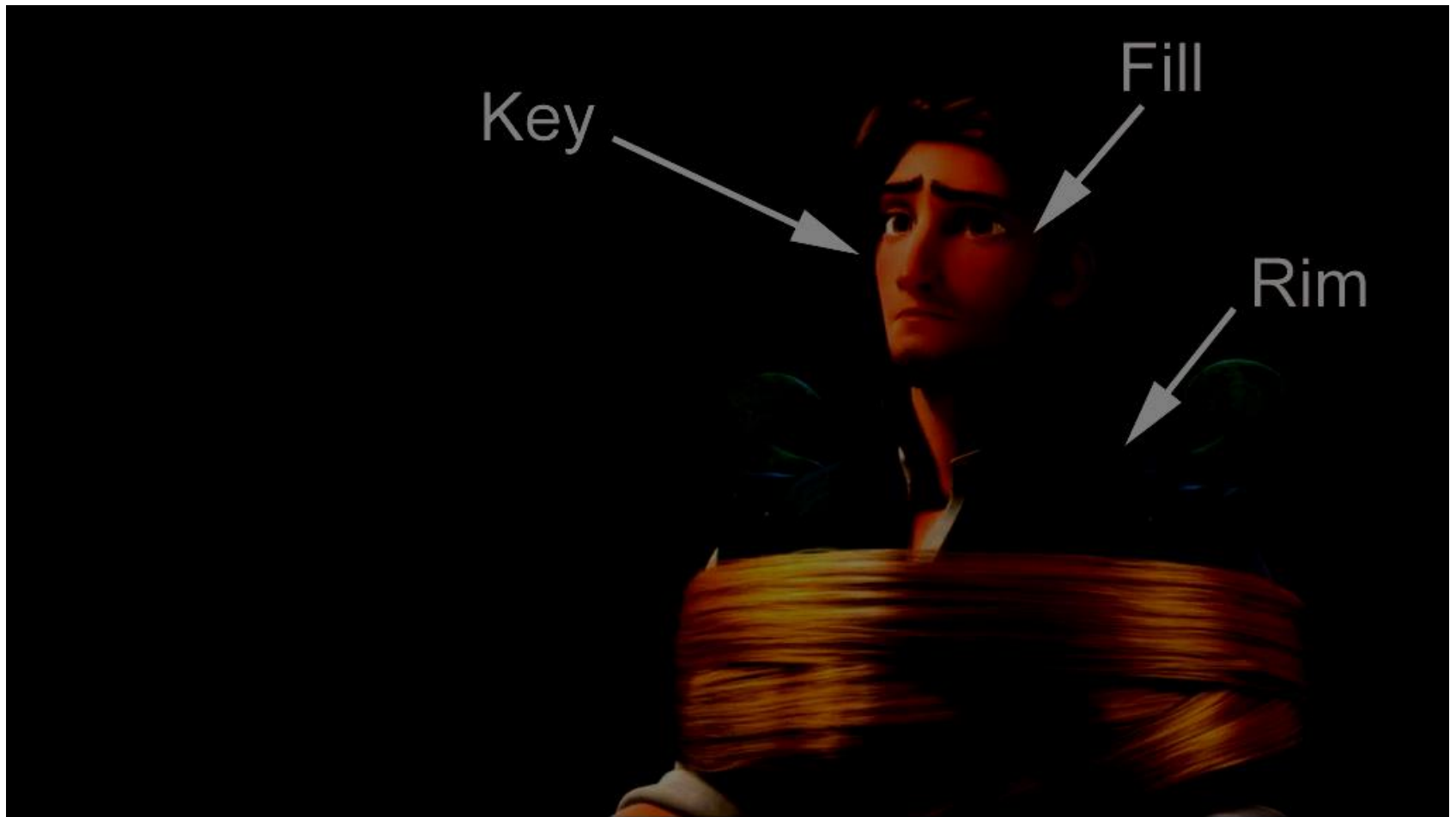
1-2 Fill light

opposite the key light

at the same level or slightly lower than the subject.

The fill light has half of the light intensity of the key light

prevents the shadows of the subject from dropping
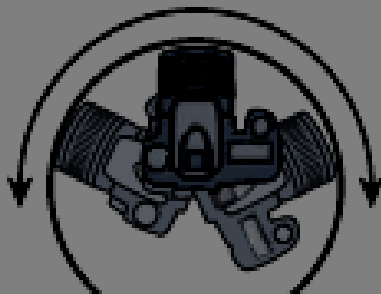down to pure black

Example

Camera Movements

1.Pan A pan (Figure 4.40) is a rotation of the camera side to side, horizontally

2.Tilt A tilt (Figure 4.41) is a rotation of the camera up or down, vertically.
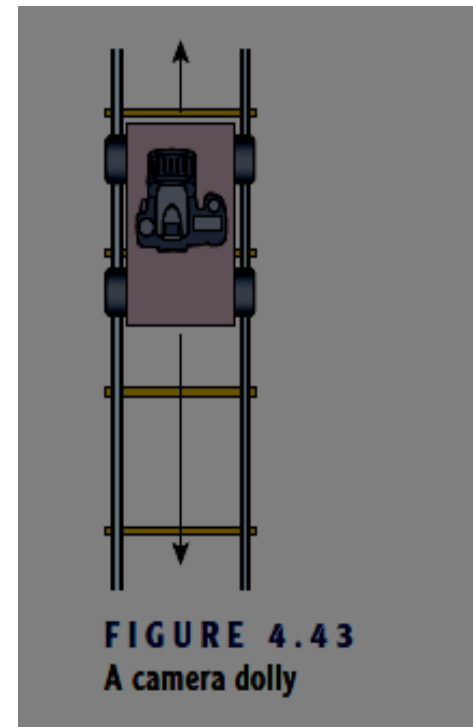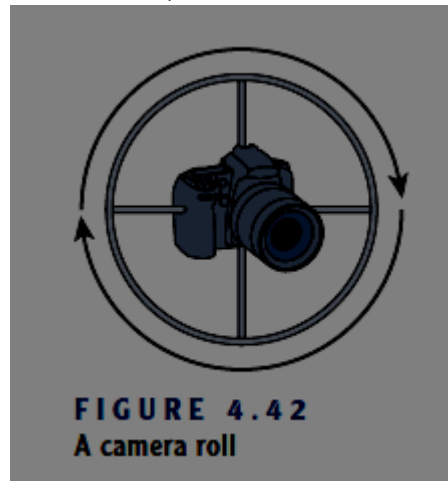

FIGURE 4.40
A camera pan

FIGURE 4.41
A camera tilt

Camera Movements (2)

3- Roll This involves rotating the camera on its side or upside down. This kind of camera rotation will allow the camera to roll 360 degrees if needed for effect.(See Figure 4.42.)

4- Dolly A dolly (Figure 4.43) can move the camera forward or backward, toward or away from the subject. This move can be completed by hand, but typically the camera is mounted to a dolly مزلقان مستقيم that is pushed.

FIGURE 4.42
A camera roll

FIGURE 4.43
A camera dolly

Camera Movements (3)

5-Track You can move the camera along a track مزلقان بأي شكلto follow the subject,

6-**Pedestal** A pedestal (Figure 4.45) tracks the **height** of an object by moving the camera **vertically**.

FIGURE 4.44 A camera track

FIGURE 4.45
A camera pedestal
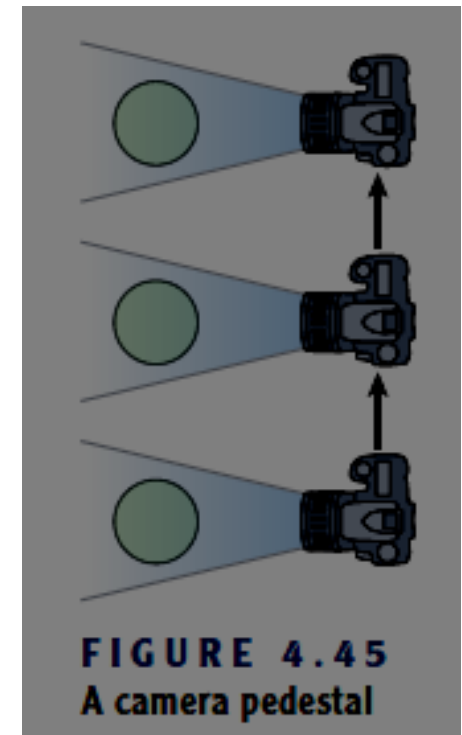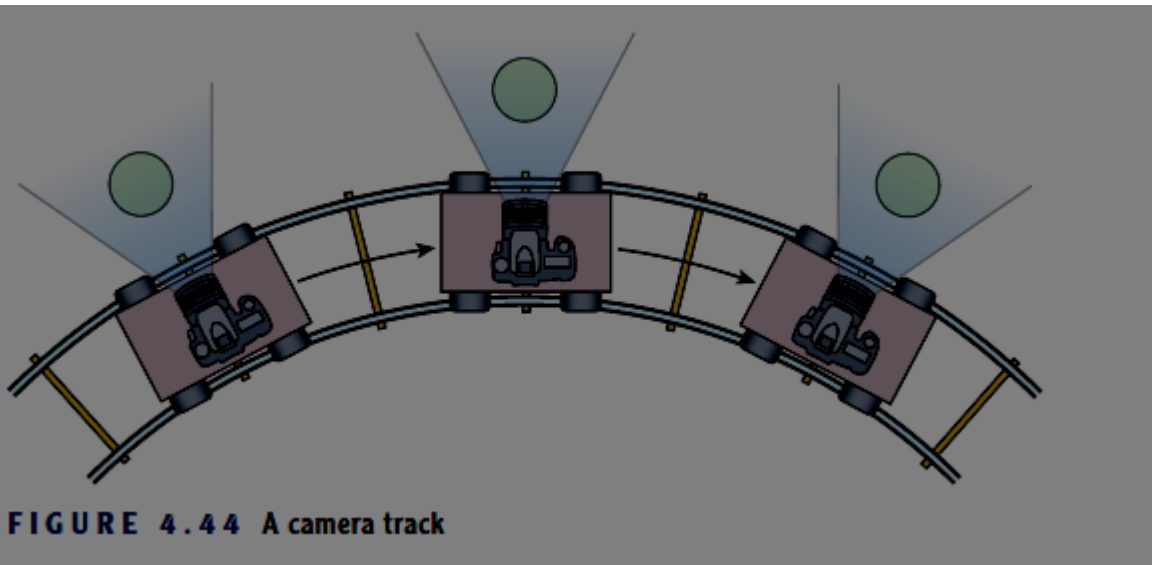
Camera Movements (4)

6- **Crane** shot: رافعة

film technique where the camera takes a shot from a crane/drone. A crane enables the camera to traverse great distances up and down, as well as forward and backward and from side to side, while flying high in the air.

Camera Movements (5)

7- Zoom A zoom is different from the other camera moves because you are not physically moving the camera, but rather moving the lens to create movement in the frame.


FIGURE 4.46 Two shots of the same character, using a wide-angle view (left) and a zoomed-in view (right)

Camera Movements (5)

8-Handheld A handheld movement (Figure 4.47) entails holding the camera and moving freely in any direction to follow the action. This movement is good for horror or action films because it allows the viewers to feel as if they are actually immersed in the action.

But handheld Movement must be used wisely; overuse can be jarring to the audience. (Example Cloverfield movie).

FIGURE 4.47
Example of a handheld camera move
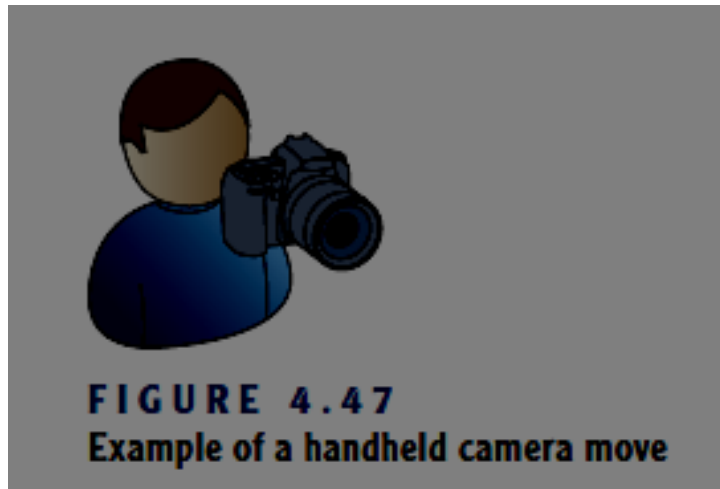
Camera Movements (6)

9-Rack Focus This involves changing the focus of the camera to push an object that is out of focus into focus, or vice versa. Figure 4.48 shows an example. In the 3D animation world, rack focus can be accomplished in the software's 3D camera but is typically achieved in the postproduction stage with compositing software.

FIGURE 4.48 Example of a rack focus change from one object to another

References

- www.roguebasin.com/index.php?title=Dungeon-Building_Algorithm - www.roguebasin.com/index.php?title=Java_Example_of_Dungeon-Building_Algorithm - openprocessing.org/visuals/?visualID=18822
- - www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation - stackoverflow.com/questions/4997642/simple-example-of-bsp-dungeon-generation
- - pcg.wikidot.com/pcg-algorithm:dungeon-generation - donjon.bin.sh/dungeon/about/ - www.policyalmanac.org/games/aStarTutorial.htm