

ARSI UNIVERSITY

Computer Graphics

Lab Manual

Prepared BY: Debi Leta

@Asella


Contact: ☎ +2519 836 726 20  <https://t.me/debileta>

Table of Contents

Table of Contents	1
Introduction	2
Configuring DevC++	2
Circle Program	3
Line Program	4
Rectangle program	5
Ellipse Program	6
Sector Program	6
Polygon Program	7
Color, Pixels and Bar	7
Three Merged Circles with Color	9
Three Circles with Different Pattern	10
DDA – Line Program	11
Animated Circle	14
Concentric Circles	15
Bar chart	15
Progress Bar	17

Introduction

Computer graphics deals with creation, manipulation and storage of different type of images and objects. Some of the applications of computer graphics are: Computer Art, Presentation Graphics, Computer Aided Drawing, Presentation Graphics, Entertainment, Education, Training, Visualizations, Image Processing, Machine Drawing, and Graphical User Interface.

So far we have been using C language for simple console output only. Most of us are unaware that using C++, low level graphics program can also be made. This means we can incorporate shapes, colors and designer fonts in our program. This article deals with the steps to enable the DevC++ compiler to generate graphics

Configuring DevC++

To install and use **DevC++** for graphics codes follow the following steps:

Step 1: Download the **DevC++** version 5.11 from here → <https://sourceforge.net/projects/orwelldvcpp/>

Step 2: Download the Graphics header, from here → <https://github.com/SagarGaniga/Graphics-Library/>

Step 3: Install DevC++

Step 4: Extract the contents of the **rar** file of Graphics header file.

Step 4: Go to the location where **DevC++** is installed. For me it's C:\Dev-Cpp.

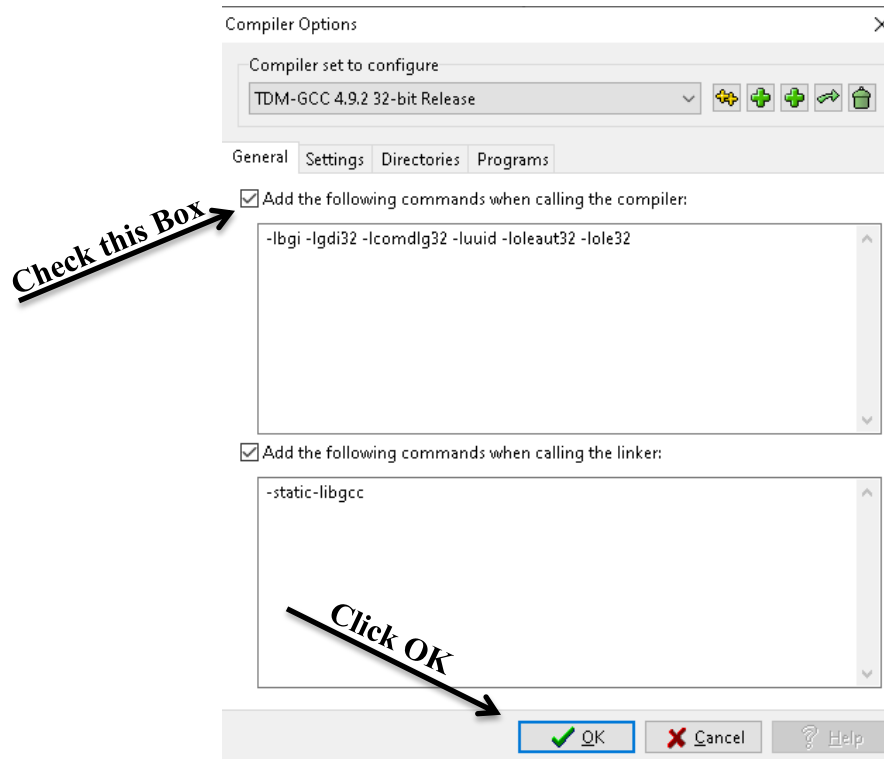
Copy the **graphics.h** and **winbgim.h** in the include folder and paste it in **C:\Dev-Cpp\MinGW64\include** and **C:\Dev-Cpp\MinGW64\x86_64-w64-mingw32\include** folders.

Step 5: Copy the **libbgi.a** file in extracted folder into **lib** folder in **C:\Dev-Cpp\MinGW64\lib** and **C:\Dev-Cpp\MinGW64\x86_64-w64-mingw32\lib** folders.

Step 6: Open DevC++, the goto **Tools** menu, click on **Compiler Options** submenu. Select **TDM-CGG 4.9.2 32-bit release** under **Compiler set to configure** drop down box. Then Copy the following code inside **compiler** field under **General** tab as it is.

```
-lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32
```

Lastly check for the first option as follows.



Circle Program

```
#include<graphics.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    circle(100, 100, 50);
    delay(2000);
    getch();
    closegraph();
    return 0;
}
```

Explanation:

1. **#include<graphics.h>:** This line includes the header file **graphics.h**, which contains declarations for functions and constants used in BGI (Borland Graphics Interface) graphics programming. This header file is specific to compilers like Borland Turbo C++.
2. **int main():** This is the main function of the program, where execution begins. In C/C++, every program must have a **main()** function.

3. **int gd = DETECT, gm;;** This line declares two integer variables **gd** and **gm**. **gd** stands for "*graphics driver*" and **gm** stands for "*graphics mode*". **DETECT** is a constant defined in **graphics.h** that instructs the graphics library to detect the graphics driver and mode automatically.
4. **initgraph(&gd, &gm, NULL);** This function initializes the graphics system. It sets up the graphics environment using the parameters **gd** (*graphics driver*) and **gm** (*graphics mode*). Passing **NULL** as the third parameter indicates that no additional initialization options are provided.
5. **circle(100, 100, 50);** This function draws a circle on the graphics screen. It takes three parameters: the *x-coordinate* and *y-coordinate* of the center of the circle (100, 100 in this case), and the *radius* of the circle (50 in this case).
6. **delay(2000);** This function pauses the execution of the program for 2000 milliseconds (2 seconds). It allows the user to see the drawn circle for a brief period before continuing.
7. **getch();** This function waits for a key press from the user. It prevents the program from closing immediately after drawing the circle, allowing the user to see the output.
8. **closegraph();** This function closes the graphics system and releases the resources allocated by **initgraph()**. It's essential to call this function at the end of the program to clean up properly.
9. **return 0;** This statement indicates that the program execution was successful, and it's returning 0 to the operating system, which typically indicates success.

Overall, this program initializes a graphics window, draws a circle in it, waits for 2 seconds, waits for a key press, and then closes the window. It's a simple example of how to use the BGI library for basic graphics programming in C/C++.

Line Program

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    line (100, 100, 300, 400);
    line (150, 100, 350, 400);
    outtextxy (200, 400, "TWO LINES");
    getch();
    closegraph();
}
```

Explanation:

Let's explain a new thing here. Because the others explained in the Circle programs above.

1. **conio.h** provides functions for console input/output, such as **getch()**
2. **line(100, 100, 300, 400); line(150, 100, 350, 400);** These two lines draw straight lines on the graphics window. The line function takes four parameters: the **x** and **y coordinates** of the *starting* point and the **x** and **y coordinates** of the *ending* point. So, the first line call draws a line from (100, 100) to (300, 400), and the second call draws a line from (150, 100) to (350, 400).
3. **outtextxy(200, 400, "TWO LINES");** This line prints the text "TWO LINES" at the specified coordinates (200, 400) on the graphics window. The **outtextxy** function is used to output text at a specific position on the graphics screen.

Overall, this program initializes a graphics window, draws two lines, prints text on the screen, waits for a key press, and then closes the graphics window before exiting.

Rectangle program

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    rectangle (200, 50, 450, 200);
    outtextxy (300, 230, "RECTANGLE");
    getch();
    closegraph();
    return 0;
}
```

Explanation:

Let's explain a new thing here.

1. The code includes the necessary header files for the **graphics** and **console** input/output functions. **graphics.h** provides graphics functions, while **conio.h** provides functions for console input/output, such as **getch()**.
2. **rectangle(200, 50, 450, 200);** This line draws a rectangle on the graphics window. The rectangle function takes four parameters: the **x** and **y coordinates** of the **top-left corner** of the rectangle and the **x** and **y coordinates** of the **bottom-right corner**. So, the rectangle in this code will be drawn with the **top-left corner** at (200, 50) and the **bottom-right corner** at (450, 200).

Overall, this program initializes a graphics window, draws a rectangle, prints text on the screen, waits for a key press, and then closes the graphics window before exiting.

Ellipse Program

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    ellipse (400, 250, 0, 360, 200, 100);
    outtextxy (360, 380, "ELLIPSE");
    getch();
    closegraph();
    return 0;
}
```

Explanation:

Let's explain a new thing here.

1. **ellipse(400, 250, 0, 360, 200, 100);** This line draws an ellipse on the graphics window. The ellipse function takes six parameters: the **x** and **y coordinates** of the **center** of the ellipse, the **starting angle** of the ellipse arc, the **ending angle** of the ellipse arc, the **horizontal radius** of the ellipse, and the **vertical radius** of the ellipse. In this case, an ellipse **centered** at (400, 250) with **horizontal radius 200, vertical radius 100**, and a full **360-degree arc** will be drawn.

Sector Program

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    sector (200, 200, 30, 300, 150, 100);
    outtextxy (180, 330, "SECTOR");
    getch();
    closegraph();
    return 0;
}
```

Explanation:

1. **sector(200, 200, 30, 300, 150, 100);** This line draws a **sector** (a portion of a circle) on the graphics window. The sector function takes six parameters: the **x** and **y coordinates** of the **center** of the sector, the **starting angle** of the sector in degrees, the **ending angle** of the sector in degrees,

the **horizontal radius** of the sector, and the **vertical radius** of the sector. In this case, a sector centered at (200, 200) with a starting angle of 30 degrees, an ending angle of 300 degrees, a horizontal radius of 150, and a vertical radius of 100 will be drawn.

Polygon Program

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    int poly[12] = {200, 300, 500, 300, 300, 100, 300, 400, 450, 100, 200, 300 };
    initgraph(&gd, &gm, NULL);
    drawpoly(6, poly);
    outtextxy (400, 350, "POLYGON");
    getch();
    closegraph();
    return 0;
}
```

Explanation:

```
int poly[12] = {200, 300, 500, 300, 300, 100, 300, 400, 450, 100, 200, 300 };
```

This line initializes an array poly with **12 elements**, representing the **x and y coordinates** of the vertices of a polygon. Each pair of elements represents the x and y coordinates of a vertex. For example, (200, 300) represents the first vertex, (500, 300) represents the second vertex, and so on.

Color, Pixels and Bar

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    setcolor(3);
    setfillstyle(SOLID_FILL, RED);
    bar(50, 50, 590, 430);

    setfillstyle(1, 14);
    bar(100, 100, 540, 380);

    while (!kbhit())
```



```

    {
        putpixel(rand() % 439 + 101, rand() % 279 + 101, rand() % 16);
        setcolor(rand() % 16);
        circle(320, 240, rand() % 100);
    }

    getch();
    closegraph();
    return 0;
}

```

Explanation:

1. **stdlib.h** provides general utility functions (such as **rand()**).

2. **setcolor(3);**

```
setfillstyle(SOLID_FILL, RED);
```

```
bar(50, 50, 590, 430);
```

These lines set the current drawing color to the color with index **3** (which is typically **cyan**), set the **fill style** to **solid** with the fill color as **red**, and draw a filled **rectangle (bar)** on the graphics window with the specified coordinates.

3. **setfillstyle(1, 14);**

```
bar(100, 100, 540, 380);
```

Similarly, these lines set the **fill style** to pattern fill with pattern **ID 1** and **pattern color** index **14** (which is typically light **gray**), and draw another filled **rectangle** on the graphics window.

4. **while (!kbhit())**

```

{
    putpixel(rand() % 439 + 101, rand() % 279 + 101, rand() % 16);
    setcolor(rand() % 16);
    circle(320, 240, rand() % 100);
}

```

This **while loop** continues until a key is pressed (using **kbhit()** from **conio.h**). Within the loop, **putpixel()** is used to draw a single pixel with a random color at a random location within a specified range. **setcolor()** sets the drawing color to a random color, and **circle()** draws a circle with a random radius at the fixed center coordinates (320, 240).

Three Merged Circles with Color

```
#include<graphics.h>
#include<conio.h>
#include<dos.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    setbkcolor(BLUE);
    setcolor(RED);
    setfillstyle(1, YELLOW);
    circle(200, 150, 100);
    circle(350, 150, 100);
    circle(280, 280, 100);
    floodfill(200, 150, RED);
    floodfill(350,150, RED);
    floodfill(280,280, RED);
    getch();
    closegraph();
    return 0;
}
```

Explanation:

1. **dos.h** provides functions for DOS interrupt handling. It includes functions for accessing low-level hardware and system services in DOS-based operating systems (Microsoft Disk Operating System).
2. **setbkcolor(BLUE);** This line sets the background color of the graphics window to blue.
3. **circle(200, 150, 100);**
circle(350, 150, 100);
circle(280, 280, 100);

These lines draw three circles on the graphics window with the specified center coordinates and radius.

4. **floodfill(200, 150, RED);**
floodfill(350, 150, RED);
floodfill(280, 280, RED);

These lines perform flood fill operations inside the circles. The **floodfill()** function fills the area with the current fill color starting from the specified seed point. In this case, the seed point is at the center of each circle, and the fill color is red.

Three Circles with Different Pattern

```
#include<graphics.h>
#include<conio.h>
#include<dos.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    setbkcolor(6);
    setcolor(RED);
    setfillstyle(1, YELLOW);
    circle(200, 150, 100);
    floodfill(200, 150, RED);

    setcolor(WHITE);
    setfillstyle(2, CYAN);
    circle(400, 150, 100);
    floodfill(400, 150, WHITE);

    setcolor(GREEN);
    setfillstyle(4, MAGENTA);
    circle(300, 325, 100);
    floodfill(300, 330, GREEN);

    getch();
    closegraph();
    return 0;
}
```

Explanation:

1. **setbkcolor(6);** sets the background color to color index 6, which typically represents **cyan**.
2. **setcolor(RED);** sets the drawing color to red.

3. **setfillstyle(1, YELLOW);** sets the fill pattern to solid fill (1) with the fill color as yellow.
4. **circle(200, 150, 100);** draws a circle with center coordinates (200, 150) and radius 100.
5. **floodfill(200, 150, RED);** fills the area around the circle with red color using flood fill starting from point (200, 150).

Similarly, the next two blocks of code draw circles with different colors and fill patterns, and then perform flood fill to color the areas around the circles.

DDA – Line Program

DDA stands for **Digital Differential Analyzer**, and it's a line drawing algorithm used in computer graphics to approximate lines on a digital display. The DDA algorithm calculates the coordinates of the pixels that lie on the line between two given points.

Here's how the DDA algorithm works:

Input: The algorithm takes as input the coordinates of two endpoints of the line: (x1, y1) and (x2, y2).

Calculate Differences: Calculate the differences in x and y coordinates between the two endpoints:

$$dx = x2 - x1$$

$$dy = y2 - y1$$

Determine Steps: Determine the number of steps needed to increment either the x or y coordinate to traverse from one endpoint to the other.

This is typically based on the larger difference between dx and dy:

$$steps = \max(|dx|, |dy|)$$

Calculate Increments: Calculate the increment values for each step in the x and y directions:

$$xIncrement = dx / steps$$

$$yIncrement = dy / steps$$

Initialize Coordinates: Initialize the starting coordinates (x, y) to (x1, y1).

Draw Line: For each step, increment the coordinates (x, y) by the corresponding increments (*xIncrement*, *yIncrement*) and plot the pixel at the rounded coordinates:

$$x = x + xIncrement$$

$$y = y + yIncrement$$

$$putpixel(round(x), round(y), color)$$

Repeat: Repeat the process for each step until the final endpoint is reached.

The DDA algorithm approximates the line by incrementally moving from one endpoint to the other and plotting the pixels along the way. It's relatively simple and efficient, making it suitable for drawing lines on digital displays. However, it may not always produce the most accurate results, especially for lines with large slopes or when high precision is required. In such cases, alternative algorithms like Bresenham's line algorithm may be preferred.

```
#include <dos.h>
#include <graphics.h>
#include <conio.h>
#include <iostream>
#include <cmath>
using namespace std;

void lineDDA(int, int, int, int);

int main()
{
    int x1, y1, xn, yn;

    // Create a separate console window for input/output
    AllocConsole();
    freopen("CONIN$", "r", stdin);
    freopen("CONOUT$", "w", stdout);

    cout << "Enter the starting coordinates of the line: ";
    cin >> x1 >> y1;
    cout << "Enter the ending coordinates of the line: ";
    cin >> xn >> yn;

    // Initialize the graphics system after getting input
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    // Call lineDDA to draw the line
    lineDDA(x1, y1, xn, yn);

    getch();
    closegraph();
    return 0;
}

void lineDDA(int x1, int y1, int xn, int yn)
{
    int dx, dy, steps;
    float xIncrement, yIncrement, x, y;

    // Calculate differences and take absolute values
    dx = xn - x1;
```

```

dy = yn - y1;
if (abs(dx) > abs(dy))
    steps = abs(dx);
else
    steps = abs(dy);

// Calculate increments for x and y
xIncrement = (float)dx / steps;
yIncrement = (float)dy / steps;

// Initialize starting coordinates
x = x1;
y = y1;

// Draw each pixel along the line
for (int i = 0; i <= steps; i++)
{
    putpixel((int)(x + 0.5), (int)(y + 0.5), RED);
    // Round coordinates before plotting
    x += xIncrement;
    y += yIncrement;
}
}

```

Explanation:

1. **cmath** provides mathematical functions.
2. **void lineDDA(int, int, int, int);** This line declares the prototype of the lineDDA function, which will be defined later in the code. This function is responsible for drawing a line using the DDA algorithm.
3. The **main()** function prompts the user to enter the starting and ending coordinates of the line. It then initializes the graphics system using **initgraph()** after creating a separate console window for input/output.
The **lineDDA** function is called to draw the line between the given points.
Finally, the program waits for a key press (**getch()**) and closes the graphics window (**closegraph()**).
4. The **lineDDA** function calculates the differences in x and y coordinates between the given endpoints. It determines the number of steps needed to traverse from one endpoint to the other.
Increments for x and y are calculated based on the number of steps.
Starting coordinates are initialized, and then each pixel along the line is plotted using **putpixel()**.

Coordinates are rounded before plotting to ensure accurate pixel placement.

Overall, this program allows the user to input the starting and ending coordinates of a line and then draws the line using the DDA algorithm in a graphics window.

Animated Circle

```
#include<stdlib.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
int main()
{
    int x,y,i;
    int g=DETECT,d;
    initgraph(&g, &d, NULL);
    cleardevice();
    x=getmaxx()/2;
    y=getmaxy()/2;
    settxtstyle(TRIPLEX_FONT, HORIZ_DIR, 3);
    setbkcolor(rand());
    setcolor(4);
    outtextxy(30,100,"Press any key to Quit");
    while (!kbhit())
    {
        setcolor(rand());
        for (int i=0;i<50;i++)
            circle(x,y,i );
        setcolor(rand());
        for (int j=70;j<120;j++)
            circle(x,y,j);
        setcolor(rand());
        for (int k=140;k<190;k++)
            circle(x,y,k);
        setcolor(rand());
        for (int l=210;l<230;l++)
            circle(x,y,l);
        delay(200);
    }
    getch();
    closegraph();
}
```

Explanation:

1. **stdlib.h**: Provides functions like **rand()** for generating random numbers.
2. **cleardevice()** clears the screen.
3. **getmaxx()** and **getmaxy()** get the maximum screen width and height, respectively. Dividing by 2 centers **x** and **y** on the screen.
4. **settextstyle(TRIPLEX_FONT, HORIZ_DIR, 3)** sets the font to **TRIPLEX**, with horizontal orientation and a size of 3.
5. **setbkcolor(rand())** sets a random background color, where **rand()** returns a pseudo-random number.
6. **setcolor(4)** changes the drawing color to 4 (typically red).
7. **outtextxy(30, 100, "Press any key to Quit")** displays the text "Press any key to Quit" at position (30, 100).

Concentric Circles

```
#include <graphics.h>
#include <conio.h>
int main()
{
    int gd=DETECT, gm;
    int x=320, y=240, r;
    initgraph(&gd, &gm, NULL);

    for ( r=25; r <=225 ; r=r+20)
        circle(x, y, r);
    getch();
    closegraph();
}
```

Bar chart

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    // Drawing the Chart Border and Title
    setcolor(YELLOW);
    rectangle(0,30,639,450);
```



```
settextstyle(SANS_SERIF_FONT,HORIZ_DIR,2);
setcolor(WHITE);
outtextxy(275,20,"Bar Chart");
// Drawing the Axes
setlinestyle(SOLID_LINE,0,2);
line(100,420,100,60);
line(100,420,600,420);
line(90,70,100,60);
line(110,70,100,60);
line(590,410,600,420);
line(590,430,600,420);

// Labeling the Axes:
outtextxy(95,35,"Y");
outtextxy(610,405,"X");
outtextxy(85,415,"O");

// Drawing the Bars:
setfillstyle(LINE_FILL,BLUE);
bar(150,100,200,419);

setfillstyle(XHATCH_FILL,RED);
bar(225,150,275,419);

setfillstyle(WIDE_DOT_FILL,GREEN);
bar(300,200,350,419);

setfillstyle(INTERLEAVE_FILL,MAGENTA);
bar(375,125,425,419);

setfillstyle(HATCH_FILL,BROWN);
bar(450,175,500,419);
getch();
closegraph();
}
```

Explanation:

1. **dos.h**: Used here, though its functions are not explicitly called in this code. This is sometimes included for compatibility purposes when using certain graphics modes or functions.
2. Drawing the Chart Border and Title:
 - **setcolor(YELLOW)** sets the color of the border to yellow.

- **rectangle(0, 30, 639, 450)** draws a yellow rectangle from the top-left (0, 30) to the bottom-right (639, 450), forming a border for the chart.
- **settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 2)** sets the text style to a *sans-serif* font with a horizontal direction and a font size of 2.
- **outtextxy(275, 20, "Bar Chart")** places the chart title at position (275, 20) in white.

3. Drawing the Axes:

- **Axes:** The Y-axis runs vertically from (100, 420) to (100, 60), and the X-axis runs horizontally from (100, 420) to (600, 420).
- **Arrows:** Arrowheads are added to indicate the axes' directions, with lines creating small triangles at the top of the Y-axis and at the end of the X-axis.

4. Labeling the Axes:

- Labels **Y**, **X**, and **O** (origin) are added near the ends of the **Y-axis** and **X-axis** and at the point (100, 420) where they intersect.

5. Drawing the Bars:

- Each bar is drawn using **bar()** with a unique color and fill style to represent different values.
- The first bar is drawn from (150, 100) to (200, 419) with a blue **LINE_FILL** pattern. The bar's top coordinate (150, 100) represents its height based on data value, and (200, 419) defines its width and bottom position.
- The second bar, from (225, 150) to (275, 419), has a red **XHATCH_FILL** pattern. The height is lower than the first bar, creating a visual difference in data representation.
- The third bar, with a green **WIDE_DOT_FILL** pattern, extends from (300, 200) to (350, 419). It's lower than the previous two bars.
- The fourth bar is from (375, 125) to (425, 419) with a magenta **INTERLEAVE_FILL** pattern, taller than some bars but shorter than the first.
- The fifth and final bar is from (450, 175) to (500, 419) with a brown **HATCH_FILL** pattern. Its height represents its data value, making it taller than some previous bars.

Progress Bar

This program creates a simple animated progress bar in graphics mode using the **<graphics.h>** library. It uses some basic graphics functions to draw a rectangle and move the bar within it.

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
```

```
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    void *buffer;
    unsigned int size;
    setbkcolor(BLUE);
    line(230, 330, 370, 330);
    line(230, 350, 370, 350);
    //form the left and right borders (progress bar), giving it a 3D-like effect.
    line(226, 335, 226, 345);
    line(226, 335, 230, 330);
    line(226, 345, 230, 350);

    line(374, 335, 374, 345);
    line(374, 335, 370, 330);
    line(374, 345, 370, 350);

    outtextxy(275, 365, "Loading");

    int i, x = 232, y = 336, x1 = 236, y1 = 344;
    for (i = 1; i < 5; i++)
    {
        setfillstyle(1, RED);
        bar(x, y, x1, y1);
        x = x1 + 2;
        x1 = x1 + 6;
    }

    size = imagesize(232, 336, 256, 344);
    buffer = malloc(size);
    if (buffer == NULL) {
        printf("Memory allocation failed");
        closegraph();
        return 1; // Return 1 for error case
    }

    getimage(232, 336, 256, 344, buffer);
    x = 232;
    int m = 0;

    while (!kbhit())
```

```

{
    putimage(x, 336, buffer, XOR_PUT);
    x = x + 2;
    if (x >= 350)
    {
        m++;
        x = 232;
        if (m == 10) // Number of loops
            break;
    }
    putimage(x, 336, buffer, XOR_PUT);
    delay(20); // Controls the speed of movement
}
free(buffer); // Free allocated memory
getch();
closegraph();
return 0; // Return 0 to indicate successful completion
}

```

Explanation:

1. **imagesize()** calculates the memory size required to store the filled bar segment as an image.
2. **malloc()** allocates memory for the **buffer** where the segment will be stored.
3. **getimage()** copies the bar segment from the screen into the **buffer**.
4. **putimage(x, 336, buffer, XOR_PUT);** draws the bar segment at the x position.
5. If x reaches **350**, the bar resets to the starting position at $x = 232$, repeating **10** times.
6. **delay(20);** controls the speed, with a lower delay resulting in faster movement.
7. **free(buffer);** releases the dynamically allocated memory.

Display ROAD Format

The program prompts the user to enter five colors: one for the road, one for the lane separator, one for the dashed lines, and one for the left and right grass. The user must input numbers corresponding to the desired colors. For example:

- 8 for gray (road)
- 15 for white (lane separator and dashed lines)
- 2 for green (grass)

```

#include <graphics.h>
#include <iostream>
#include <conio.h>

```

```
using namespace std;
int main() {
    int gd = DETECT, gm;
    int roadColor, laneColor, dashedLineColor, grassColorLeft, grassColorRight;

    // Initialize the graphics mode
    initgraph(&gd, &gm, NULL);

    // Ask user for colors for the road, lane, dashed lines, and grass
    cout << "Enter the color for the road (e.g., 8 for gray): ";
    cin >> roadColor; // e.g., 8 for gray

    cout << "Enter the color for the lane separator (e.g., 15 for white): ";
    cin >> laneColor; // e.g., 15 for white

    cout << "Enter the color for the dashed lines (e.g., 15 for white): ";
    cin >> dashedLineColor; // e.g., 15 for white

    cout << "Enter the color for the left grass (e.g., 2 for green): ";
    cin >> grassColorLeft; // e.g., 2 for green

    cout << "Enter the color for the right grass (e.g., 2 for green): ";
    cin >> grassColorRight; // e.g., 2 for green

    // Set background color to light blue (sky color)
    setbkcolor(CYAN);
    cleardevice();

    // Draw the road (a large gray rectangle)
    setcolor(roadColor); // Road color
    setfillstyle(SOLID_FILL, roadColor); // Fill color
    rectangle(100, 200, 500, 400); // Road's main rectangle
    floodfill(300, 300, roadColor); // Fill the road with color

    // Draw the center dashed line (representing lanes)
    setcolor(dashedLineColor); // Line color for dashed lines
    setlinestyle(DASHED_LINE, 0, 3); // Dashed line style
    for (int i = 110; i < 500; i += 40) { // Draw dashes across the road
        line(i, 300, i + 20, 300);
    }

    // Draw road boundaries (left and right edges)
    setcolor(dashedLineColor); // Color for road edges
    line(100, 200, 100, 400); // Left edge of the road
    line(500, 200, 500, 400); // Right edge of the road
```

```
// Add lane dividers
setcolor(laneColor); // Lane divider color
setlinestyle(SOLID_LINE, 0, 1); // Solid line for lane dividers
for (int i = 150; i < 500; i += 100) { // Lane dividers
    line(i, 200, i, 400);
}

// Add road shoulder or grass on the sides
setcolor(grassColorLeft); // Color for left grass
setfillstyle(SOLID_FILL, grassColorLeft);
rectangle(0, 200, 100, 400); // Left side (grass)
floodfill(50, 300, grassColorLeft);

setcolor(grassColorRight); // Color for right grass
setfillstyle(SOLID_FILL, grassColorRight);
rectangle(500, 200, 640, 400); // Right side (grass)
floodfill(550, 300, grassColorRight);

// Display message
outtextxy(200, 450, "Road Visualization");

// Wait for user input to close the window
getch();
closegraph();
}
```

Explanation:

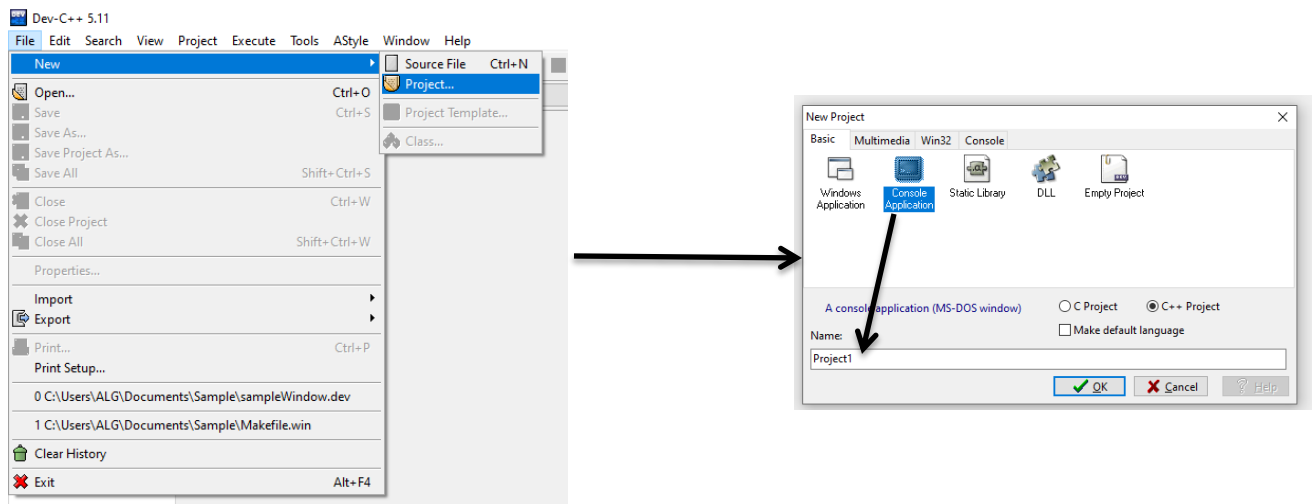
1. **setcolor()** sets the color for drawing shapes (like lines and rectangles).
2. **setfillstyle()** sets the fill style for the shapes (in this case, **SOLID_FILL** fills the shapes with a single color).
3. **rectangle()** draws a rectangle to represent the road. It starts at coordinates **(100, 200)** and ends at **(500, 400)**.
4. **floodfill()** fills the area inside the rectangle with the specified color (**roadColor**).
5. **setlinestyle(DASHED_LINE, 0, 3)** sets the line style to dashed. **DASHED_LINE** specifies the dashed pattern.

OpenGL

After you install Dev C++ as we discussed above under introduction section, then download **freeglut** under **MinGW** from this link. <https://www.transmissionzero.co.uk/software/freeglut-devel/>

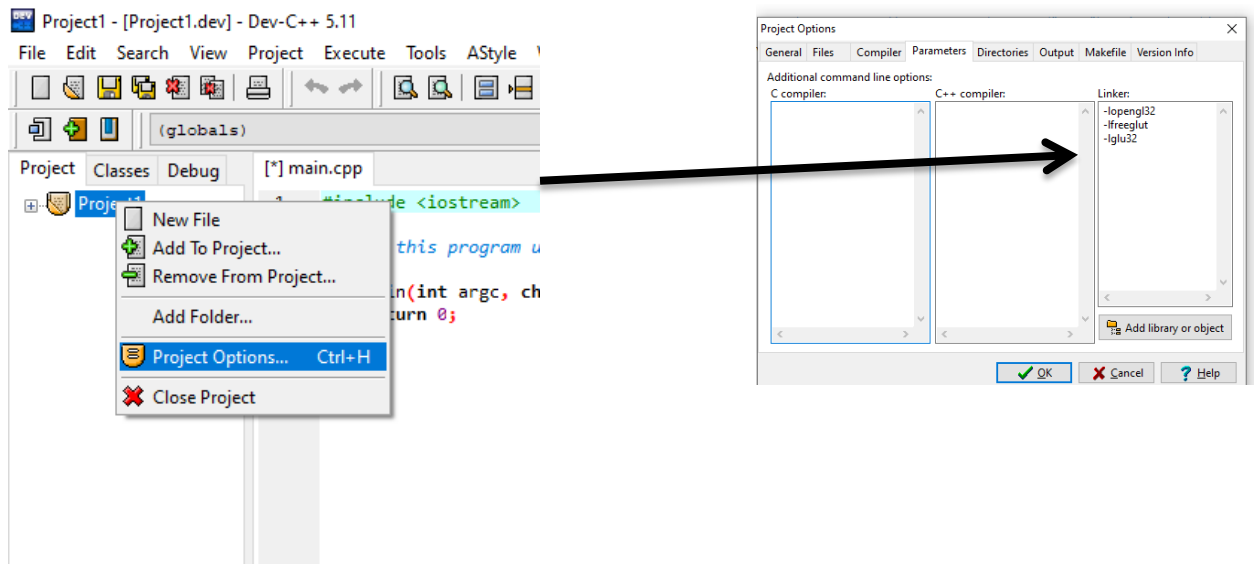
Then follow the following steps to configure freeglut.

- **Step 1:** Extract downloaded rar
- **Step 2:** Copy all the files inside **freeglut\include\GL** into
C:\Dev-Cpp\MinGW64\x86_64-w64-mingw32\include\GL path.
- **Step 3:** Copy all the files inside **freeglut\lib\x64** into
C:\Dev-Cpp\MinGW64\x86_64-w64-mingw32\lib path
- **Step 4:** Copy file inside **freeglut\bin\x64** into **C:\Windows\System32** path
- **Step 5:** Open Dev C++ and create new project as follows (*Click on Project*)



Select Console Application and give project name as you see above.

- **Step 6:** Select project location (*where you save*) and click **save**
- **Step 7:** Right click on project name → Select **Project Options** → Select **Parameter** → copy the following code and paste it under **Linker** → Click **OK**



Sample GLUT Program

When the following program executed, a window appears with the title "**Simple C++ OpenGL Program**". Inside the window, a square (polygon) is displayed, centered at the origin. The square occupies half the width and height of the window (since its vertices range from **-0.5** to **0.5** in normalized device coordinates).

```
#include <GL/glut.h> // OpenGL and GLUT libraries
#include <iostream> // For C++ input/output
// Function to display the polygon
void mdisplay() {
    glClear(GL_COLOR_BUFFER_BIT); // Clear the screen
    glBegin(GL_POLYGON);         // Begin drawing a polygon
    glVertex2f(-0.5f, -0.5f);     // Define vertices
    glVertex2f(-0.5f, 0.5f);
    glVertex2f(0.5f, 0.5f);
    glVertex2f(0.5f, -0.5f);
    glEnd();
    glFlush(); // Force execution of GL commands
}

// Main function
int main(int argc, char **argv) {
    // Initialize GLUT
    glutInit(&argc, argv);
```



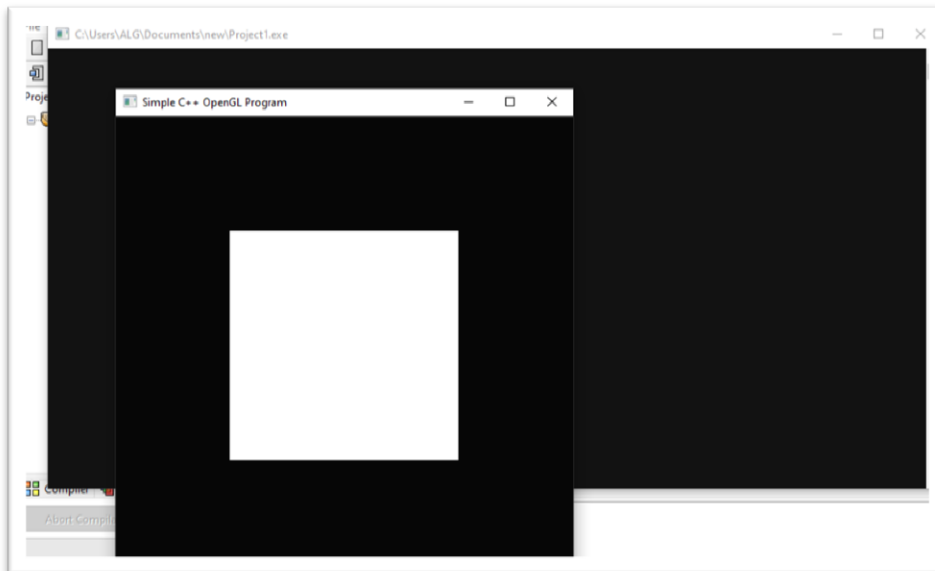
```
// Set display mode (single buffer and RGB color)
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
// Set window size
glutInitWindowSize(500, 500);
// Set window position
glutInitWindowPosition(100, 100);
// Create the window with a title
glutCreateWindow("Simple C++ OpenGL Program");
// Set the display callback function
glutDisplayFunc(mdisplay);
// Enter the GLUT main loop
glutMainLoop();
return 0; // Return success
}
```

Explanation:

1. **<GL/glut.h>**: This header includes functions from the **OpenGL Utility Toolkit (GLUT)** library, which provides tools to create windows, handle user input, and interact with OpenGL.
2. **<iostream>**: This is the standard C++ library for input and output operations, allowing you to use **std::cout**, **std::cin**, etc. It's included here for potential debugging or extra features.
3. **glClear(GL_COLOR_BUFFER_BIT)**: Clears the color buffer, which is the area of memory that stores the color of each pixel on the screen. This ensures the previous frame doesn't interfere with the current one.
4. **glBegin(GL_POLYGON)**: Begins the process of drawing a polygon (a closed shape with multiple vertices).
5. **glVertex2f(x, y)**: Specifies the vertices of the polygon in 2D space (X and Y coordinates). Each vertex is defined as a floating-point number (hence the **f** suffix).
 - The vertices here define a square centered at the origin:
 - **(-0.5, -0.5)**: Bottom-left corner
 - **(-0.5, 0.5)**: Top-left corner
 - **(0.5, 0.5)**: Top-right corner
 - **(0.5, -0.5)**: Bottom-right corner
6. **glEnd()**: Ends the polygon-drawing process. Everything between **glBegin** and **glEnd** defines the shape.
7. **glFlush()**: Forces all OpenGL commands (e.g., drawing the polygon) to execute immediately. This ensures the shape appears on the screen.
8. **glutInit(&argc, argv)**: Initializes the GLUT library. It processes command-line arguments (if any) and prepares GLUT for use.

9. **glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB):** Specifies the display mode for the window:
 - **GLUT_SINGLE:** Uses a single buffer for rendering (simpler but less flexible than double buffering).
 - **GLUT_RGB:** Uses the RGB color model (no alpha or depth buffers).
10. **glutInitWindowSize(500, 500):** Sets the window size to 500x500 pixels.
11. **glutInitWindowPosition(100, 100):** Specifies the position of the top-left corner of the window on the screen, relative to the screen's origin (top-left).
12. **glutCreateWindow("Simple C++ OpenGL Program"):** Creates a window with the title "Simple C++ OpenGL Program". This window is where all rendering occurs.
13. **glutDisplayFunc(mdisplay):** Registers the **mdisplay** function as the **display callback**. GLUT will call this function whenever the window needs to be redrawn (e.g., when resized or exposed).
14. **glutMainLoop():** Enters the **main loop**, where GLUT listens for events like drawing, resizing, or user input. The program runs indefinitely until the user closes the window.

Output looks like



Set up a GLUT window and draw some points in it