Module 1

**1.1**

**Intro To CG -**

[**https://geeksforgeeks.org/introduction-to-computer-graphics/**](https://geeksforgeeks.org/introduction-to-computer-graphics/)

**Application of CG-**

[**https://www.geeksforgeeks.org/applications-of-computer-graphics/**](https://www.geeksforgeeks.org/applications-of-computer-graphics/)

**Display Processor -**

[**https://www.geeksforgeeks.org/display-processor-in-computer-graphics/?ref=lbp**](https://www.geeksforgeeks.org/display-processor-in-computer-graphics/?ref=lbp)

**Raster Scan -**

[**https://www.geeksforgeeks.org/raster-scan-displays/?ref=lbp**](https://www.geeksforgeeks.org/raster-scan-displays/?ref=lbp)

**Random / Vector Scan -**

[**https://www.geeksforgeeks.org/random-scan-display/?ref=lbp**](https://www.geeksforgeeks.org/random-scan-display/?ref=lbp)

[**https://www.geeksforgeeks.org/vector-graphics-in-computer-graphics/**](https://www.geeksforgeeks.org/vector-graphics-in-computer-graphics/)

**1.2**

**Line Drawing Algorithm-**

**DDA -** [**https://www.gatevidyalay.com/dda-algorithm-line-drawing-algorithms/**](https://www.gatevidyalay.com/dda-algorithm-line-drawing-algorithms/)

**Bresenham’s -** [**https://www.gatevidyalay.com/bresenham-line-drawing-algorithm/**](https://www.gatevidyalay.com/bresenham-line-drawing-algorithm/)

**MidPoint -** [**https://www.gatevidyalay.com/mid-point-line-drawing-algorithm/**](https://www.gatevidyalay.com/mid-point-line-drawing-algorithm/)

**Circle Drawing Algorithm-**

**Bresenham’s -** [**https://www.gatevidyalay.com/bresenham-circle-drawing-algorithm/**](https://www.gatevidyalay.com/bresenham-circle-drawing-algorithm/)

**Midpoint -** [**https://www.gatevidyalay.com/mid-point-circle-drawing-algorithm/**](https://www.gatevidyalay.com/mid-point-circle-drawing-algorithm/)

2D Viewing Pipeline

### **OpenGL 2D Viewing Pipeline**

The OpenGL 2D viewing pipeline involves transforming 2D objects from their original coordinate system into a viewable space on the screen. This process includes several stages that involve transformations from world coordinates to normalized device coordinates, and finally to screen coordinates.

#### **Stages of the 2D Viewing Pipeline:**

1. **Model Coordinate Transformation**:
   * The object is initially defined in model coordinates.
   * **Transformation to World Coordinates**: Here, the object may be translated, rotated, or scaled to fit the world's coordinate system.
2. **World Coordinate Transformation**:
   * Objects are defined within a larger world coordinate space.
   * **Transformation to Viewing Coordinates**: Objects are transformed based on the viewport, defining which part of the world will be visible in the final rendering.
3. **Window-to-Viewport Mapping**:
   * **Clipping and Normalization**: Objects that fall outside the viewing window are clipped.
   * **Viewport Transformation**: The visible region is mapped from the viewing coordinates to device coordinates, typically normalized between -1 and 1.
4. **Device Coordinate Transformation**:
   * In this final step, the normalized coordinates are converted to screen (pixel) coordinates.

3D Viewing Pipeline

The OpenGL 3D viewing pipeline is more complex than the 2D pipeline because it includes depth and perspective transformations. It involves mapping 3D objects from their object coordinates through several transformations until they are displayed on a 2D screen.

#### **Stages of the 3D Viewing Pipeline:**

1. **Model Transformation**:
   * The 3D model is first defined in its local coordinates, referred to as model coordinates.
   * **Transformation to World Coordinates**: Similar to the 2D pipeline, objects are transformed from model to world coordinates through translation, scaling, and rotation.
2. **View Transformation**:
   * **Transformation to Eye (Camera) Coordinates**: This involves setting up a "camera" or view point in the world coordinates. The world coordinates are transformed into camera coordinates, allowing us to view objects from the camera's perspective.
3. **Projection Transformation**:
   * In this stage, the 3D scene is transformed into a 2D view.
   * **Perspective Projection**: Applies a perspective transformation for realistic depth, where objects farther from the camera appear smaller.
   * **Orthographic Projection**: Applies an orthographic transformation, where objects maintain their relative size regardless of distance.
4. **Clipping and Normalization**:
   * The transformed objects are clipped to fit within a defined view volume (typically a frustum for perspective projection or a box for orthographic projection).
   * **Normalization**: After clipping, coordinates are mapped to normalized device coordinates, generally between -1 and 1 in each dimension.
5. **Viewport Transformation**:
   * **Device Coordinates**: Finally, the normalized coordinates are mapped to device coordinates (pixel coordinates on the screen).

Affine Function

### **Affine Function**

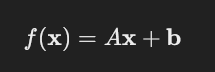
Affine functions represent a linear transformation followed by a translation, preserving parallelism and straight lines. These transformations are pivotal in graphics for manipulating shapes and objects in space without altering their fundamental structure.

#### **Properties:**

* **Preserves collinearity**: Points on a line before transformation remain on a line afterward.
* **Preserves ratios**: Divisions of line segments (e.g., the midpoint) are preserved.

#### **Mathematical Representation:**

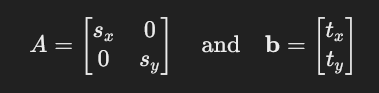
An affine function in 2D or higher dimensions can be written as:



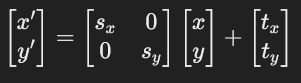
where:

* A is a matrix that scales, rotates, or skews the vector x,
* b is a translation vector shifting all points by a constant amount.

For example, applying a scaling transformation with scale factors sx​ and sy​ and translating by tx​ and ty would use:



This transforms a point (x,y) to a new location:



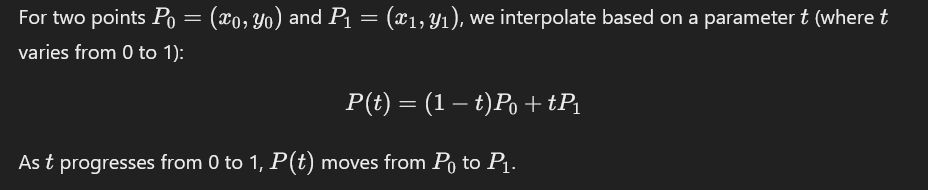
#### **Applications in Graphics:**

Affine transformations are essential for rendering scenes in 2D and 3D graphics, allowing for translations, rotations, scaling, and shearing of objects without distorting them. They're implemented extensively in vector graphics software like Adobe Illustrator and animation software.

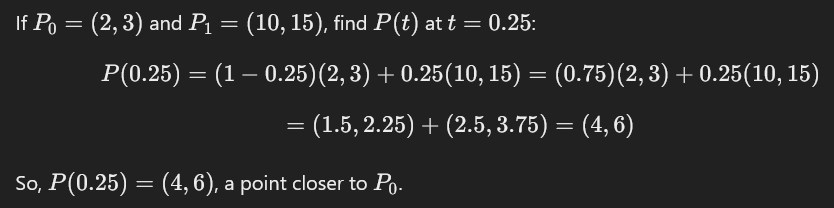
### **Linear Interpolation of Two Points (Lerp)**

Linear interpolation (lerp) is used to find intermediate values between two endpoints in animations, color gradients, and position tracking.

#### **Formula and Derivation:**



#### **Example:**



#### **Uses:**

Linear interpolation is widely used in computer graphics for:

* **Animating movement** between two positions.
* **Transitioning colors** in gradients.
* **Smoothing camera motions** in 3D scenes.

Tweening

### **Tweening (for Art and Animation)**

Tweening, or in-betweening, is a technique in animation to generate frames that transition from one "key frame" to another. It reduces manual workload and enhances smoothness, especially for complex animations.

#### **Types of Tweening:**

1. **Linear Tweening**: Directly interpolates between frames. This is simple but may appear mechanical.
2. **Ease-In and Ease-Out**: Gradual acceleration at the beginning and deceleration at the end, often modeled with quadratic or cubic equations.
3. **Custom Tweening Curves**: Allows animators to define custom timing curves for nuanced effects.

#### **Example:**

In a bouncing ball animation, keyframes might represent the highest point of each bounce, and tweening frames depict the ball moving smoothly along the arc between each bounce.

#### **Role in Modern Animation:**

Tweening is key in digital animation software (e.g., Adobe Animate, Blender). It enables realistic motion and makes animation software more efficient and accessible.

### **Application of Tweening**

Tweening has applications beyond just traditional character animation. It’s essential for creating polished, dynamic visuals across various fields:

* **UI/UX Design**: Tweening makes transitions in applications smoother, providing visual continuity (e.g., zooms, fades, slides).
* **3D Modeling and Simulation**: Tweening animates camera movements or object rotations.
* **Game Development**: It helps create fluid character motions, object transformations, and scene transitions.
* **Infographics and Data Visualizations**: Tweening adds movement to data transitions, making changes between data states more intuitive for viewers.

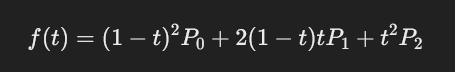
### 

### **Quadratic and Cubic Tweening**

Non-linear tweening adds a dynamic, natural feel to animations. By changing the speed of the transition over time, animations can feel more fluid.

#### **Quadratic Tweening:**

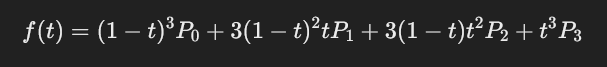
This form of tweening uses a quadratic polynomial to ease-in or ease-out an animation:



* **Ease-In**: Starts slow and speeds up.
* **Ease-Out**: Starts fast and slows down.

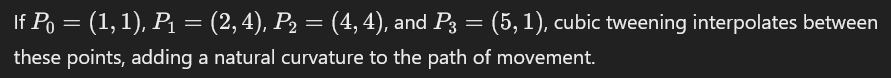
#### **Cubic Tweening:**

Using a cubic polynomial curve allows more complex easing:



* Common for simulating inertia and acceleration, allowing for smooth, customizable animations.

#### **Example:**

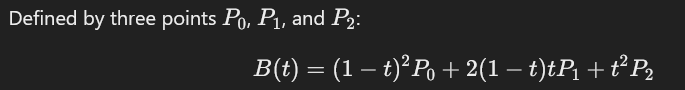


Bezier Curve

### **Bezier Curve**

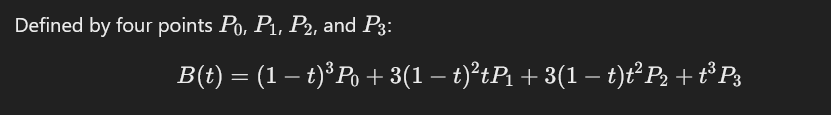
Bezier curves provide a powerful tool for creating smooth, scalable curves. They are especially useful in graphics, fonts, and animation paths.

#### **Quadratic Bezier Curve:**



This curve provides a single "bend" between two endpoints and a control point that dictates the curve’s direction.

#### **Cubic Bezier Curve:**

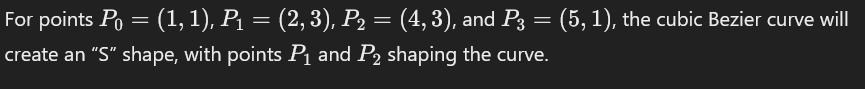


This curve has two control points, allowing more complex shapes.

#### **Applications of Bezier Curves:**

* **Vector Graphics**: Essential for drawing smooth, scalable paths.
* **Font Design**: Used in typeface design for shaping letters.
* **Animation Paths**: Define the trajectory of objects in animation, such as camera paths or character motions.
* **Path-based Animation**: Used in tools like Adobe After Effects to create complex animations along user-defined paths.

#### **Example:**



**Refer -** <https://www.gatevidyalay.com/bezier-curve-in-computer-graphics-examples/>

Types of Curves

In computer graphics, curves are essential for modeling smooth shapes and paths. Different types of curves provide various levels of control and flexibility for shaping objects, animating paths, and designing smooth transitions. Here’s an overview of common types of curves:

### **1. Line or Linear Curve**

* **Definition**: The simplest type of curve is a straight line between two points.
* **Equation**: Given two points P0=(x0,y0) and P1​=(x1​,y1​), the linear interpolation of these points can be written as:



* **Applications**: Used in linear interpolation (lerp), path animation, and simple motion paths.

### **2. Polynomial Curve**

* **Definition**: Polynomial curves are defined by polynomial equations and can take on various shapes depending on the degree of the polynomial.
* **Equation**: A general polynomial curve in 2D is represented as: 
* **Applications**: Used in basic modeling but can become unstable for higher degrees; primarily used for fitting or approximating simpler shapes.

### **3. Bezier Curve**

* **Definition**: A widely used curve in computer graphics, a Bezier curve is defined by a set of control points. It provides a smooth path that is intuitively controlled.
* **Types**:
  + **Linear Bezier** (2 points): Essentially a straight line.
  + **Quadratic Bezier** (3 points): Curves with one control point.
  + **Cubic Bezier** (4 points): Allows more complex shapes with two control points.
* **Equation (Cubic Bezier)**:



* **Applications**: Used in vector graphics, font design, animation, and path planning.

### **4. B-Spline Curve (Basis Spline)**

* **Definition**: B-Splines are piecewise-defined curves that provide more flexibility and control over the curve's shape by adjusting control points, even locally.
* **Types**:
  + **Uniform B-Splines**: Control points are evenly spaced along the curve.
  + **Non-Uniform B-Splines (NURBS)**: Control points are not evenly spaced, allowing more complex and precise shapes.
* **Properties**:
  + Local control: Moving a control point affects only a part of the curve.
  + Defined by a "knot vector" that determines the spacing of control points.
* **Applications**: Ideal for CAD, 3D modeling, and any application needing smooth, editable curves.

### **5. Parametric Curves**

* **Definition**: Parametric curves represent points along a curve using parameterized functions for each axis. These curves are not restricted to polynomial forms.
* **Example**: For a 2D parametric curve:



* **Applications**: Useful in creating paths with specific shapes, especially in physics simulations, engineering applications, and animation.

### **6. Arc and Circular Curves**

* **Definition**: Arcs and circular curves are defined by a center point, a radius, and start/end angles. They are essential for modeling round shapes and rotations.
* **Equation (Circle)**:

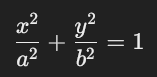


where (xc,yc) is the center and rrr is the radius.

* **Applications**: Used for creating circular shapes, arcs, and rotational paths in 2D and 3D modeling.

### **7. Elliptical and Conic Curves**

* **Definition**: Elliptical and conic curves include ellipses, parabolas, and hyperbolas. These curves have specific geometrical definitions and are essential for realistic modeling.
* **Equation (Ellipse)**:



where a and b are the ellipse's semi-major and semi-minor axes.

* **Applications**: Useful in engineering, physics simulations, and graphics applications that require precise modeling of curved paths.

Types of Projections:

Visible surface detection (also known as **hidden surface removal**) is essential in 3D graphics to ensure that only the surfaces visible to the viewer are rendered, while hidden surfaces are excluded from the final image. This process optimizes rendering and improves realism by correctly displaying object occlusions.

**Classification of Visible Surface Detection Algorithms**

Visible surface detection methods can be categorized based on the approach they take to determine visibility:

1. **Image-Space Methods**: These algorithms determine visibility pixel-by-pixel on the image plane. They analyze each pixel to determine which object is closest to the viewer along a line of sight. Examples include the depth buffer (z-buffer) method and ray casting.
2. **Object-Space Methods**: These methods work by comparing objects directly in the 3D scene to determine which parts should be visible. They assess entire surfaces or objects in relation to each other. Examples include the painter’s algorithm and binary space partitioning (BSP) trees.
3. **Hybrid Methods**: These combine aspects of both image-space and object-space methods to optimize the visibility process. Some algorithms may initially identify which objects are visible and then use a depth-based technique to refine visibility at the pixel level.

**Depth Buffer Method (Z-Buffering)**

The **depth buffer** or **z-buffer** method is an **image-space technique** widely used in real-time rendering, including OpenGL, due to its simplicity and efficiency. Here's how it works:

1. **Depth Buffer Setup**: A depth buffer, or z-buffer, is created alongside the color buffer (where the color of each pixel is stored). The depth buffer stores depth values (z-coordinates) for each pixel, initially set to a maximum value (often the farthest viewable distance).
2. **Rendering Process**: As each object in the scene is rendered, the depth (z-coordinate) of each pixel is compared to the corresponding value in the depth buffer:
   * If the new pixel is closer to the viewer (has a lower z-value), it is rendered, and its depth is recorded in the depth buffer.
   * If the new pixel is farther than the existing depth buffer value, it is discarded.
3. **Output Image**: Once all objects are processed, the color buffer contains the final image with only the closest (visible) surfaces rendered, while the depth buffer holds depth information for each pixel in the scene.

**Advantages of the Depth Buffer Method**

* **Simple to Implement**: The algorithm is straightforward, only requiring a depth comparison at each pixel.
* **Fast and Suitable for Hardware Acceleration**: The z-buffer method is efficient and well-supported by hardware, making it ideal for real-time applications.
* **Handles Complex Scenes and Overlapping Objects**: Z-buffering works with scenes of arbitrary complexity and does not require sorting of objects by depth.

**OpenGL Codes-**  
[**https://drive.google.com/drive/folders/1Z97lztG6E9ISaMW0BEBX-P2KhFB6oUG9?usp=sharing**](https://drive.google.com/drive/folders/1Z97lztG6E9ISaMW0BEBX-P2KhFB6oUG9?usp=sharing)