

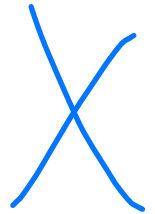
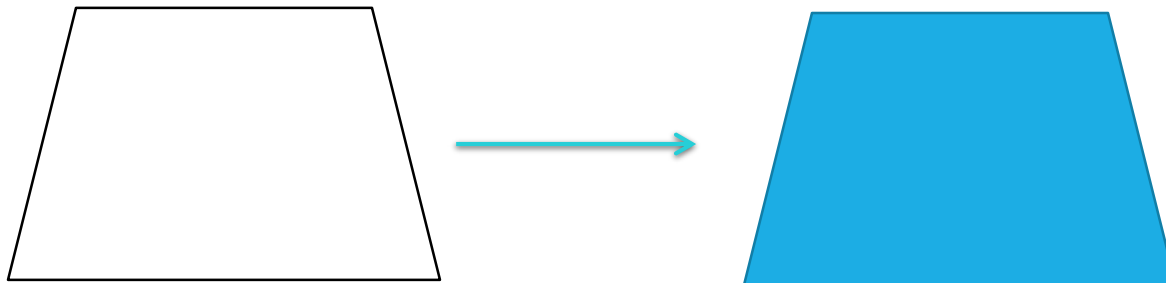


# BLENDING, ANTIALIASING, POLYGON FILLING

Tara Qadr  
2024-2025  
3rd Stage  
Computer department  
College of science  
University of sulaimani

# Polygon Filling

- For filling polygons with particular colors, you need to determine the pixels falling on the border of the polygon and those which fall inside the polygon, can fill polygons using different techniques.



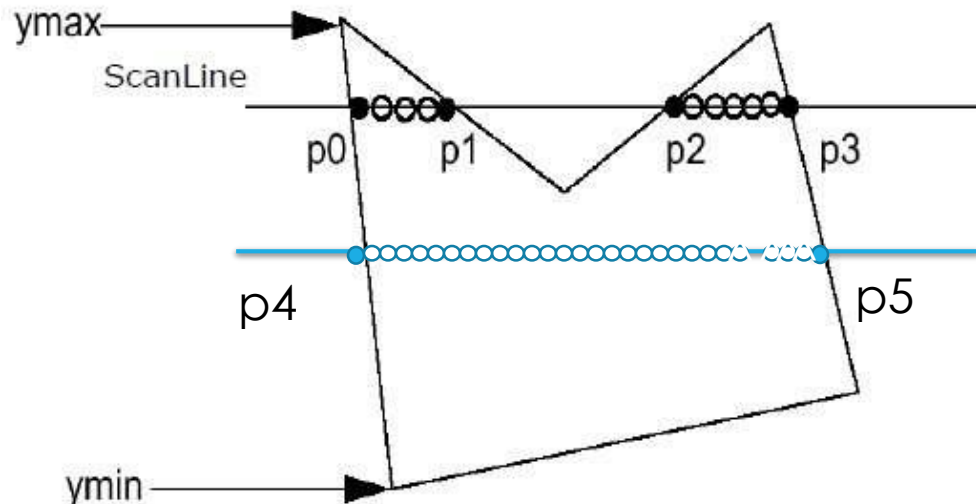
# Polygon Filling Algorithm

- 1) Scan Line Algorithm
- 2) Flood Fill Algorithm
- 3) Boundary Fill Algorithm
  - a. 4-Connected Polygon
  - b. 8-Connected Polygon

# Scan Line Algorithm

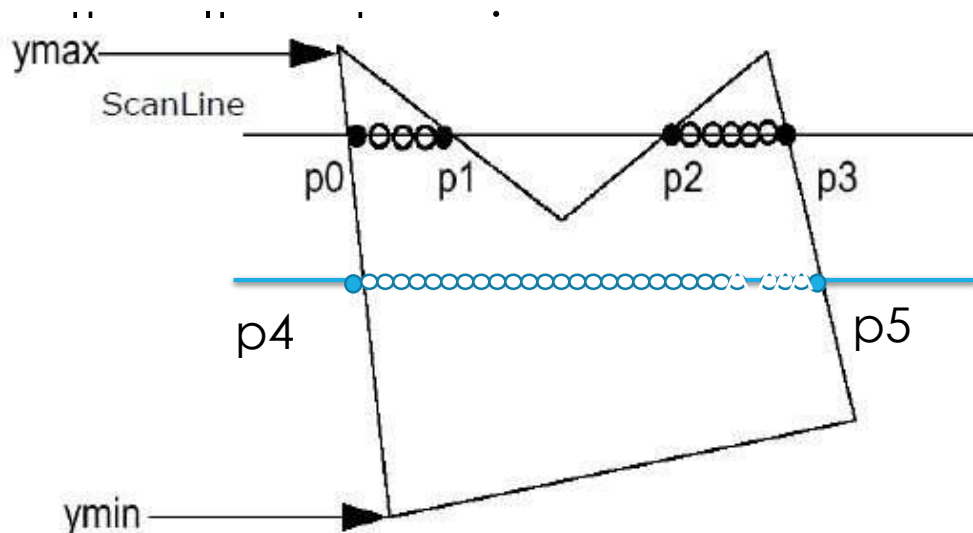
- This algorithm works by intersecting scanline with polygon edges and fills the polygon between pairs of intersections. The following steps depict how this algorithm works.

**Step 1** – Find out the Ymin and Ymax from the given polygon.



# Scan Line Algorithm

- **Step 2** – Scanline intersects with each edge of the polygon from Ymin to Ymax. Name each intersection point of the polygon.
- As per the figure show, they are named as p0, p1, p2, p3.
- **Step 3** – Sort the intersection point in the increasing order of X coordinate i.e. p0,p1 p1,p2 and p2,p3 .
- **Step 4** – Fill all those pair of coordinates that are inside polygons and ignore



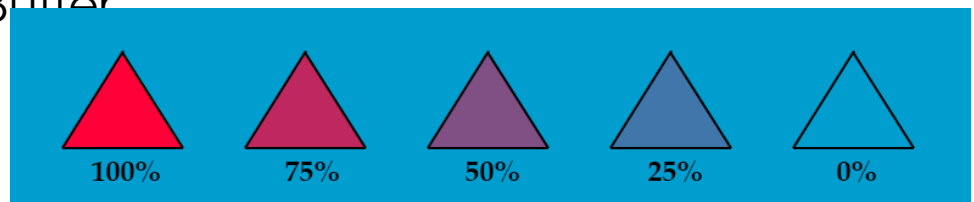
# Blending

- ❖ **Blending:** Color blending is a way to mix two colors together to produce to third color. These colors are called source and destination and they are in form  $[R,G,B,A]$   $[R,G,B,A]$  where  $R,G,B,A \in [0..1]$   $R,G,B,A \in [0..1]$ .
- ❖ Usually we use blending to represent semi transparent objects like glass.
- ❖ we can also create interesting effects by changing some parameters in the blend function.
- ❖ **Buffers:** at each drawing of a frame, OpenGL store information on all pixels of this frame into two different buffers, Color Buffer and Depth Buffer
- ❖ **fragment** : set of pixels
- ❖ **source (src):** incoming fragment, pixels that is being drawn
- ❖ **destination (dest) :** existing fragment in buffers, pixels that are already drawn in buffers

# Blending (cont.)

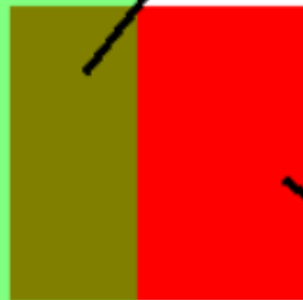
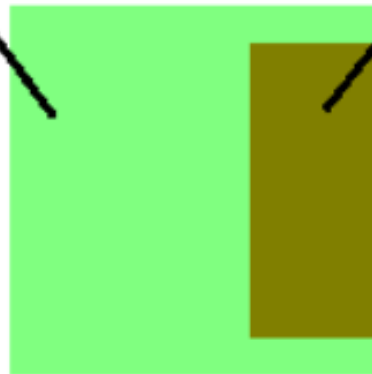
- ❖ With Blending, we can control how (and how much of) the existing color value should be combined with the new fragment's value (translucent)
- ❖ Without Blending, each new fragment overwrites any existing color values in the Color Buffer (opaque)
- ❖ alpha is the **A** in **RGBA**      `glColor4f(1.0, 0.0, 0.0, 0.0);`
- ❖ When Blending is enabled, the alpha value is used to combine the color value of the fragment being processed with that of the pixel already stored in the Color Buffer

Alpha  
value



RGBA: (0.5, 1.0, 0.5, 0.5)

RGBA: (0.5, 0.5, 0.0, 0.5)



RGBA: (1.0, 0.0, 0.0, 1.0)

# Source and Destination Factors

---

- ❖ Blending combine the source and destination being processed
- ❖ The combination is done using source and destination factors that are multiplied by their corresponding source and destination colors
- ❖ Suppose that we have:
  - source color = (srcR, srcG, srcB, srcA)      //pixel drawn
  - source factor = (sFR, sFG, sFB, sFA)      //src factor
  - destination color = (dstR, dstG, dstB, dstA)      //color in the buffer
  - destination factor = (dFR, dFG, dFB, dFA)      //dest factor

# Source and Destination Factors (cont.)

---

- ❖ The color obtained by the combination is the sum of products between the color and its associated factor, this is the Blending Formula :

- ❖  $(srcColor * srcFactor) + (destColor * destFactor)$

- ❖ Equivalent to :

$$\begin{aligned} & (srcR * sFR + dstR * dFR, \\ & \quad srcG * sFG + dstG * dFG, \\ & \quad \quad srcB * sFB + dstB * dFB, \\ & \quad \quad \quad srcA * sFA + dstA * dFA) \end{aligned}$$

# Source and Destination Factors (cont.)

---

- ❖ The source and destination factors are defined using:
- ❖ **`void glBlendFunc(GLenum srcfactor, GLenum destfactor);`**
  - `srcfactor` indicates how to compute a source blending factor
  - `destfactor` indicates how to compute a destination blending factor
- ❖ The blend factors are assumed to lie in the range [0, 1]
- ❖ After the color values in the source and destination are combined, they're clamped to the range [0, 1]

# Source and Destination Factors (4)

**srcfactor** and **destfactor** are one of these OpenGL constant

Default  
of DST

Default  
of SRC

Constant	Parameter	RGB Blend Factor	Alpha Blend Factor
GL_ZERO	src & dest	(0,0,0)	0
GL_ONE	src & dest	(1, 1, 1)	1
GL_SRC_COLOR	dest	$(R_s, G_s, B_s)$	$A_s$
GL_ONE_MINUS_SRC_COLOR	dest	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	src	$(R_d, G_d, B_d)$	$A_d$
GL_ONE_MINUS_DST_COLOR	src	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	src & dest	$(A_s, A_s, A_s)$	$A_s$
GL_ONE_MINUS_SRC_ALPHA	src & dest	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	src & dest	$(A_d, A_d, A_d)$	$A_d$
GL_ONE_MINUS_DST_ALPHA	src & dest	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$

# Enabling and Disabling Blending

---

- ❖ We need to enable blending to have it take effect:

**`glEnable(GL_BLEND)` ;**

- ❖ Use `glDisable(GL_BLEND)` to disable blending
- ❖ Using the constants **GL\_ONE** (source) and **GL\_ZERO** (destination) gives the same results as when blending is disabled

**`gl.glBlendFunc(GL.GL_ONE , GL.GL_ZERO);` // Default**

- ❖ Setting the alpha value:

`gl.glColor4f(1.0f, 1.0f, 0.0f, 0.75f);`

`gl.glColor4f(1.0f, 1.0f, 0.0f, 0.5f);`



The alpha values

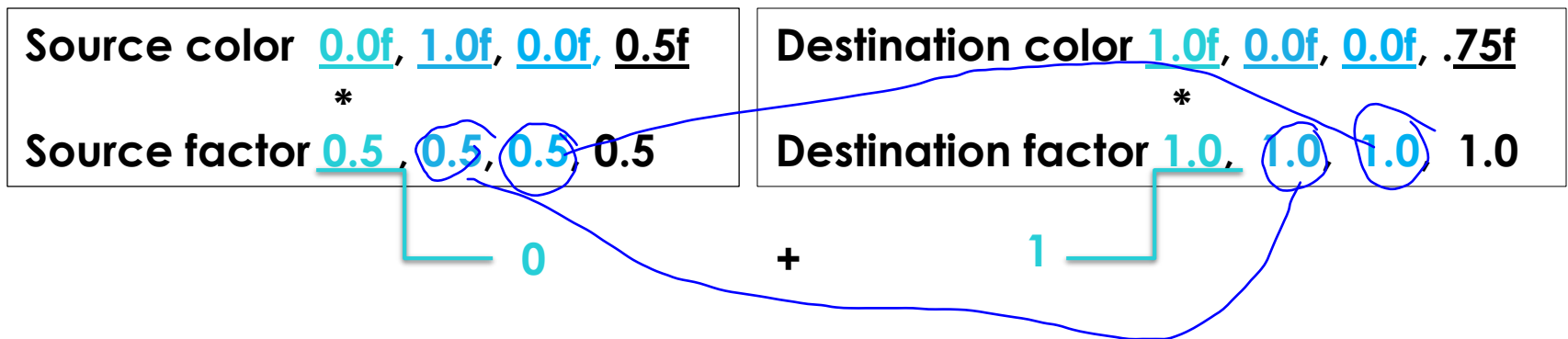
# Example



- Find blended color area of two overlapped polygon If the first polygon color (1.0f, 0.0f, 0.0f, .75f), the second polygon color is (0.0f, 1.0f, 0.0f, 0.5f); suppose the blinding function is gl.glBlendFunc(GL.GL SRC ALPHA, GL.GL ONE) .
- Solution:

Source factor

Destination factor



blended area color = 1.0f, 0.5f, 0.0f, 1f

# Exersice

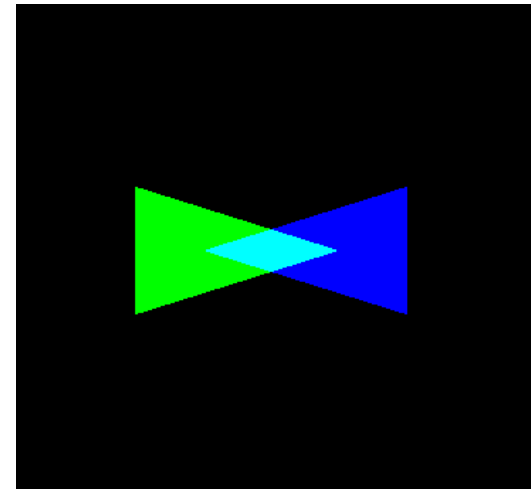
Find blended color area of three overlapped polygon If the first polygon color (1.0f, 0.0f, 0.0f, .75f), the second polygon color is (0.0f, 1.0f, 0.0f, 0.5f); and the third one is (0.0f, 0.0f, 1.0f, 0.2 f), suppose the blinding function is `gl.glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL.GL_ONE)` .

# Practical example

```
gl.glEnable(GL2.GL_BLEND) ;  
gl.glBlendFunc(GL2.GL_ONE, GL2.GL_ONE);
```

```
1.      gl.glColor4f(0,1f,0,.75f);  
2.      gl.glBegin(GL2.GL_TRIANGLES);  
3.      gl.glVertex2i(-100,50);  
4.      gl.glVertex2i(-100,-50);  
5.      gl.glVertex2i(50,0);  
6.      gl.glEnd();
```

```
1.      gl.glColor4f(0,0f,1f,.75f);  
2.      gl.glBegin(GL2.GL_TRIANGLES);  
3.      gl.glVertex2i(100,50);  
4.      gl.glVertex2i(100,-50);  
5.      gl.glVertex2i(-50,0);  
6.      gl.glEnd();
```



# Aliasing versus Antialiasing

---

## Aliasing

- ❖ In the drawing algorithms, all rasterized locations do not match with the true line, an optimal location is selected to represent a straight line
- ❖ This problem is severe in low resolution screens
- ❖ In such screens line appears like a stair-step, the effect is known as aliasing

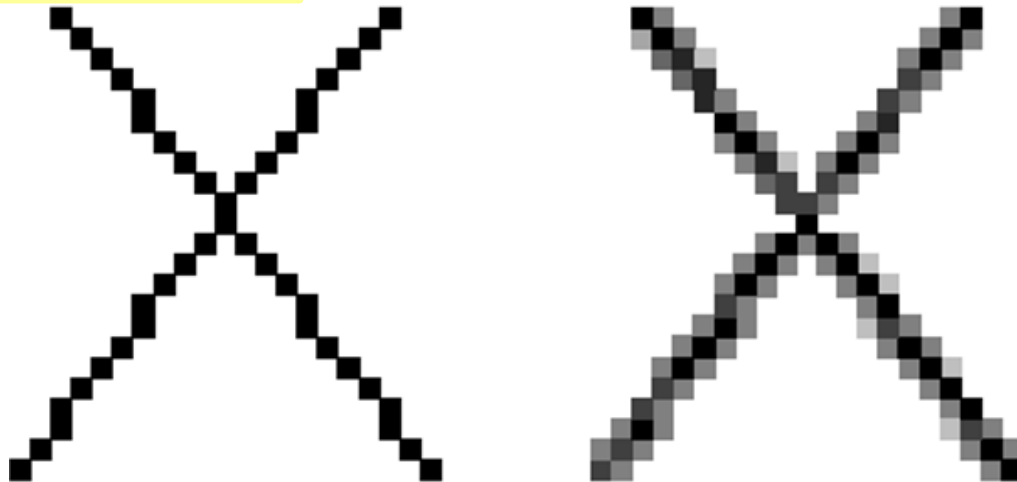
## Antialiasing

- ❖ This effect can be reduced by adjusting intensities of the pixels along the line
- ❖ The process of adjusting intensities of the pixels along the line to minimize the effect of aliasing is called antialiasing

# Antialiasing

---

- ❖ In some of OpenGL pictures, lines appear jagged
- ❖ The jaggedness is called **Aliasing**, and the techniques for reducing it are called **Antialiasing**



Aliased and Antialiased Lines

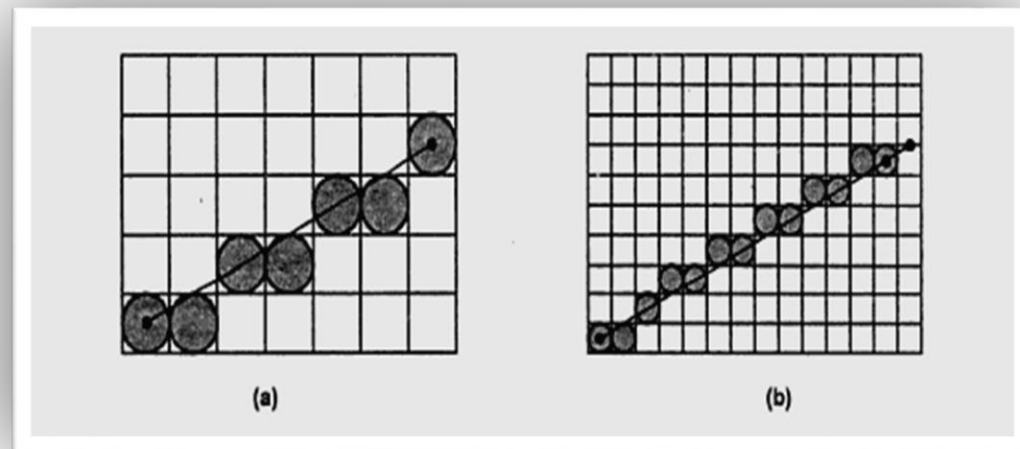
# Methods of Antialiasing

---

There are three methods which can be used to apply Antialiasing:

## First Method: Increasing Resolution

- ❖ Increase resolution of the raster display
- ❖ Improvement comes at the price of memory, bandwidth of memory and scan – conversion time

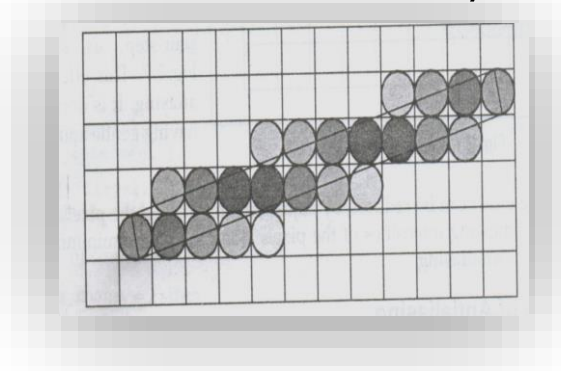


# Methods of Antialiasing (cont.)

---

## Second Method: Unweighted Area Sampling

- ❖ For sloped lines, many times the line passes between two pixels
- ❖ Line drawing algorithms select the pixel which is closer to the true line, this leads to perform antialiasing
- ❖ In antialiasing instead of picking closest pixel, **both pixels are highlighted**. However, their intensity values may differ

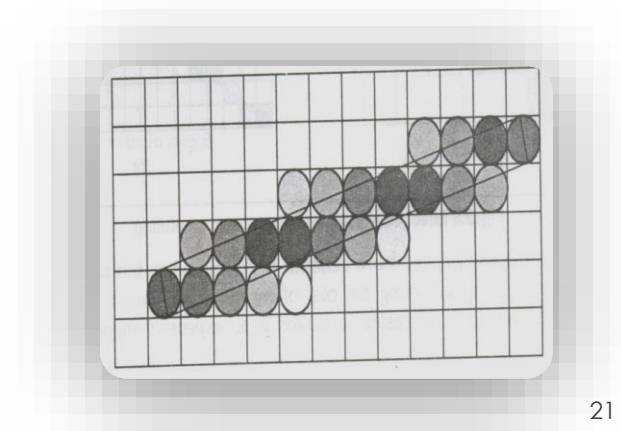


# Methods of Antialiasing (cont.)

---

## Third Method: Weighted Area Sampling

- ❖ In weighted area sampling equal areas contribute unequally i.e. a small area closer to the pixel center has greater intensity than does one at a greater distance. Thus, in weighted area sampling the intensity of the pixel is **dependent on the line area** occupied and the distance of area from the pixel's center



# Using Antialiasing

```
void glHint(GLenum target,  
            GLenum hint);
```



GL\_FASTEST  
GL\_NICEST  
GL\_DONT\_CARE

Controls certain aspects of OpenGL behavior

Target	Specifies
GL_POINT_SMOOTH_HINT GL_LINE_SMOOTH_HINT GL_POLYGON_SMOOTH_HINT	sampling quality of points, lines, or polygons during antialiasing operations
...	

# Using Antialiasing (cont.)

## 1. **Enable antialiasing**

`glEnable(primitiveType)`

`GL_POINT_SMOOTH`

`GL_LINE_SMOOTH`

`GL_POLYGON_SMOOTH`

## 2. **Enable Blending**

`glEnable(GL_BLEND);`

## 3. **Apply Blending**

`glBlendFunc( GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA );`

## 4. **Provide a quality hint**

`glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);`

# Point and Line Antialiasing

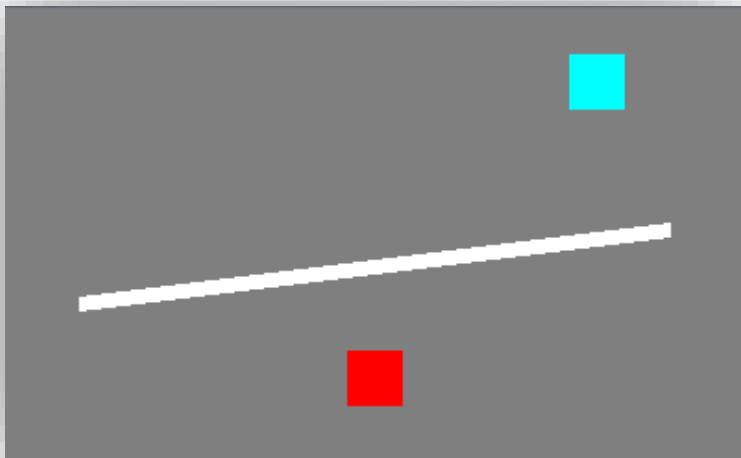
---

## Point Smoothing

```
glEnable (GL_POINT_SMOOTH);
```

(no need for blending)

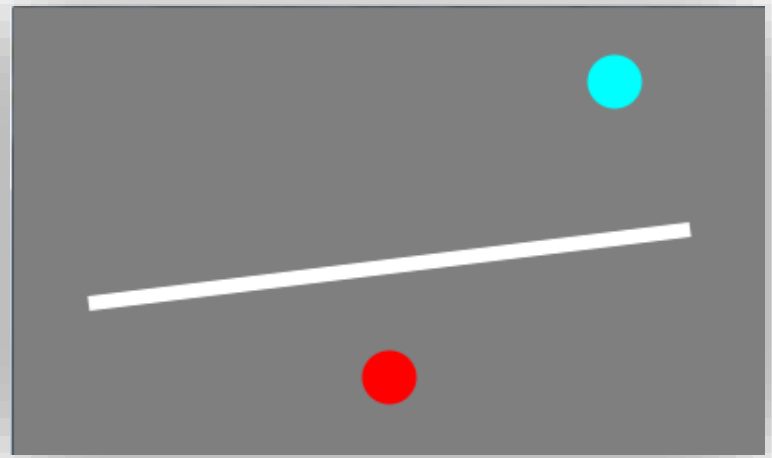
Aliasing



## Line Smoothing

```
glEnable (GL_LINE_SMOOTH);  
glEnable (GL_BLEND);  
glBlendFunc (GL_SRC_ALPHA,  
             GL_ONE_MINUS_SRC_ALPHA);
```

Antialiasing



# Polygon Antialiasing

---

Aliasing



Antialiasing



# Reference

- Reference book of the course
- [https://www.tutorialspoint.com/computer\\_graphics/polygon\\_filling\\_algorithm.htm](https://www.tutorialspoint.com/computer_graphics/polygon_filling_algorithm.htm)