Computer Graphics (Finals) - Jaish Khan

It is the branch of science and technology concerned with methods and techniques for
converting data to or from visual presentation using computers.

It is used in many different areas like CAD, Education and Entertainment and creating things like GUIs, Art, Simulations/VR and Visualization.

A **Graphics API** is used for creating/interacting with graphics without having to deal with low level system system stuff like window handling.

There are 6 components in a Computer Graphics system.

- CPU, GPU, Input, Output, Memory, Frame Buffer

Graphics/Display Controller → Translator between the computer and the the display.

Common Resolutions

Resolution Name	Dimensions (pixels)	Aspect Ratio
Ultra High-Definition 8K (8K UHD)	7680 × 4320	16:9
Ultra-Wide High-Definition 5K (5K UHD)	5120 × 2160	21:9
Ultra High-Definition 4K (4K UHD)	3840 × 2160	16:9
Ultra-Wide Quad High-Definition 2K (UWQHD)	3440 × 1440	21:9
Quad High-Definition 2K (QHD)	2160 × 1440	16:9
Ultra-Wide Full High-Definition 1080p (UWFHD)	2560 × 1080	21:9
Full High-Definition 1080p (FHD)	1920 × 1080	16:9
High-Definition 720p (HD)	1080 × 720	16:9
Standard-Definition 480p (SD)	640 × 480	16:9

Graphics Pipeline

The processes it takes to render graphics on the screen. It can generally be divided into 3 parts.

1. Application Step

This stage happens entirely on the CPU and is controlled by your application.

This is where you define the scene data

- 3D Models making up the scene.
- Materials applied to those models.
- Lights illuminating the scene.
- Camera position/orientation.

Animation, Collision detection and Scene management also happen here.

2. Geometry Step

This is where all the objects are prepared for rendering.

This prepares the 3D models for the rasterization step.

- Vertices are transformed (by model, view or projection).
- Shaders are used (like Vertex Shader, Geometry Shader etc.).

Clipping (The process of removing geometry that is not in view) also happens here.

3. Rasterization Step

This is where primitives (usually triangles) are converted into fragments.

This bridges the gap between 3D and 2D.

- Checks which triangles are visible from the Camera.
- Things like **Culling** are used to removing triangles that are not seen by the Camera.
- Lighting, Texture, Blending and Shading calculations are done.

At the end **Fragment Shaders** are used to determine the final pixel (color) that is to be shown based on the calculations done.

Displaying Images

Display Attributes

Refresh Rate → The redrawing of image on a screen.
It is measured in Hertz.
Higher refresh rate results in smoother motion and better for eye comfort.

Contrast Ratio → The difference between the brightest and the darkest colors that a display is able to produce.
It is shown using a ratio.
Higher contrast ratio results in darker blacks and brighter whites.

Response Time → The time it takes for a monitor to change from one color to another.
It is measured in milliseconds.
Higher response time results in reduced ghosting (images leaving a trail behind).

Display Resolution → The total amount of pixels used in the screen.
It is shown using width x height.
Higher resolution results in sharper/clearer images.

Pixel Density → The average amount of pixels per an inch of screen.
It is measured in PPI (pixels per inch)
Higher pixel density results in sharper/clearer images.
`To calculate` → $\sqrt{(Width^2 + Height^2)}/DiagonalInches$

Aspect Ratio → The width divided by the height of a display.
It is shown using a ratio.
Common aspect ratios include `16:9` (Wide), `4:3` (Standard) and `21:9` (Ultra Wide).
`To calculate` → Reduce $Width/Height$ into the simplest form.

Frame Buffer → The area where image information of next frame is temporarily stored.
It is calculated in terms of Bytes but can be converted to KBs.
Higher resolutions and require bigger frame buffer.
`To calculate` → $(Resolution \times BitsPerPixel)/8$

Bit Depth → No of bits per pixel.
Bitmap (Frame buffer with 1 bit depth), **Pixmap** (Frame buffer with 1+ bit depth).
High Color → 16 bit depth, **True Color** → 32 bit depth.

Pixel Access Rate → $Resolution \times Framerate$
Access Time per Pixel → $1/AccessRate$

Types of Displays

CRT (Cathode Ray Tube)

Uses an electron gun on a phosphorus film to create images.

A **focusing system** and a **deflection system** are used to aim the light exactly where we want.

- **Monochrome CRT** (Black and White) → Used a single white colored electron gun.
- **Color CRT** (Full Color)
  - **Beam Penetration Method** → Uses 2 color films of phosphor. Color depends on how far the beam penerates in the layers.
  - **Shadow Masking Method** → Uses 3 color dots of phosphor and 3 electron guns of each color. Color depends on which gun is used.

Raster Scan

The electron beam is swept across the screen, one row at a time from top to bottom.

Each row that is scanned is called a **Scan Line**. These scan lines can be scanned one-by-one in order (**Progressive**) or in odd-even pairs (**Interlacing**).

Horizontal Retrace → When it scans an entire row it goes to the start of the next row.
Vertical Retrace → When it scans an entire screen (all rows) it goes to the start of the screen.

Random Scan (Vector Scan)

The electron beam is directed to the parts of the screen where a picture is to be drawn.

Flat Panel Displays

Non-Emissive Displays

They borrow light from an external light source (usually a backlight).

Transmissive Displays (backlight as a light source)

LCD (Liquid Crystal Display) → Uses an RGB filter with a CCFL backlight to create images.

- **TN** (Twisted Nematic) → Uses liquid crystals that twist to pass light through. Best refresh rate and response time.

- **IPS** (In-Plane Switching) → Uses liquid crystals that are fixed and aligned with the glass.
  Best color and viewing angle.
- **VA** (Vertical Alignment) → Mix of TN and IPS.
  Average for everything.

Reflective Displays -> (reflections as a light source)

Transflective Displays -> (both Transmissive and Reflective)

Emissive Displays

They create their own light and each pixel is capable of individually switching on/off which results in pure blacks (high contrast ratios).

LED (Light Emitting Diode) → Uses a layer of pixels infront of an LED backlight.

- **OLED** (Organic LED) → Uses a layer of pixels which have their own light.
  - **AMOLED** (Active Matrix OLED)

PDP (Plasma Display) → Uses small pockets of gas/plasma. Very expensive and not much used.
FED (Field Emission Display) → Uses a field of electron to target a phosphorus film to create color. Not used these days.

Touch Enabled Displays

Displays that can also be provided input by using a conductive surface like a finger or a pen.

They have a **digitizer** layer which creates an electric field, which when disturbed catches touch inputs.

Image Formats

Raster Image Formats

Raster Image Storage → Stored in individual points to display.

1. JPEG (Joint Photographic Experts Group) → Lossy Format, uses an algorithm to remove excess information that human eyes cannot see.
   - 8 bits per channel for R, G, B each → 24 bits total.
2. PNG (Portable Network Group) → Lossless Format, allows for transparency.
   - PNG-32 → 8 bits, 4 channels (R, G, B, A).
   - PNG-24 → 8 bits, 3 channels (R, G, B).
   - PNG-8 → 8 bits, single chanel.

3. GIF (Graphics Interchange Format) →
4. TIFF (Tagged Image File Format) →
    - 16 bits per channel for R, G, B

Vector Image Formats

Vector Image Storage → Stored in instructions to draw with.

1. SVG (Scalable Vector Graphics) → Used for drawings/icons/shapes.
2. PDF (Portable Document Format) → Used for finalized documents.

Color Representation

Human Vision is more suited to Brightness/Darkness compared to it being suited to Hue(Color).

Additive Color Model → Based on emitted light. Combining the primary colors results in *White*. Used for displays.
Subtractive Color Model → Based on absorbed light. Combining the primary colors results in *Black*. Used for pigments.

Color Models

A theoretical system used to describe colors by using numbers.

Color Model	Primaries	Type
RGB	Red, Green, Blue	Additive
CMY	Cyan, Magenta, Yellow	Subtractive
HSV	Hue, Saturation, Value	Additive
HSL	Hue, Saturation, Lightness	Additive
LAB	`L*, a*, b*`	Additive

Color Spaces

A practical implementation based on a color models which decides how many colors are available for use.

Color Space	For	Model	Human Perception
sRGB	Displays	RGB	
AdobeRGB	Photography	RGB	50%
ProPhoto RGB	Photography	RGB	76%

Color Space	For	Model	Human Perception
CMYK	Printing	CMY	
CIELAB	Human Perception	LAB	100%

Drawing Graphics

Line Drawing

Drawing even a single line on the screen is a complicated process.

The general approach would be to use the line equation.
Slope-Intercept Form → $y = mx + b$

```
def naive_line(x0, y0, x1, y1):

    #Calculate the difference between x and y values.
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)

    #Calculate the slope.
    m = dy / dx
    c = y0 - m * x0

    points = []

    if dx > dy:
        start, end = sorted([x0, x1])
        for x in range(start, end + 1):
            y = m * x + c
            points.append((x, round(y)))
    else:
        start, end = sorted([y0, y1])
        for y in range(start, end + 1):
            x = (y - c) / m
            points.append((round(x), y))

    print(f"Points are: {points}\n")
```

But this approach is extremely slow and requires the floats and rounding numbers.

Digital Differential Analyser Algorithm

A line drawing algorithm that uses floating point arithmetic.

```
def dda(x0, y0, x1, y1):

    #Calculate the difference between x and y values.
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)

    #Compare dx and dy and choose the bigger number as number of steps.
    steps = max(dx,dy)

    #Calculate the increment in each axis.
    x_inc = dx / steps
    y_inc = dy / steps

    #Initialize the loop variables.
    x, y = x0, y0

    #Loop and append the points to a list.
    points = []
    for i in range(int(steps)+1):
        points.append((round(x),round(y)))
        x += x_inc
        y += y_inc

    #Printing the points list.
    print(f"Points are: {points}\n")
```

Bresenham's Line Algorithm

A fast line drawing algorithm that only uses integer operations.

It is incremental and avoids having to round down numbers and avoids using floats. It is also called the Mid-Point Line Drawing Algorithm.

```
def bresenham_line(x0, y0, x1, y1):

    #Calculate the constants.
    dx = x1 - x0
    dy = y1 - y0
    dx2, dy2 = 2*dx, 2*dy
    D = dy2 - dx2

    #Calculate the initial decision parameter.
    P0 = dy2 - dx

    #Check for slope being greater than 1.
    m = dy / dx
    if m < 1:

        #Initialize the loop variables.
        x, y = x0, y0

        #Loop and append the points to a list.
        points = []
        for x in range(x0, x1+1):
            points.append((x,y))

            if P0 < 0:
                P0 = P0 + dy2
            else:
                P0 = P0 + dy2 - dx2
                y = y + 1

        #Printing the points list.
        print(f"Points are: {points}\n")
    else:
        #Printing in case slope is bigger than 1.
        print(f"Slope is greater than 1: {m}\n")
```

Circle Drawing

Drawing a circle requires different thinking compared to drawing a line and there are many approaches to it.

Cartesian Coordinates Circle Drawing

It is the simplest circle drawing algorithm would use the **Cartesian Coordinates** to calculate the points using the Equation of Circle:

$$x^2 + y^2 = r^2$$

Tweaking this equation gets us → $y = \pm\sqrt{(r^2 - x^2)}$

We can get every point by putting values for x in the range of $[-R, R]$.

Polar Coordinates Circle Drawing

Other approach can be to use **Polar Coordinates** instead using these equations.

$$x = r \times \cos\theta + x_c$$
$$y = r \times \sin\theta + y_c$$

Here, x_c and y_c are center offsets and are 0 if the circle is at origin. The problem comes in trying to decide an increment for θ (theta).

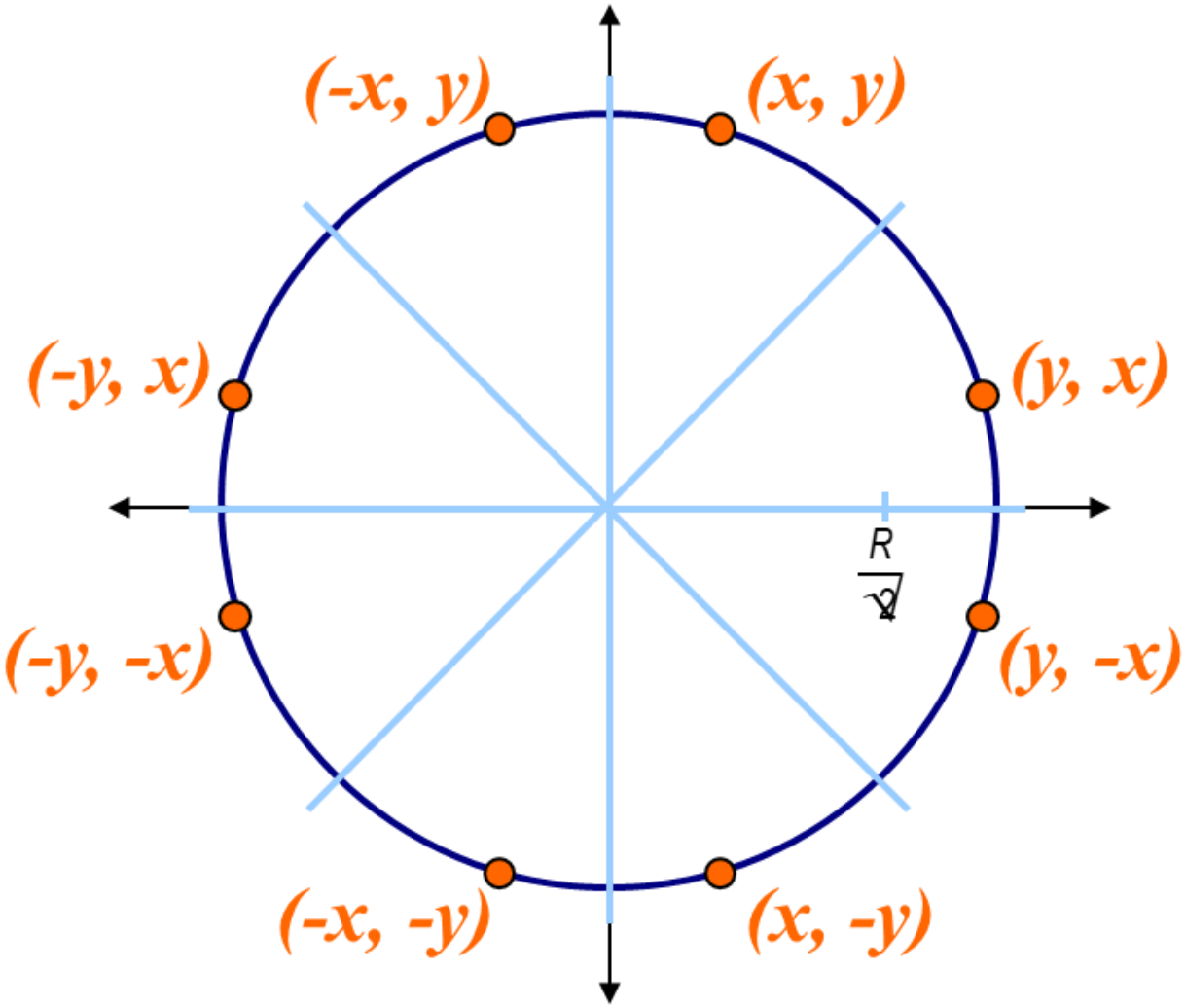
Both of these approaches are inefficient and extremely bad because they use heavy operations like **round**, **sqrt** (Cartesian) and **sin**, **cos** (Polar). These approaches also result in circles that have gaps in them.

Midpoint Circle Algorithm

Works by using the fact that circles have 8-way symmetry.

We can draw 8-points simultaneously because they're mirrors of each other. This saves a lot of resources as we only have to care about a single half of any quadrant

(octet) and we can derive the other 7 octets from it.



Initial Decision Parameter → `1-r`

```
def midpoint_circle(x0, y0, radius):
    x = 0
    y = radius
    d = 1 - radius
    while y ⩾ x:
        # Plot points in the first octant and mirror them in other octants
        plot_point(x + x0, y + y0)  # Quadrant I
        plot_point(y + x0, x + y0)  # Quadrant II
        plot_point(-x + x0, y + y0)  # Quadrant VIII
        plot_point(-y + x0, x + y0)  # Quadrant VII
        plot_point(x + x0, -y + y0)  # Quadrant IV
        plot_point(y + x0, -x + y0)  # Quadrant III
        plot_point(-x + x0, -y + y0)  # Quadrant V
        plot_point(-y + x0, -x + y0)  # Quadrant VI

        x += 1 #increases no matter what

        if d < 0:
            d = d + 2 * x + 3
```

```
        else:
            d = d + 2 * (x - y) + 5
            y -= 1 #decreases only if d is positive or zero.
```

Bresenham's Circle Algorithm

It is a modified version of the midpoint circle algorithm which only uses integer operations. Only the *initial decision parameter* and the way to calculate the next one are different otherwise its the same algorithm.

Initial Decision Parameter → `3-2r`

```
def bresenham_circle(x0, y0, radius):
    x = 0
    y = radius
    d = 3 - 2 * radius
    while y ⩾ x:
        # Plot points in the first octant and mirror them in other octants
        plot_point(x + x0, y + y0)  # Quadrant I
        plot_point(y + x0, x + y0)  # Quadrant II
        plot_point(-x + x0, y + y0)  # Quadrant VIII
        plot_point(-y + x0, x + y0)  # Quadrant VII
        plot_point(x + x0, -y + y0)  # Quadrant IV
        plot_point(y + x0, -x + y0)  # Quadrant III
        plot_point(-x + x0, -y + y0)  # Quadrant V
        plot_point(-y + x0, -x + y0)  # Quadrant VI

        x += 1 #increases no matter what

        if d < 0:
            d = d + 4 * x + 6
        else:
            d = d + 4 * (x - y) + 10
            y -= 1 #decreases only if d is positive or zero.
```

Sir has MidPoint Circle Algorithm named as Bresenham Circle Algorithm in his slides.

Examples

Find the coordinates of the circle. When the coordinates are (0,0) and radius is 8.

Cartesian Coordinates

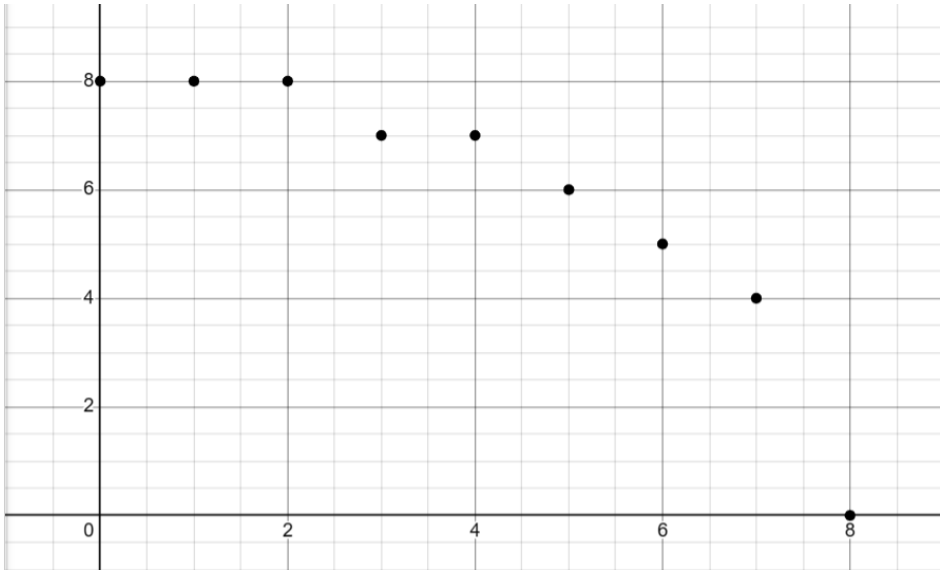
1. **Initial Points** → `x = 0` and `y = r = 8`.
2. We then calculate the next `y` using this equation.

$$y = \pm\sqrt{(r^2 - x^2)}$$

3. Place the initial `r` and `x` into it. `r` stays the same while `x` increases by 1 each time.

$$y = \pm\sqrt{(8^2 - 0^2)} = \pm 8$$
$$y = \pm\sqrt{(8^2 - 1^2)} \approx \pm 8$$
$$y = \pm\sqrt{(8^2 - 2^2)} \approx \pm 8$$
$$y = \pm\sqrt{(8^2 - 3^2)} \approx \pm 7$$
$$y = \pm\sqrt{(8^2 - 4^2)} \approx \pm 7$$
$$y = \pm\sqrt{(8^2 - 5^2)} \approx \pm 6$$
$$y = \pm\sqrt{(8^2 - 6^2)} \approx \pm 5$$
$$y = \pm\sqrt{(8^2 - 7^2)} \approx \pm 4$$
$$y = \pm\sqrt{(8^2 - 8^2)} = \pm 0$$

x_1	y_1
0	8
1	8
2	8
3	7
4	7
5	6
6	5
7	4
8	0
.....	.....



Midpoint (Bresenham)

1. **Initial Points** → `x = 0` and `y = r = 8`.
2. The initial decision parameter `d = 1 - r = -7`
3. We can calculate the next points by
   - `x` increments by 1 each time.
   - `y` only decrements if the `d` is positive/zero.
   - if `d` is negative, use $d = d + 2x + 3$ to calculate next `d`.
   - if `d` is positive/zero, use $d = d + 2x - 2y + 5$ to calculate next `d`.

x	y	d	calculate next d
0	8	-7	$(-7) + 2 * (0) + 3 = -4$
1	8	-4	$(-4) + 2 * (1) + 3 = -1$
2	8	-1	$(-1) + 2 * (2) + 3 = 6$
3	7	6	$(6) + 2 * (3) - 2 * (7) + 5 = 3$

x	y	d	calculate next d
4	6	3	$(3) + 2 * (4) - 2 * (6) + 5 = 4$
5	5	4	$(4) + 2 * (5) - 2 * (5) + 5 = 9$

We calculate until `x` and `y` become equal. Now we have all the points for a single octant. To get that octant's other half we can switch the values of x and y.

To get the other three quadrants we can then have them by changing the signs:

- (-x, y) for 2nd Quadrant.
- (-x, -y) for 3rd Quadrant.
- (x, -y) for 4th Quadrant.

1st Q (x,y)	2nd Q (-x,y)	3rd Q (-x,-y)	4th Q (x,-y)
$(0, 8)$	$(-0, 8)$	$(-0, -8)$	$(0, -8)$
$(1, 8)$	$(-1, 8)$	$(-1, -8)$	$(1, -8)$
$(2, 8)$	$(-2, 8)$	$(-2, -8)$	$(2, -8)$
$(3, 7)$	$(-3, 7)$	$(-3, -7)$	$(3, -7)$
$(4, 6)$	$(-4, 6)$	$(-4, -6)$	$(4, -6)$
$(5, 5)$	$(-5, 5)$	$(-5, -5)$	$(5, -5)$
$(6, 4)$	$(-6, 4)$	$(-6, -4)$	$(6, -4)$
$(7, 3)$	$(-7, 3)$	$(-7, -3)$	$(7, -3)$
$(8, 2)$	$(-8, 2)$	$(-8, -2)$	$(8, -2)$
$(8, 1)$	$(-8, 1)$	$(-8, -1)$	$(8, -1)$
$(8, 0)$	$(-8, 0)$	$(-8, -0)$	$(8, -0)$

Ellipse Drawing

We can also draw ellipse using a modified version of the Midpoint algorithm.

However, Ellipse only have 4-way symmetry and **no 8-way symmetry**. So, we calculate an entire Quadrant and calculate the other three from it.

This ellipse equation is used where `rx` and `ry` are the two radii (semi-major and semi-minor axes) of the ellipse.

$$f = (r_y^2 \times x) + (r_x^2 \times y) - (r_y^2 \times r_x^2)$$

```
(Insanely Complicated so steps skipped).
```

Aliasing

A problem in graphics caused when a diagonal/curved shape is drawn.

This is due to the fact that a screen is made up of **square** pixels. It results in the image having jagged edges, distortion or pixelation.

Anti Aliasing

The process of smoothing out lines by using different techniques. This can usually done by

1. Increasing Resolution to have more pixels.
2. **Super Sampling** → Rendering at a higher resolution then down-sampling to a lower one.
3. **Fast Approximate** → Looking at nearby pixels and averaging out the color. Can cause blurry images.
4. **Time Sampling** → Averaging out pixels based on previous frames.

All of these methods do impact performance a little because they all require extra processing.

Transformations

2D Transformations

A change of any kind is called a Transformation.

Suppose we have an object (x, y) where x and y are the coordinates of the points that are in it.

Transformation	Equation	What it does
Translation	$f(x, y) = (x + a, y + b)$	*Moving the object from one point to another.*
Rotation	$f(x, y) = (xcos(\theta) + ysin(\theta), -xsin(\theta) + ycos(\theta))$	*Turning the object on its axis.*
Scale	$f(x, y) = (ax, by)$	*Changing the size of the object.*
Shear (Skew)	$f(x, y) = (x + ay, y + bx)$	*Translating the individual pixel of the image based on an angle.*

Transformation	Equation	What it does
Reflection	$f(x, y) = (\pm x, \pm y)$	*Mirroring the object in one or both axes.*

Rigid Transformation → If a transformation keeps the distance between points the same then its a rigid transformation.
Similarity Transformation → If a transformation keeps the angles of the shape the same then its a similarity transformation.
Affine Transformation → If a transformation keeps the geometry of the object the same then its an affine transformation.

Transformation	Rigid?	Similar?	Affine?
Translation			
Rotation			
Scale		Non-Uniform Uniform	
Shear			
Reflection			

Transformation Matrices

We can represent these transformations through matrices as well.

So our object (x, y) can be represented as a 2×1 matrix:

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

Then any transformation applied onto it is *a new 2×2 matrix that you can multiply*.

- **Scaling**

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + 0y \\ 0x + by \end{bmatrix} => \begin{bmatrix} ax \\ by \end{bmatrix}$$

- **Reflection**

1. Vertical Flip (reflection at y-axis)
2. Horizontal Flip (reflection at x-axis)
3. Vertical + Horizontal Flip (reflection at both axes)

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

- **Shear**

$$\begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ bx + y \end{bmatrix}$$

- **Rotation**

$$\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\,cos\theta + y\,sin\theta \\ -x\,sin\theta + y\,cos\theta \end{bmatrix}$$

  We can't represent **translation** using 2×2 matrices. For that we can use homogenous coordinates.

Homogenous Coordinates

Representing coordinates in 2 dimensions using a 3-point vector where:

$$(x , y , 1)$$

This way we can use a 3×3 matrix for **translation**.

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ 1 \end{bmatrix}$$

We can now use 3×3 matrices for any **Affine Transformation**.

This allows us to chain Transformations for complex things like *Rotation at some random point* or *Reflection at some random line*.

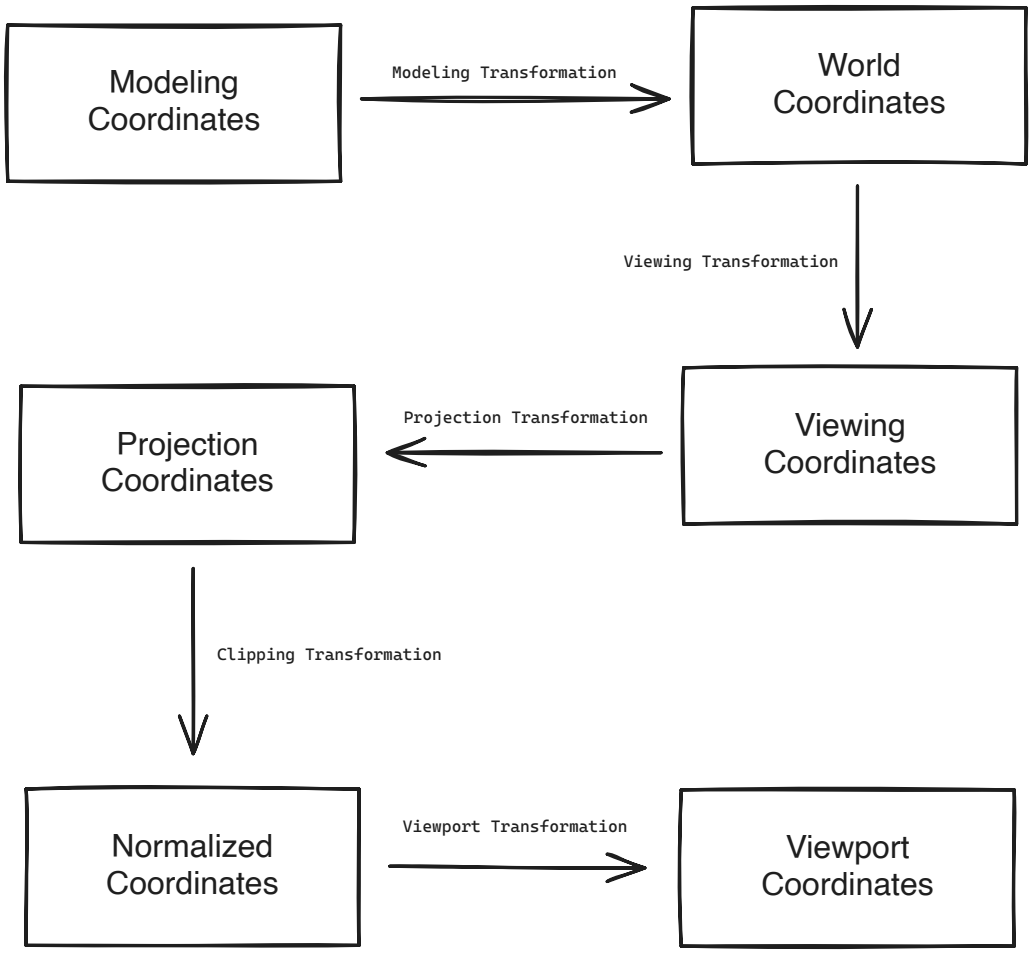
1. Rotation at some random point
    1. `Translation` → take the point to origin.
    2. `Rotation` → apply the rotation.
    3. Undo `Translation`
2. Reflection at some random line
    1. `Translation` → take the line to origin.
    2. `Rotation` → rotate it so that it aligns with x-axis.
    3. `Reflection` → apply the reflection.
    4. Undo `Rotation`
    5. Undo `Translation`

Degrees of Freedom (DOF) → *Number of ways that something can move/rotate in.*

- 1D Objects only have **1** DOF
    - Moving (in x-axis)
- 2D Objects have **3** DOF
    - Moving (in x-axis)
    - Moving (in y-axis)
    - Rotating (between x-y)
- 3D Objects have **6** DOF
    - Moving (in x-axis) → *Sway*
    - Moving (in y-axis) → *Heave*
    - Moving (in z-axis) → *Surge*
    - Rotating (between x-y) → *Roll*
    - Rotating (between y-z) → *Pitch*

- Rotating (between x-z) → *Yaw*

Viewing



1. Every object has its **Modeling Coordinates** at first.
2. They are then converted to **World Coordinates**, when they are loaded as graphics by the graphics pipeline renders.
3. They are then converted to **Viewing Coordinates** which are based on the camera.
4. They are then converted to **Normalized Coordinates** by applying transformation for translating to origin, scaling up/down for device and clipping.
5. They are then converted to **Device Viewport Coordinates** which are based on the end-device itself.

World Coordinates -> Viewing Coordinates

$$\begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

World Coordinates -> Viewport Coordinates

```
(Insanely Complicated so skipped).
```

Projections

A **viewing plane** is set up parallel to x,y and perpendicular to z.

Projections are formed when **projection lines** intersect with this viewing plane. These lines start at the **center of projection** of that object and go through the viewing plane. If the center of projection is at "infinity" then we get a *parallel projection* otherwise we get a *perspective projection*.

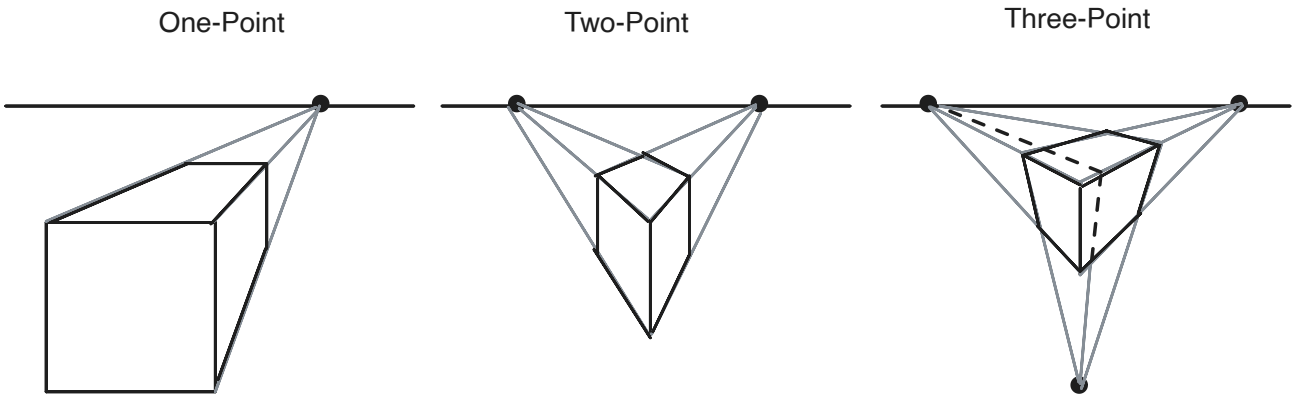
Parallel Projection	**Perspective Projection**
Center of Projection $= \infty$	Center of Projection $\neq \infty$
Preserves relative sizes/proportions.	Doesn't preserve relative sizes/proportions.
Doesn't produce realistic images.	**Produces realistic images.**

Types of Projections

Parallel Projection

- Orthographic → The projection lines are perpendicular to the surface. (Top, Side, Front)
- Oblique → The projection lines are not perpendicular to the surface and have an angle.
  - Cavalier → The depth is the same. This preserves length.
  - Cabinet → The depth is half. This looks more realistic.

  Perspective Projection
- One Point, Two Point, Three Point



Windowing and Clipping

Since, objects are specified in their **World Coordinates** and we only see the ones that are within the **Window** so everything else that cannot be seen/is outside the window is "clipped".

Line Clipping

One way to do this is to clip all the objects whose endpoints are outside the window but this introduces a problem for objects that have:

1. Even a single endpoint outside are clipped.
2. Both endpoints outside are not clipped.

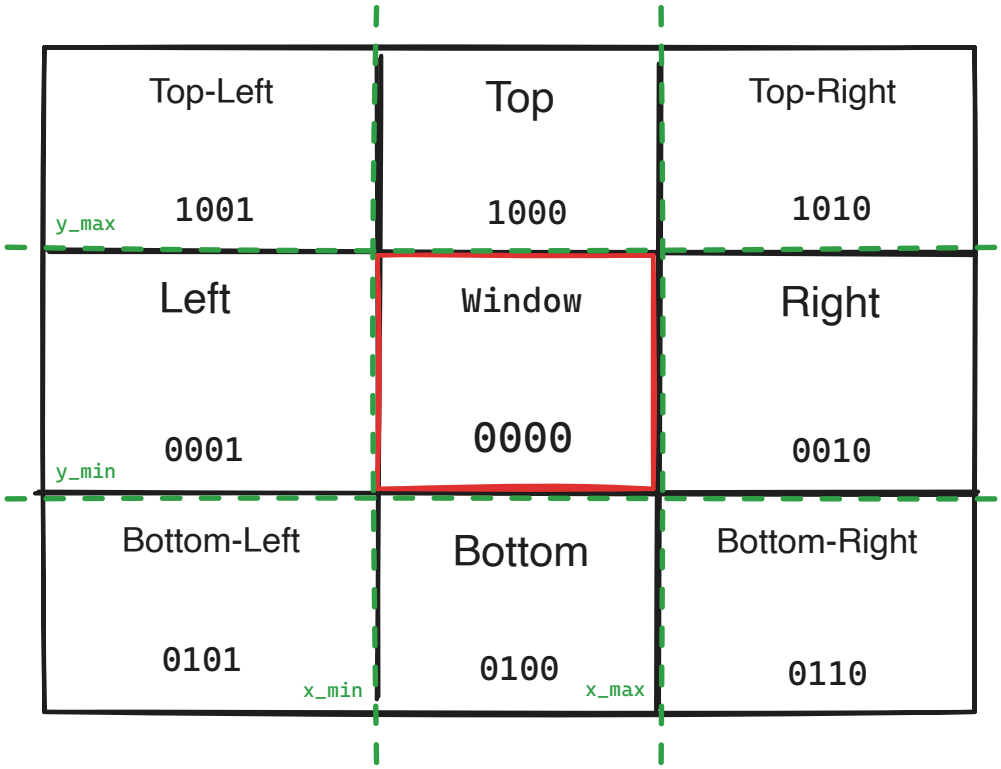
Brute Force Line Clipping

This method uses the line equation at the endpoints to find the intersection as to where to clip from. This method is **extremely slow** because a single scene can have many manyyyy lines to clip.

Cohen-Sutherland Line Clipping

This is a very efficient line-clipping algorithm. It is

1. Divide the screen into 9 regions. The center region is the **window** where objects would be visible and the remaining 8 regions are where they wouldn't be visible.
2. All regions are given a *4-digit* binary code.



3. Each digit of the code represents a boundary/direction → (top, bottom, right, left).
4. This code is used to identify if a point is inside the window or not. Give this code to both endpoints of the line.
5. Perform the OR operation between the two endpoints.
    1. **if** $P_1 \, OR \, P_2 = 0000$, then our line is completely inside and is saved.

2. **else** Perform the AND operation between the two endpoints
    1. **if** $P_1\ AND\ P_2 \neq 0000$, then our line is completely outside and is deleted.
    2. **else if** $P_1\ AND\ P_2 = 0000$, then our line is *partially inside* and we apply the next steps to find its intersections to partially clip it.
6. Pick the endpoint that is outside the window. We will then find a point $\bar{P_2}$ which will be just at the intersection boundary.
7. This equation is used to find m. → $m = y_2 - y_1\ /\ x_2 - x_1$
8. Then to get x, y values of $\bar{P_2}$ like this.
    1. if the line crosses through the **top** boundary.

$$x = x_1 + (\ y_{max} - y_1\ )/\ m$$
$$y = y_{max}$$

    2. if the line crosses through the **bottom** boundary.

$$x = x_1 + (\ y_{min} - y_1\ )/\ m$$
$$y = y_{min}$$

    3. if the line crosses though the **right** boundary.

$$x = x_{max}$$
$$y = y_1 + (\ x_{max} - x_1\ ) \times m$$

    4. if the line crosses though the **left** boundary.

$$x = x_{min}$$
$$y = y_1 + (\ x_{min} - x_1\ ) \times m$$

9. Update the region code of the point to 0000 and delete everything after this point.

Liang-Barsky Line Clipping

It is an even faster algorithm (but more complex) compared to the Cohen-Sutherland Algorithm.

1. Originally we have these equations for the values of x and y. (where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$).

$$x = x_1 + u \cdot \Delta x, \quad y = y_1 + u \cdot \Delta y$$

2. They need to be between the boundary values. (where u is between 0 and 1).

$$x_{min} \leq x_1 + u \cdot \Delta x \leq x_{max}\ , \quad y_{min} \leq y_1 + u \cdot \Delta y \leq y_{max}$$

3. We can get four equations from the two above.

$$x_{min} \leq x_1 + u \cdot \Delta x\ , \quad x_{max} \geq x_1 + u \cdot \Delta x$$
$$y_{min} \leq y_1 + u \cdot \Delta y\ , \quad y_{max} \geq y_1 + u \cdot \Delta y$$

4. Rearrange them to get. This is of the form $u \cdot p_k \leq q_k$.

$$-u \cdot \Delta x \leq x_1 - x_{min}\ , \quad u \cdot \Delta x \leq x_{max} - x_1$$
$$-u \cdot \Delta y \leq y_1 - y_{min}\ , \quad u \cdot \Delta y \leq y_{max} - y_1$$

5. If
    1. $p_k = 0$ → line is parallel. (if then $q_k < 0$ → line is outside).
    2. $p_k < 0$ → make a table for where P<0 and P>0.

$P_k < 0$	$P_k > 0$
$t_1 = max(0, \text{All the Ps that are less than } 0)$	$t_2 = min(1, \text{All the Ps that are more than } 0)$

6. if
    1. $t1 > t2$ → line is outside.
    2. $t2 > t1$ → Find x and y using these equations.

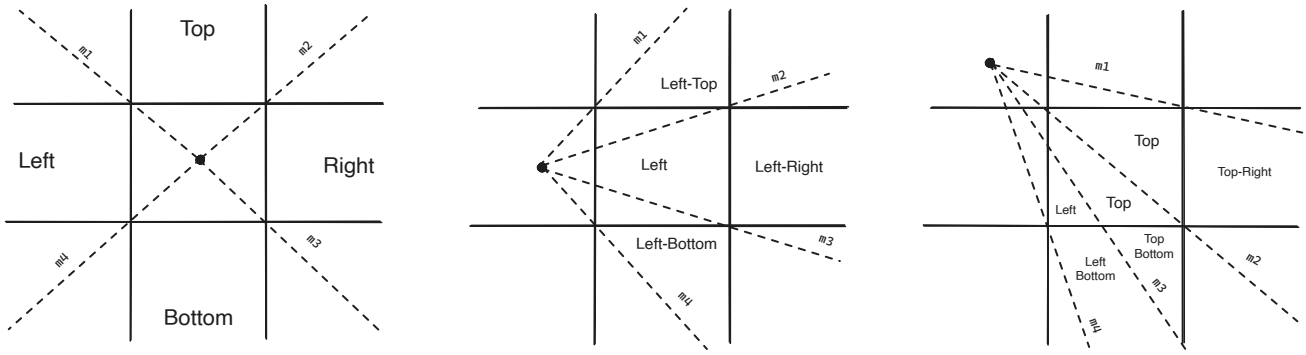
$$x = x_1 + t_1 \cdot \Delta x$$
$$y = y_1 + t_1 \cdot \Delta y$$

7. The (x,y) we get is the intersection point and the answer.

Nicholl-Lee-Nicholl Line Clipping

It is fastest and the most complicated algorithm. It only works well for 2D and it avoids clipping the same line multiple times.

We first make sure that our point P_1 is in one of these three positions and if it's not then we apply transformations to bring it there.



Then we find the slope between point P_1 and the second point P_2 . Also find the slopes of the four bounding lines $\mathrm{m1}, \mathrm{m2}, \mathrm{m3}, \mathrm{m4}$.

If our line with slope m comes out to be in some region that we haven't labeled then it is discarded. We also find the two bounding lines its slope lies in between.

Area Clipping

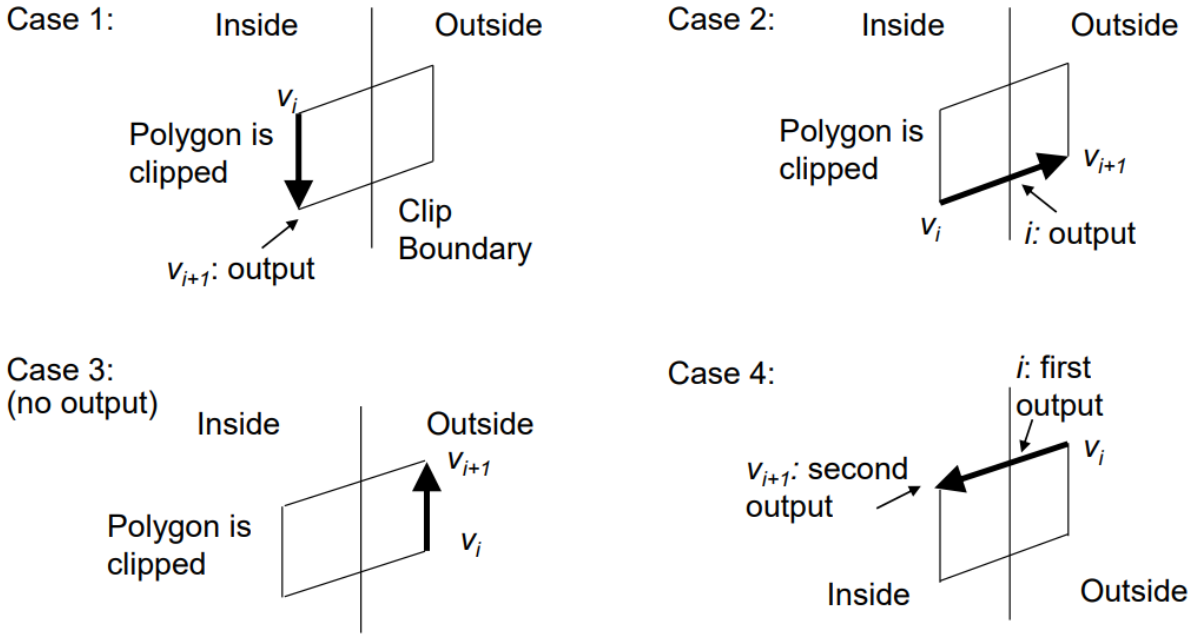
Clipping area of polygons is much much harder compared to clipping lines. This type of clipping is difficult because of us knowing nothing about the end result and how many sides it will have.

Sutherland-Hodgeman Polygon Clipping

This is a divide-and-conquer algorithm that clips by doing each boundary, one by one $(\mathrm{L}, \mathrm{R}, \mathrm{T}, \mathrm{B})$.

These 4 cases are applied one by one, while going through each boundary clipping.

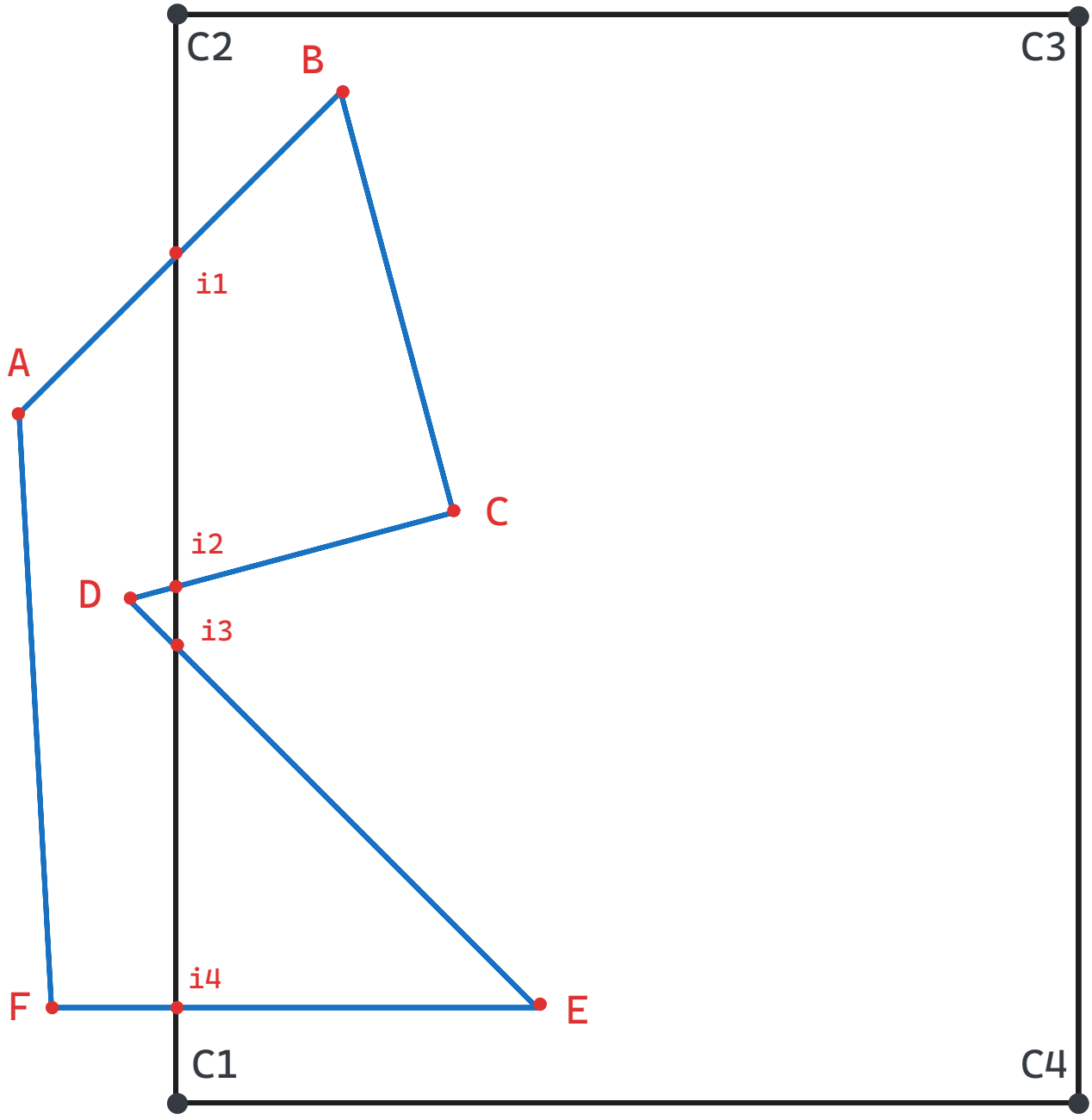
1. **Case 1**: Inside v_i → Inside v_{i+1} (output v_{i+1})
2. **Case 2**: Inside v_i → Outside v_{i+1} (output the intersection point i)
3. **Case 3**: Outside v_i → Inside v_{i+1} (output the intersection point i and v_{i+1})
4. **Case 4**: Outside v_i → Outside v_{i+1} (no output)



Repeatedly apply these 4 cases on every single edge.Sutherland-Hodgeman fails when dealing with concave polygons and adds an extra line in between.

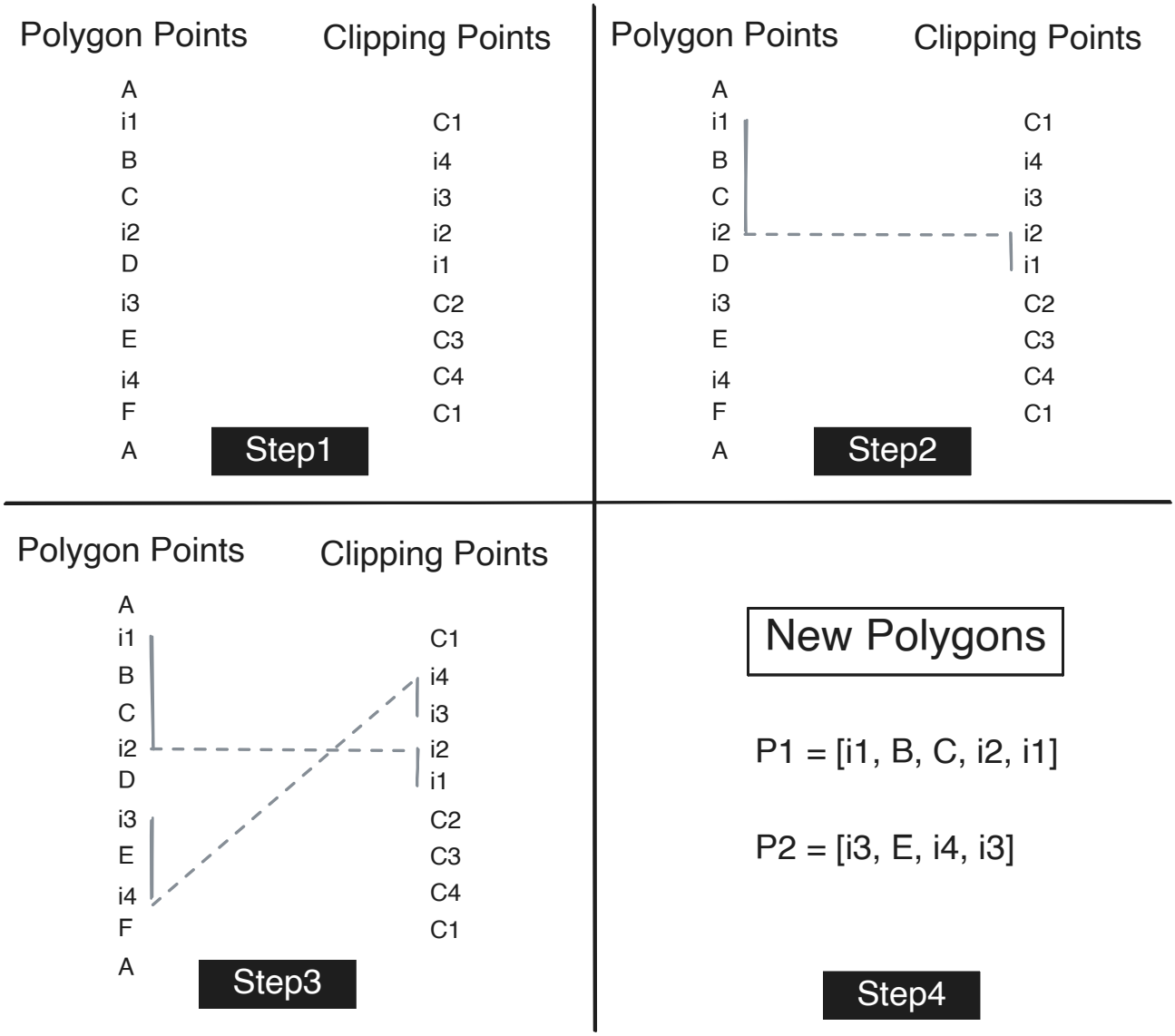
Weiler-Atherton Polygon Clipping

This is a polygon clipping algorithm where we don't get extra lines while dealing with concave polygons. This is done by either going clockwise or anticlockwise through every single pair of points.



Here, we have a polygon $ABCDEF$ inside the window $C_1C_2C_3C_4$ with intersections $i_1,\ i_2,\ i_3,\ i_4$.

1. First, exhaustively write all the points of the polygon and the clipping window.
2. Then, draw a line across the points of the first polygon part that remains inside the clipping window.
3. Do the same for the second polygon part.
4. Write the new polygon points separately.


Polygon Points
Clipping Points
A
i1
B
C
i2
D
i3
E
i4
F
A
C1
i4
i3
i2
i1
C2
C3
C4
C1
Step1
Polygon Points
Clipping Points
A
i1
B
C
i2
D
i3
E
i4
F
A
C1
i4
i3
i2
i1
C2
C3
C4
C1
Step2
Polygon Points
Clipping Points
A
i1
B
C
i2
D
i3
E
i4
F
A
C1
i4
i3
i2
i1
C2
C3
C4
C1
Step3
New Polygons
P1 = [i1, B, C, i2, i1]
P2 = [i3, E, i4, i3]
Step4

Curve Clipping

Curves can also be clipped using similar algorithms like

1. We can approximate small straight lines out of the curves and then apply one of the polygon clipping alogrithms
2. Use non-linear equations.
3. Create a bounding rectangle around the curve and use that to clip the curve.

Text Clipping

We can clip text by using one of these three aproaches:

- **All or None String Clipping** → Create a bounding rectangle around the entire text and just check if all points are inside the window. If even a single point is outside then the entire string is discarded.
- **All or None Character Clipping** → Create a bounding rectangle around every single character of the text and then check individually for each character for if all points of its bounding rectangle are inside the window. If even a single point is outside then that character is discarded.
- **Individual Character Clipping** → Check individual pixels of the text to be able to partially clip the parts of the characters which are outside and keep the ones inside. This approach give the most precise results but requires heavy processing.

3D Clipping

Clipping 3D shapes requires us to extend the Cohen-Sutherland Line Algorithm by making the region codes 6-bits $\text{Front}, \text{Back}, \text{Top}, \text{Bottom}, \text{Right}, \text{Left}$ instead of 4-bits. We can also extend the Sutherland-Hodgeman Algorithm as well by just running it 6 times instead of 4 (to cover each boundary).

The view volume is a box for orthographic projections and a frustrum for perspective projections.

Visibility and Culling

Visibility is when an object is visible

Culling is the process of removing sides of objects that are not visible to the camera. This is a very challenging thing to achieve but it makes the video/game run much much smoother.

Hidden Surface Removal

There are many "hidden surface removal" algorithms and all of them fall into one of these two classes:

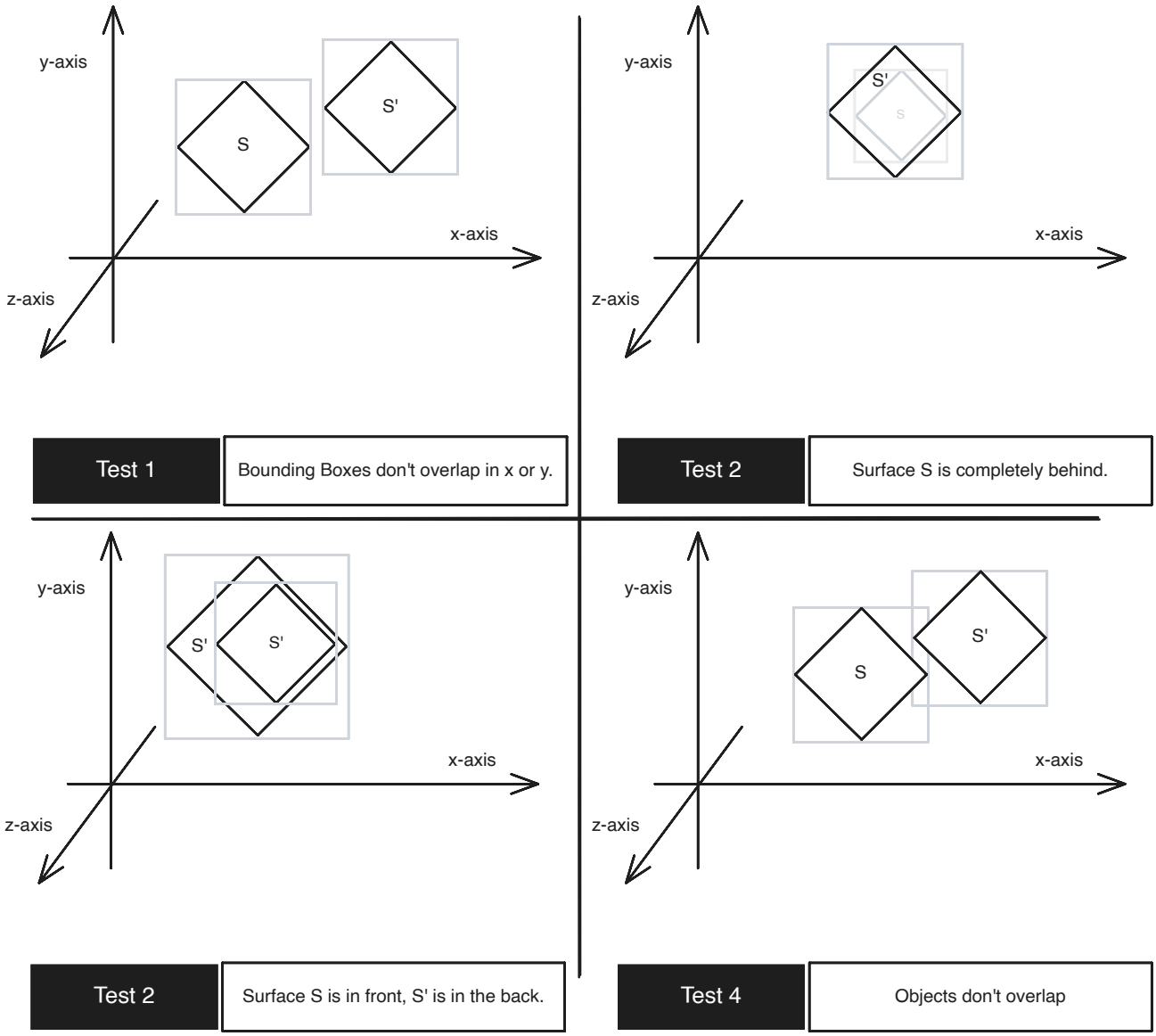
- **Object Precision** → Computations are done on the primitives (basic shapes).
- **Image Precision** → Computations are done at the pixel level.

Any viewing space works as they all maintain depth. This mean we can use World or View or Window coordinate space. There are however many problems to tackle like *Efficiency*, *Accuracy*, *Complexity* and *Performance*.

Painter's Algorithm (Depth Sorting)

It is called a depth-sorting algorithm. We choose a specific order and then draw surfaces in that order. The usual order is "depth sorting", in this way we render "nearer" polygons after the "farther" polygons.

Suppose we have two objects; the original surface → S and and overlapping surface → S'. We perform these 4 tests to decide the ordering:



Back-Face Culling

This is an algorithm used before applying any viewing algorithms. It is used to draw only those surfaces which will face the camera. The objects on the back side are not visible. This method will remove $\geq 50\%$ of the polygons from the scene.

It uses **Object Precision** and we identify the backfaces in it.

1. Normal Vectors (numbered N_1, N_2, etc...) project outwards from the faces.
2. A vector p projects outwards from the camera/projectionplane.
3. If $p \times N_i < 0$, then it is considered frontface and isn't culled.
4. If $p \times N_i > 0$, then it is considered backface and is culled.

Warnock's Area Subdivision

It is an image-precision algorithm that uses divide-and-conquer. We take a region and then find out which polygons are infront of the region in what order.

There are 3 cases in this algorithm for finding what is in front:

1. a polygon is completely in front of everything else in that region.
2. no polygons overlap with the region.
3. only one polygon overlaps or completely surrounds the region.

If none of these cases apply then just subdivide the region into smaller regions and test for those cases again.

BSP Trees

```
(Skipped)
```

Animation

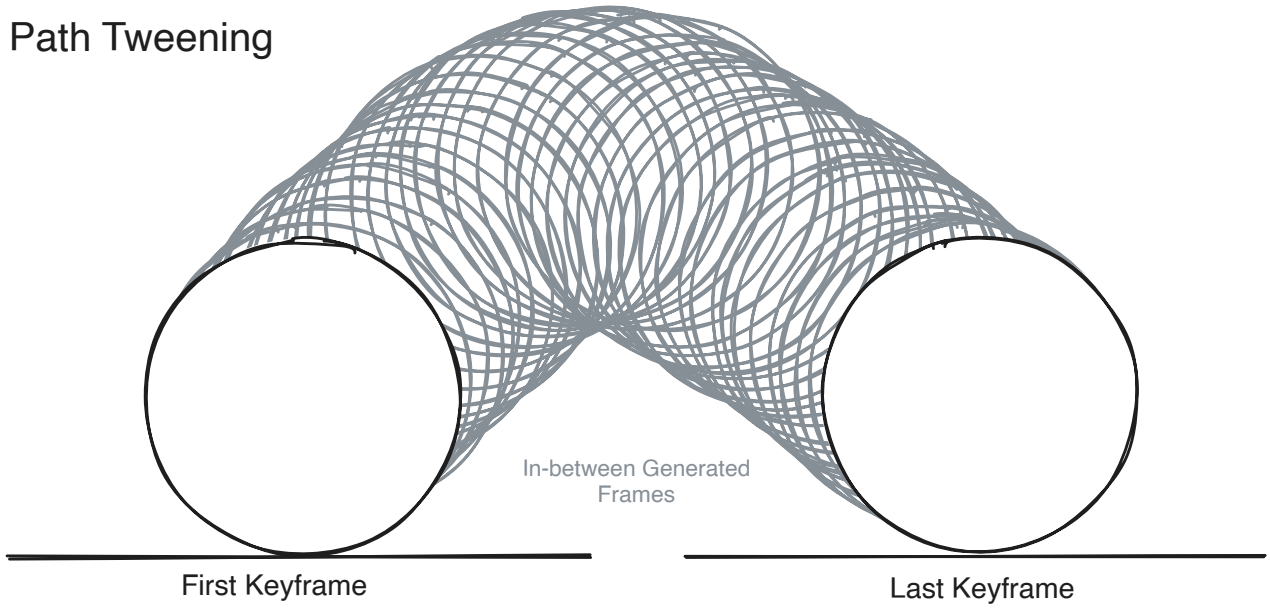
An animation is the act of making something move/come alive. It is an object moving/coming in and out in the screen.

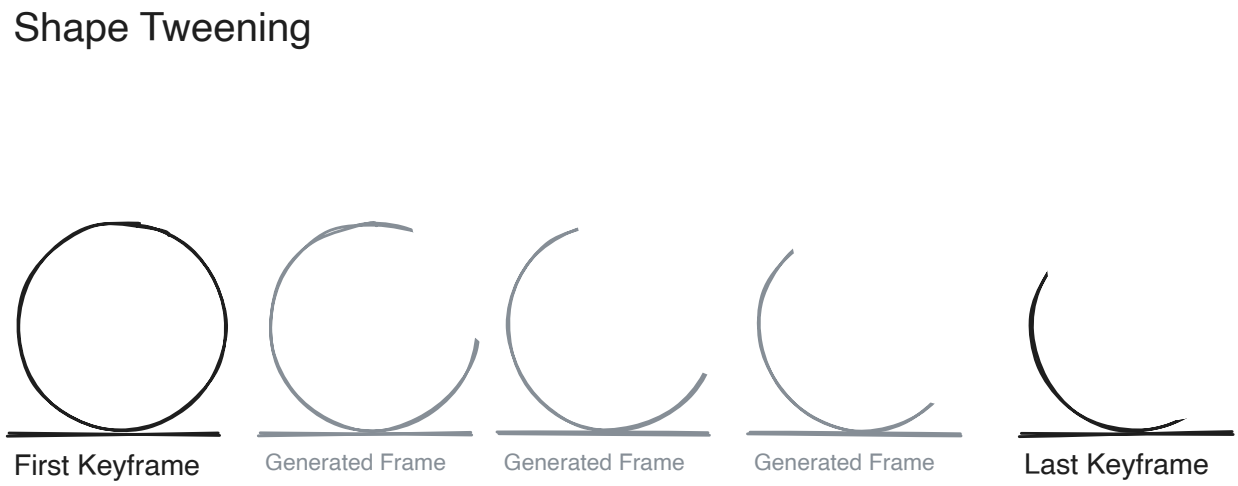
It is used for many purposes like Art, Games, Visualizations, Aesthics and Storytelling. **Transitions** are simple animations that happen between scenes.

File Formats → Windows Media (.avi), Apple Media (.mov), Motion Graphics (.mpeg), Flash (.swf), Animated .gif etc.

Cel Animation

Old Form of Animation where clear plastic sheets were used to draw frames. 24 Frames per 1 second was the FPS (frames per second). This meant $24 \times 60 = 1440$ frames in total to draw for a 1 minute animation. **Keyframes** are the first and last frame of an animation sequence. **Tweening** is the process of generating in-between frames between Keyframes to make animations. This can be for **Movement** (Path Tweening) or **Morphing** (Shape Tweening).





Computer Generated Animation

This is a different type of animation where 3D objects are created in a modeling software which are then moved and transformed to create animation. This way animators don't have to draw every single frame and the entire processing lies on the computer itself.

Kinematics is the study of how an object with joints move. In Computer Generated Animation, objects have joints which can be used to move individuals parts of that object.

Illumination

Illumination in computer graphics refers to the process of simulating how light interacts with objects in a scene. Three types of objects are focused on when talking about illumination. **Light source**, **Surface**, **Camera**.

Light Sources

This is the origin/location of the incoming light in the scene.

Point Source → All light rays come from a single point.
Parallel Source → All light rays are parallel to each other.
Distributed Source → All light rays come from a small area (this is used to emulate the effect of a real life light source like bulbs).

Illumination Models

These are fast mathematical methods of lighting up a scene.

Ambient Illumination → Creates equal/constant lighting on every object. It is very simple and lightweight but not realistic.
Diffuse Illumination → Creates the effect of scattered light. The brightness depends on the angle of the light rays hitting the object.
Specular Reflection → Creates the effect of shiny/glossy reflective surfaces. It is calculate different depending on the location of the camera.

$$\text{Ambient Reflected Intensity} = K_A \times I_A$$
$$\text{Diffuse Reflected Intensity} = K_D \times I_P \times N \times L$$
$$\text{Specular Reflected Intensity} = K_S \times I_P \times R \times V$$

All three combined: $K_A\, I_A + I_P\, (\, K_D\, (\, N \times L\,) + K_S\, (\, R \times V\,)\, n\,)$

Rendering Methods

These are methods used to color objects with.

1. **Flat Shading** → One color is calculated for every polygon then that polygon is filled with this color. *Least Realistic*.
2. **Gouraud Shading** → One color is calculated for every vertex of every polygon then that polygon is blended with the vertex colors.
3. **Phong Shading** → One color is calculated for every single pixel of the polygon, based on the normal vector. *Most Realistic*.