Dev cpp installation:

https://www.youtube.com/watch?v=CHFyEnlMnxg

LINKS

1) Download Dev C++ Latest Vesrion https://sourceforge.net/projects/orwe...

2) Download graphics.h library files https://drive.google.com/file/d/16xZB...

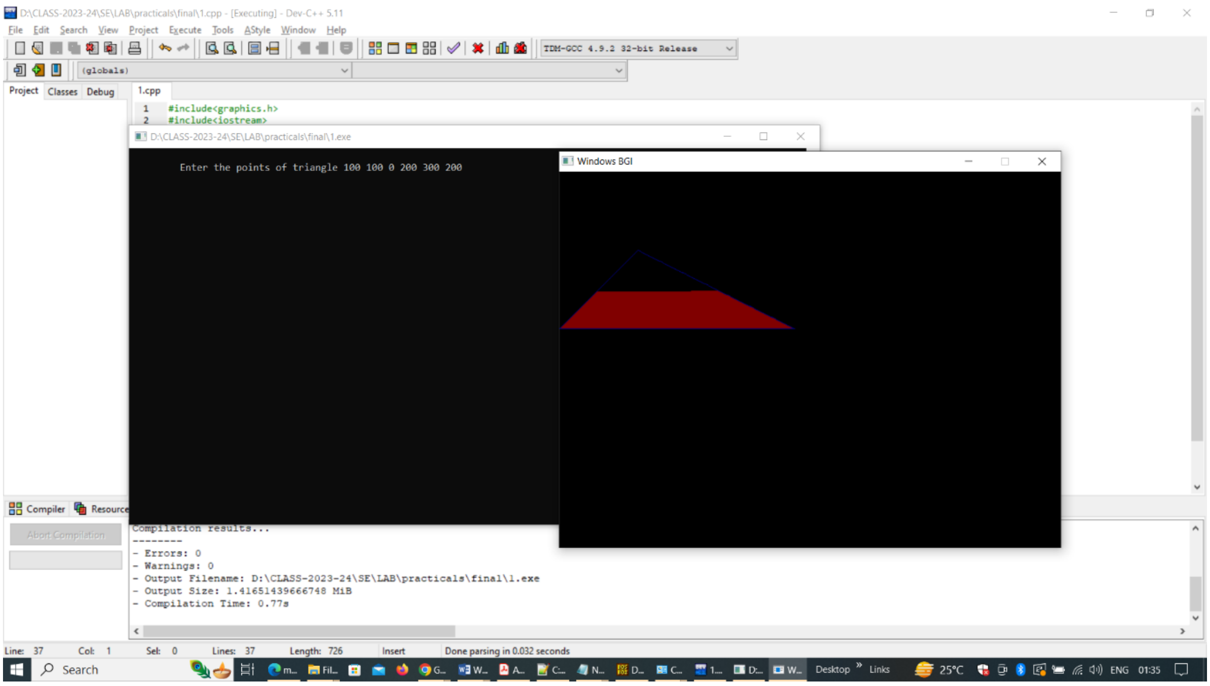
-static-libgcc -lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32

Group A 1)

Write C++ program to draw a concave polygon and fill it with desired color using scan fill algorithm. Apply the concept of inheritance.

```
#include<graphics.h>

#include<iostream>

#include<stdlib.h>

using namespace std;

void ffill(int x,int y,int o_col,int n_col)

{

int current = getpixel(x,y);

if(current==o_col)

{

delay(1);

putpixel(x,y,n_col);

ffill(x+1,y,o_col,n_col);

ffill(x-1,y,o_col,n_col);

ffill(x,y+1,o_col,n_col);

ffill(x,y-1,o_col,n_col);

}

}

int main()

{
```

```
int x1,y1,x2,y2,x3,y3,xavg,yavg;

int gdriver = DETECT,gmode;

initgraph(&gdriver,&gmode, (char*)"");

cout << " \n\t Enter the points of triangle";

setcolor(1);

cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;

xavg = (int)(x1+x2+x3)/3;

yavg = (int)(y1+y2+y3)/3;

line(x1,y1,x2,y2);

line(x2,y2,x3,y3);

line(x3,y3,x1,y1);

ffill(xavg,yavg,BLACK,RED);

getch();

return 0;

}
```



This C++ program is a simple graphics program that draws a triangle and performs a flood-fill operation to fill the interior of the triangle with a different color. Let's break down the code:

1. **Header Files:**

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

- **graphics.h**: BGI graphics library header.
- **iostream**: Input/output stream header.
- **stdlib.h**: Standard C library for general-purpose functions.

2. **Flood Fill Function:**

```
void ffill(int x, int y, int o_col, int n_col)
```

- This function implements a recursive flood-fill algorithm to fill a connected region with a new color.
- Parameters:
  - **x**, **y**: Starting coordinates for flood fill.
  - **o_col**: Original color to be replaced.
  - **n_col**: New color to fill.

3. **Flood Fill Algorithm:**

```
int current = getpixel(x, y); if (current == o_col) { delay(1); putpixel(x, y, n_col); ffill(x + 1, y, o_col, n_col); ffill(x - 1, y, o_col, n_col); ffill(x, y + 1, o_col, n_col); ffill(x, y - 1, o_col, n_col); }
```

- The flood-fill algorithm checks the current pixel color at **(x, y)** and if it matches the original color (**o_col**), it changes the color to the new color (**n_col**) and recursively calls the flood-fill function for neighboring pixels.

4. **Main Function:**

```
int main()
```

- The main function initializes the graphics window, takes input for three vertices of a triangle, draws the triangle, and performs the flood-fill operation.

5. **Graphics Initialization:**

```
int gdriver = DETECT, gmode; initgraph(&gdriver, &gmode, (char*)"");
```

- Initializes the graphics window.

6. **Input Triangle Vertices:**

```
cout << " \n\t Enter the points of triangle"; setcolor(1); cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
```

- Asks the user to enter the coordinates of three vertices of a triangle.

7. **Drawing Triangle:**

```
xavg = (int)(x1 + x2 + x3) / 3; yavg = (int)(y1 + y2 + y3) / 3; line(x1, y1, x2, y2); line(x2, y2, x3, y3); line(x3, y3, x1, y1);
```

- Calculates the average coordinates of the vertices and draws the triangle using the **line** function.

8. **Flood Fill Operation:**

```
ffill(xavg, yavg, BLACK, RED);
```

- Calls the **ffill** function to perform flood-fill starting from the average coordinates of the triangle.

9. **Close Graphics Window:**

```
getch(); return 0;
```

- Pauses for user input (waits for a key press) and then closes the graphics window.

In summary, the program draws a triangle based on user input and uses a flood-fill algorithm to fill the interior of the triangle with a different color. The flood-fill operation starts from the average coordinates of the triangle.

Group A

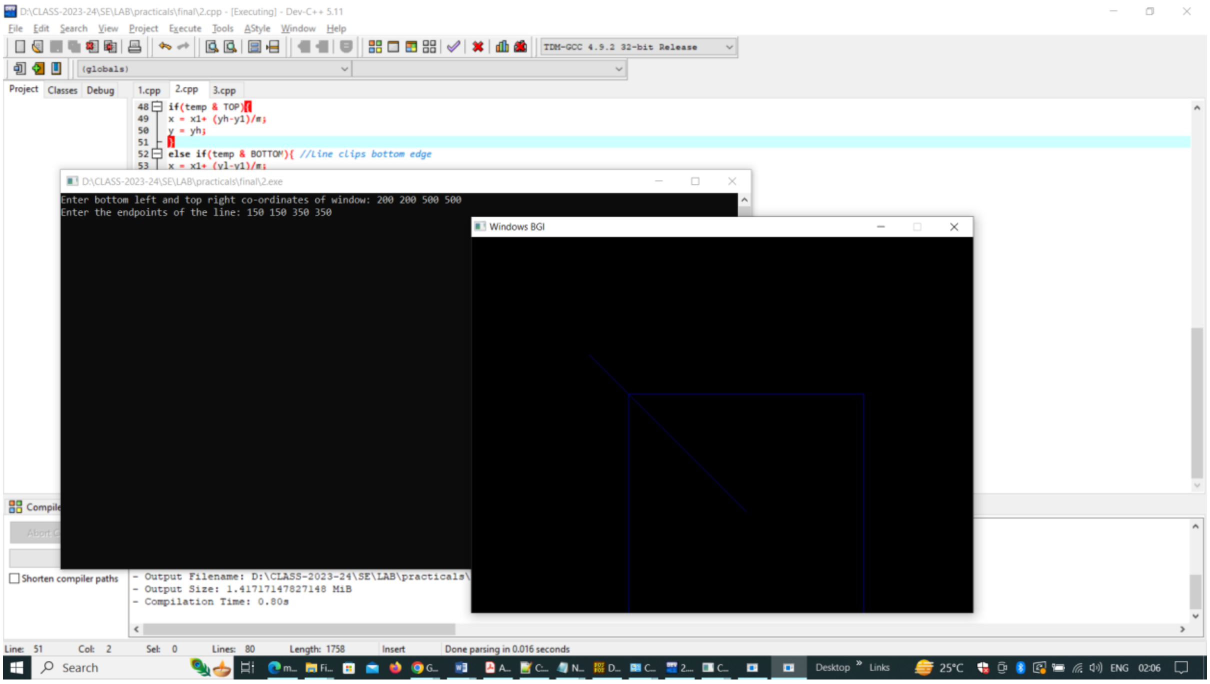
2)

Title: - Write C++ program to implement Cohen Southerland line clipping algorithm.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

Program-

```
#include<iostream>
#include<graphics.h>
using namespace std;
static int LEFT=1,RIGHT=2,BOTTOM=4,TOP=8,xl,yl,xh,yh;
int getcode(int x,int y){
int code = 0;
//Perform Bitwise OR to get outcode
if(y > yh) code |=TOP;
if(y < yl) code |=BOTTOM;
if(x < xl) code |=LEFT;
if(x > xh) code |=RIGHT;
return code;
}
int main()
{
int gdriver = DETECT,gmode;
initgraph(&gdriver,&gmode,NULL);
setcolor(BLUE);
cout<<"Enter bottom left and top right co-ordinates of window: ";
cin>>xl>>yl>>xh>>yh;
rectangle(xl,yl,xh,yh);
int x1,y1,x2,y2;
cout<<"Enter the endpoints of the line: ";
cin>>x1>>y1>>x2>>y2;
line(x1,y1,x2,y2);
getch();
int outcode1=getcode(x1,y1), outcode2=getcode(x2,y2);
int accept = 0; //decides if line is to be drawn
while(1){
float m =(float)(y2-y1)/(x2-x1);
//Both points inside. Accept line
if(outcode1==0 && outcode2==0){
accept = 1;
break;
}
//AND of both codes != 0.Line is outside. Reject line
else if((outcode1 & outcode2)!=0){
break;
}else{
```

```
int x,y;
int temp;
//Decide if point1 is inside, if not, calculate intersection
if(outcode1==0)
temp = outcode2;
else
temp = outcode1;
//Line clips top edge
if(temp & TOP){
x = x1+ (yh-y1)/m;
y = yh;
}
else if(temp & BOTTOM){ //Line clips bottom edge
x = x1+ (yl-y1)/m;
y = yl;
}else if(temp & LEFT){ //Line clips left edge
x = xl;
y = y1+ m*(xl-x1);
}else if(temp & RIGHT){ //Line clips right edge
x = xh;
y = y1+ m*(xh-x1);
}
//Check which point we had selected earlier as temp, and replace its coordinates
if(temp == outcode1){
x1 = x;
y1 = y;
outcode1 = getcode(x1,y1);
}else{
x2 = x;
y2 = y;
outcode2 = getcode(x2,y2);
}
}
}
setcolor(WHITE);
cout<<"After clipping:";
if(accept)
line(x1,y1,x2,y2);
return 0;
closegraph();
```



This C++ program performs line clipping using the Cohen-Sutherland algorithm in computer graphics. It uses the BGI (Borland Graphics Interface) library for drawing and user input.

Let's break down the code:

1. **Header Files:**

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

- **iostream**: Input/output stream header.
- **graphics.h**: BGI graphics library header.

2. **Global Constants and Variables:**

static int LEFT=1,RIGHT=2,BOTTOM=4,TOP=8,xl,yl,xh,yh;

- Global constants representing the region codes (LEFT, RIGHT, BOTTOM, TOP).
- **xl**, **yl**: Bottom-left coordinates of the clipping window.
- **xh**, **yh**: Top-right coordinates of the clipping window.

3. **Function to Get Region Code:**

int getcode(int x, int y)

- Computes the region code for a point **(x, y)** based on its position with respect to the clipping window.

4. **Main Function:**

int main()

- Initializes the graphics window, takes input for the clipping window and line endpoints, and performs the Cohen-Sutherland line clipping algorithm.

5. **Graphics Initialization:**

int gdriver = DETECT, gmode; initgraph(&gdriver, &gmode, NULL);

- Initializes the graphics window using the BGI library.

6. **Input Clipping Window:**

cout << "Enter bottom left and top right co-ordinates of window: "; cin >> xl >> yl >> xh >> yh; rectangle(xl, yl, xh, yh);

- Takes user input for the bottom-left and top-right coordinates of the clipping window and draws a rectangle to represent the clipping window.

7. **Input Line Endpoints:**

int x1, y1, x2, y2; cout << "Enter the endpoints of the line: "; cin >> x1 >> y1 >> x2 >> y2; line(x1, y1, x2, y2);

- Takes user input for the endpoints of the line and draws the original line.

8. **Cohen-Sutherland Line Clipping:**

int outcode1 = getcode(x1, y1), outcode2 = getcode(x2, y2); int accept = 0; while (1) { // ... (details explained below) }

9. **Loop for Clipping:**

while (1) { float m = (float)(y2 - y1) / (x2 - x1); // Both points inside. Accept line if (outcode1 == 0 && outcode2 == 0) { accept = 1; break; } // AND of both codes != 0. Line is outside. Reject line else if ((outcode1 & outcode2) != 0) { break; } else { // ... (details explained below) } }

10. **Calculating Intersection Points:**

int x, y; int temp; // Decide if point1 is inside, if not, calculate intersection if (outcode1 == 0) temp = outcode2; else temp = outcode1; // Line clips top edge if (temp & TOP) { x = x1 + (yh - y1) / m; y = yh; } else if (temp & BOTTOM) { // Line clips bottom edge x = x1 + (yl - y1) / m; y = yl; } else if (temp & LEFT) { // Line clips left edge x = xl; y = y1 + m * (xl - x1); } else if (temp & RIGHT) { // Line clips right edge x = xh; y = y1 + m * (xh - x1); }

11. **Updating Points and Region Codes:**

// Check which point we had selected earlier as temp, and replace its coordinates if (temp == outcode1) { x1 = x; y1 = y; outcode1 = getcode(x1, y1); } else { x2 = x; y2 = y; outcode2 = getcode(x2, y2); }

12. **Drawing Clipped Line:**

setcolor(WHITE); cout << "After clipping:"; if (accept) line(x1, y1, x2, y2);

13. **Closing Graphics Window:**

getch(); return 0; closegraph();

- Pauses for user input and closes the graphics window.

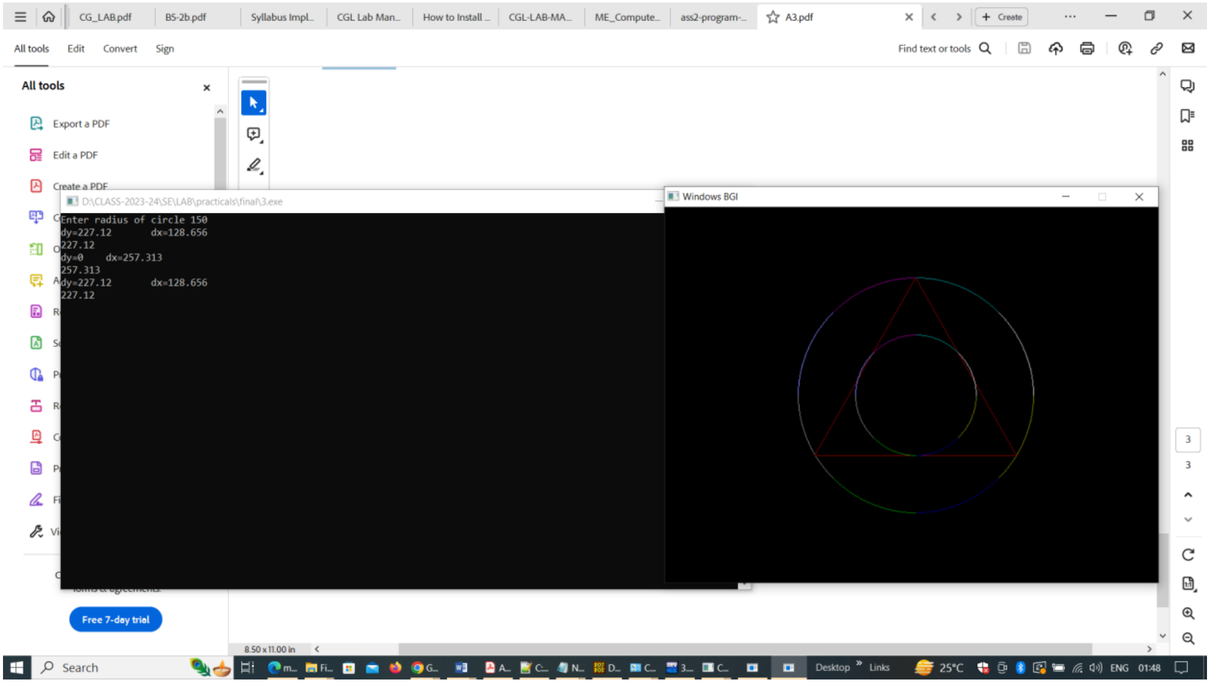
In summary, the program takes user input for a clipping window and the endpoints of a line. It then performs line clipping using the Cohen-Sutherland algorithm and displays the original and clipped lines in a graphics window. The BGI graphics library is used for graphics operations.

3: Write C++ program to draw the following pattern. Use DDA line and Bresenham drawing algorithm. Apply the concept of encapsulation.

```
#include<iostream>

#include<graphics.h>

#include <bits/stdc++.h>

using namespace std;

class algo

{

public:

void dda_line(float x1, float y1, float x2, float y2);

void bresneham_cir(int r);

};

void algo::dda_line(float x1, float y1, float x2, float y2)

{

float x,y,dx,dy,step;

int i;

//step 2

dx=abs(x2-x1);

dy=abs(y2-y1);

cout<<"dy="<<dy<<"\tdx="<<dx;

//step 3

if(dx>=dy)

step=dx;

else

step=dy;

cout<<"\n"<<step<<endl;

//step 4

float xinc=float((x2-x1)/step);

float yinc=float((y2-y1)/step);

//step 5

x=x1;
```

```
y=y1;
// outtextxy(0,0,"(0,0)");
//step 6
i=1;
while(i<=step)
{
//cout<<endl<<"\t"<<i<<"\t(x,y)=("<<x<<","<<y<<")";
putpixel(320+x,240-y,4);
x=x+xinc;
y=y+yinc;
i=i+1;
// delay(10);
}
}
void algo::bresneham_cir(int r)
{
float x,y,p;
x=0;
y=r;
p=3-(2*r);
while(x<=y)
{
putpixel(320+x,240+y,1);
putpixel(320-x,240+y,2);
putpixel(320+x,240-y,3);
putpixel(320-x,240-y,5);
putpixel(320+y,240+x,6);
putpixel(320+y,240-x,7);
putpixel(320-y,240+x,8);
putpixel(320-y,240-x,9);
x=x+1;
if(p<0)
{
```

```
p=p+4*(x)+6;
}
else
{
p=p+4*(x-y)+10;
y=y-1;
}
// delay(20);
}
}
int main()
{
algo a1;
int i;
float r,ang,r1;
initwindow(630,480);
cout<<"Enter radius of circle";
cin>>r;
a1.bresneham_cir((int)r);
ang=3.24/180;
float c=r*cos(30*ang);
float s=r*sin(30*ang);
a1.dda_line(0,r,0-c,0-s);
a1.dda_line(0-c,0-s,0+c,0-s);
a1.dda_line(0+c,0-s,0,r);
r1=s;
a1.bresneham_cir((int)r1);
getch();
closegraph();
return 0;
}
```



This C++ program is a simple graphics program using the BGI (Borland Graphics Interface) library. The program draws a circle using the Bresenham circle drawing algorithm and then draws an equilateral triangle inside it using the DDA (Digital Differential Analyzer) line drawing algorithm.

Let's break down the code:

1. **Header Files:**

   #include<iostream> #include<graphics.h> #include <bits/stdc++.h>

   - **iostream**: Input/output stream header.
   - **graphics.h**: BGI graphics library header.
   - **<bits/stdc++.h>**:

     This header is not necessary for this specific program, as it includes all the standard C++ headers. It's usually not recommended to use in production code.

2. **Namespace:**

   using namespace std;

   - This line allows using names from the **std** namespace without explicitly prefixing them with **std::**.

3. **Class Declaration:**

class algo { public: void dda_line(float x1, float y1, float x2, float y2); void bresenham_cir(int r); };

   - Declaration of a class named **algo** containing two public member functions: **dda_line** and **bresenham_cir**.

4. **DDA Line Drawing Function:**

   void algo::dda_line(float x1, float y1, float x2, float y2)

- This function uses the DDA line drawing algorithm to draw a line from **(x1, y1)** to **(x2, y2)**.

5. **Bresenham Circle Drawing Function:**

   void algo::bresenham_cir(int r)

   - This function uses the Bresenham circle drawing algorithm to draw a circle with radius **r**.

6. **Main Function:**

   int main()

   - The main function initializes the graphics window, creates an object of the **algo** class, and draws a circle and an equilateral triangle inside it.

7. **Graphics Initialization:**

   initwindow(630, 480);

   - Initializes the graphics window with a width of 630 pixels and a height of 480 pixels.

8. **User Input for Circle Radius:**

   cout << "Enter radius of circle"; cin >> r;

   - Asks the user to enter the radius of the circle.

9. **Drawing Circle:**

   a1.bresenham_cir((int)r);

   - Calls the **bresenham_cir** function to draw a circle with the specified radius.

10. **Calculating Coordinates for Equilateral Triangle:**

    ang = 3.24 / 180; float c = r * cos(30 * ang); float s = r * sin(30 * ang);

    - Calculates coordinates for an equilateral triangle inscribed inside the circle.

11. **Drawing Equilateral Triangle:**

    a1.dda_line(0, r, 0 - c, 0 - s); a1.dda_line(0 - c, 0 - s, 0 + c, 0 - s); a1.dda_line(0 + c, 0 - s, 0, r);

    - Calls the **dda_line** function three times to draw the sides of the equilateral triangle.

12. **Drawing Inner Circle:**

    r1 = s; a1.bresenham_cir((int)r1);

    - Calls the **bresenham_cir** function to draw a smaller circle inside the equilateral triangle.

13. **Closing Graphics Window:**

    getch(); closegraph(); return 0;

    - Pauses for user input (waits for a key press) and then closes the graphics window.

Please note that this code uses the BGI library, which is outdated and might not work on modern systems without a compatibility layer. Consider using more modern graphics libraries if you're working on a contemporary platform.

GROUP B

4)

/*
Title: - Write C++ program to draw 2-D object and perform following basic transformations, Scaling b) Translation c) Rotation. Apply the concept of operator overloading.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

Program-

```
#include<iostream>
#include<stdlib.h>
#include<graphics.h>
#include<math.h>
using namespace std;
class POLYGON
{
private:
int p[10][10],Trans_result[10][10],Trans_matrix[10][10];
float Rotation_result[10][10],Rotation_matrix[10][10];
float Scaling_result[10][10],Scaling_matrix[10][10];
float Shearing_result[10][10],Shearing_matrix[10][10];
int Reflection_result[10][10],Reflection_matrix[10][10];
public:
int accept_poly(int [][10]);
void draw_poly(int [][10],int);
void draw_polyfloat(float [][10],int);
void matmult(int [][10],int [][10],int,int,int,int [][10]);
void matmultfloat(float [][10],int [][10],int,int,int,float [][10]);
void shearing(int [][10],int);
void scaling(int [][10],int);
void rotation(int [][10],int);
void translation(int [][10],int);
void reflection(int [][10],int);
};
int POLYGON :: accept_poly(int p[][10])
{
int i,n;
cout<<"\n\n\t\tEnter no.of vertices:";
cin>>n;
for(i=0;i<n;i++)
{
cout<<"\n\n\t\tEnter (x,y)Co-ordinate of point P"<<i<<": ";
cin >> p[i][0] >> p[i][1];
p[i][2] = 1;
}
for(i=0;i<n;i++)
{
```

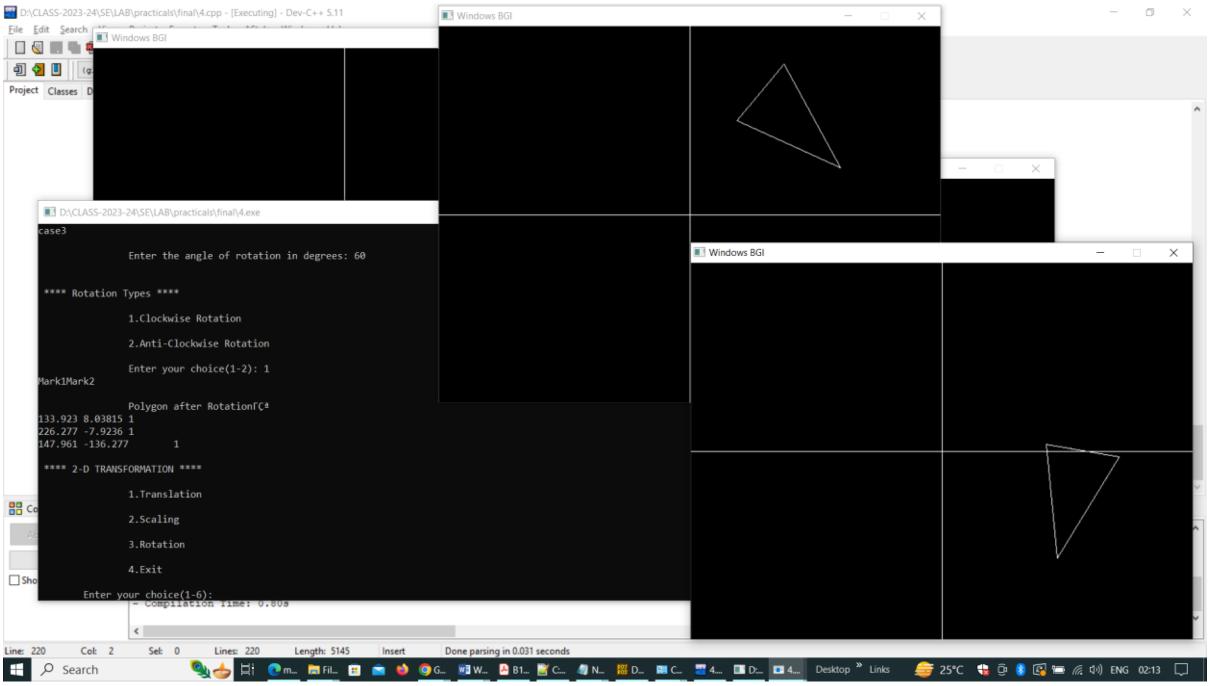
```
cout<<"\n";
for(int j=0;j<3;j++)
{
cout<<p[i][j]<<"\t";
}
}
return n;
}
void POLYGON :: draw_poly(int p[][10], int n)
{
int i,gd = DETECT,gm;
initgraph(&gd,&gm,NULL);
line(320,0,320,480);
line(0,240,640,240);
for(i=0;i<n;i++)
{
if(i<n-1)
{
line(p[i][0]+320, -p[i][1]+240, p[i+1][0]+320, -p[i+1][1]+240);
}
else
line(p[i][0]+320, -p[i][1]+240, p[0][0]+320, -p[0][1]+240);
}
delay(3000);
}
void POLYGON :: draw_polyfloat(float p[][10], int n)
{
int i,gd = DETECT,gm;
initgraph(&gd,&gm,NULL);
line(320,0,320,480);
line(0,240,640,240);
for(i=0;i<n;i++)
{
if(i<n-1)
{
line(p[i][0]+320, -p[i][1]+240, p[i+1][0]+320, -p[i+1][1]+240);
}
else
line(p[i][0]+320, -p[i][1]+240, p[0][0]+320, -p[0][1]+240);
}
//delay(8000);
}
void POLYGON :: translation(int p[10][10],int n)
{
int tx,ty,i,j; int i1,j1,k1,r1,c1,c2;
r1=n;c1=c2=3;
cout << "\n\n\t\tEnter X-Translation tx: ";
cin >> tx;
cout << "\n\n\t\tEnter Y-Translation ty: ";
cin >> ty;
for(i=0;i<3;i++)
for(j=0;j<3;j++)
Trans_matrix[i][j] = 0;
```

```
Trans_matrix[0][0] = Trans_matrix[1][1] = Trans_matrix[2][2] = 1;
Trans_matrix[2][0] = tx;
Trans_matrix[2][1] = ty;
for(i1=0;i1<10;i1++)
for(j1=0;j1<10;j1++)
Trans_result[i1][j1] = 0;
for(i1=0;i1<r1;i1++)
for(j1=0;j1<c2;j1++)
for(k1=0;k1<c1;k1++)
Trans_result[i1][j1] = Trans_result[i1][j1]+(p[i1][k1] * Trans_matrix[k1][j1]);
cout << "\n\n\t\tPolygon after Translationâ€¦";
draw_poly(Trans_result,n);
}
void POLYGON :: rotation(int p[][10],int n)
{
float type,Ang,Sinang,Cosang;
int i,j; int i1,j1,k1,r1,c1,c2;
r1=n;c1=c2=3;
cout << "\n\n\t\tEnter the angle of rotation in degrees: ";
cin >> Ang;
cout << "\n\n **** Rotation Types ****";
cout << "\n\n\t\t1.Clockwise Rotation \n\n\t\t2.Anti-Clockwise Rotation ";
cout << "\n\n\t\tEnter your choice(1-2): ";
cin >> type;
Ang = (Ang * 6.2832)/360;
Sinang = sin(Ang);
Cosang = cos(Ang);
cout<<"Mark1";
for(i=0;i<3;i++)
for(j=0;j<3;j++)
Rotation_matrix[i][j] = 0;
cout<<"Mark2";
Rotation_matrix[0][0] = Rotation_matrix[1][1] = Cosang;
Rotation_matrix[0][1] = Rotation_matrix[1][0] = Sinang;
Rotation_matrix[2][2] = 1;
if(type == 1)
Rotation_matrix[0][1] = -Sinang;
else
Rotation_matrix[1][0] = -Sinang;
for(i1=0;i1<10;i1++)
for(j1=0;j1<10;j1++)
Rotation_result[i1][j1] = 0;
for(i1=0;i1<r1;i1++)
for(j1=0;j1<c2;j1++)
for(k1=0;k1<c1;k1++)
Rotation_result[i1][j1] = Rotation_result[i1][j1]+(p[i1][k1] *
Rotation_matrix[k1][j1]);
cout << "\n\n\t\tPolygon after Rotationâ€¦";
for(i=0;i<n;i++)
{
cout<<"\n";
for(int j=0;j<3;j++)
{
```

```
cout<<Rotation_result[i][j]<<"\t";
}
}
draw_polyfloat(Rotation_result,n);
}
void POLYGON :: scaling(int p[][10],int n)
{
float Sx,Sy;
int i,j; int i1,j1,k1,r1,c1,c2;
r1=n;c1=c2=3;
cout<<"\n\n\t\tEnter X-Scaling Sx: ";
cin>>Sx;
cout<<"\n\n\t\tEnter Y-Scaling Sy: ";
cin>>Sy;
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
Scaling_matrix[i][j] = 0;
}
}
Scaling_matrix[0][0] = Sx;
Scaling_matrix[0][1] = 0;
Scaling_matrix[0][2] = 0;
Scaling_matrix[1][0] = 0;
Scaling_matrix[1][1] = Sy;
Scaling_matrix[1][2] = 0;
Scaling_matrix[2][0] = 0;
Scaling_matrix[2][1] = 0;
Scaling_matrix[2][2] = 1;
for(i1=0;i1<10;i1++)
for(j1=0;j1<10;j1++)
Scaling_result[i1][j1] = 0;
for(i1=0;i1<r1;i1++)
for(j1=0;j1<c2;j1++)
for(k1=0;k1<c1;k1++)
Scaling_result[i1][j1] = Scaling_result[i1][j1]+(p[i1][k1] *
Scaling_matrix[k1][j1]);
cout<<"\n\n\t\tPolygon after Scalingâ€¦";
draw_polyfloat(Scaling_result,n);
}
int main()
{
int ch,n,p[10][10];
POLYGON p1;
cout<<"\n\n **** 2-D TRANSFORMATION ****";
n= p1.accept_poly(p);
cout <<"\n\n\t\tOriginal Polygon â€¦";
p1.draw_poly(p,n);
do
{
int ch;
cout<<"\n\n **** 2-D TRANSFORMATION ****";
```

```
cout<<"\n\n\t\t1.Translation \n\n\t\t2.Scaling \n\n\t\t3.Rotation \n\n\t\t4.Exit";
cout<<"\n\n\tEnter your choice(1-6):";
cin>>ch;
switch(ch)
{
case 1:
//cout<<"case1";
p1.translation(p,n);
break;
case 2:
cout<<"case2";
p1.scaling(p,n);
break;
case 3:
cout<<"case3";
p1.rotation(p,n);
break;
case 4:
exit(0);
}
}while(1);
return 0;
}
/*Output:
**** 2-D TRANSFORMATION ****
Enter no.of vertices:3
Enter (x,y)Co-ordinate of point P0: 60
120
Enter (x,y)Co-ordinate of point P1: 120
192
Enter (x,y)Co-ordinate of point P2: 192
60
60 120 1
120 192 1
192 60 1
Original Polygon ΓÇª
**** 2-D TRANSFORMATION ****
1.Translation
2.Scaling
3.Rotation
4.Exit
Enter your choice(1-6):1
Enter X-Translation tx: 20
Enter Y-Translation ty: 30
Polygon after TranslationΓÇª
**** 2-D TRANSFORMATION ****
1.Translation
2.Scaling
3.Rotation
4.Exit
Enter your choice(1-6):2
case2
Enter X-Scaling Sx: 20
```

```
Enter Y-Scaling Sy: 30
Polygon after ScalingΓÇª
**** 2-D TRANSFORMATION ****
1.Translation
2.Scaling
3.Rotation
4.Exit
Enter your choice(1-6):3
case3
Enter the angle of rotation in degrees: 60
**** Rotation Types ****
1.Clockwise Rotation
2.Anti-Clockwise Rotation
Enter your choice(1-2): 1
Mark1Mark2
Polygon after RotationΓÇª
133.923 8.03815 1
226.277 -7.9236 1
147.961 -136.277 1
**** 2-D TRANSFORMATION ****
1.Translation
2.Scaling
3.Rotation
4.Exit
        Enter your choice(1-6):4 */
```



This C++ program is an implementation of 2D geometric transformations, including translation, scaling, and rotation, applied to a polygon. The transformations are performed using matrix multiplication.

Let's break down the code:

1. **Header Files and Namespace:**

   #include<iostream> #include<stdlib.h> #include<graphics.h> #include<math.h> using namespace std;

   - **iostream**: Input/output stream header.
   - **stdlib.h**: Standard C library for general-purpose functions.
   - **graphics.h**: BGI graphics library header.
   - **math.h**: Header for mathematical functions.

2. **Polygon Class:**

   class POLYGON { // private members for matrices and transformation results public: // public member functions for accepting, drawing, and performing transformations on the polygon };

3. **Accepting Polygon:**

   int accept_poly(int p[][10]);

   - Takes input for the polygon's vertices and returns the number of vertices.

4. **Drawing Polygon:**

   void draw_poly(int p[][10], int n); void draw_polyfloat(float p[][10], int n);

   - Draws the original and transformed polygons.

5. **Matrix Multiplication:**

   void matmult(int [][10], int [][10], int, int, int, int [][10]); void matmultfloat(float [][10], int [][10], int, int, int, float [][10]);

   - Functions for matrix multiplication (integer and floating-point versions).

6. **Transformation Functions:**

   void translation(int [][10], int); void scaling(int [][10], int); void rotation(int [][10], int);

   - Functions for translation, scaling, and rotation transformations.

7. **Main Function:**

   int main()

   - Initializes the graphics window, accepts the polygon, and provides a menu for selecting transformations.

8. **Menu for Transformations:**

   do { // Display menu and switch based on user choice } while (1);

9. **Switch Cases for Transformations:**

   switch(ch) { case 1: p1.translation(p, n); break; case 2: p1.scaling(p, n); break; case 3: p1.rotation(p, n); break; case 4: exit(0); }

In summary, the program allows the user to input a polygon and choose between translation, scaling, and rotation transformations. The transformations are then applied, and the original and transformed polygons are displayed in a graphics window. The BGI graphics library is used for drawing.

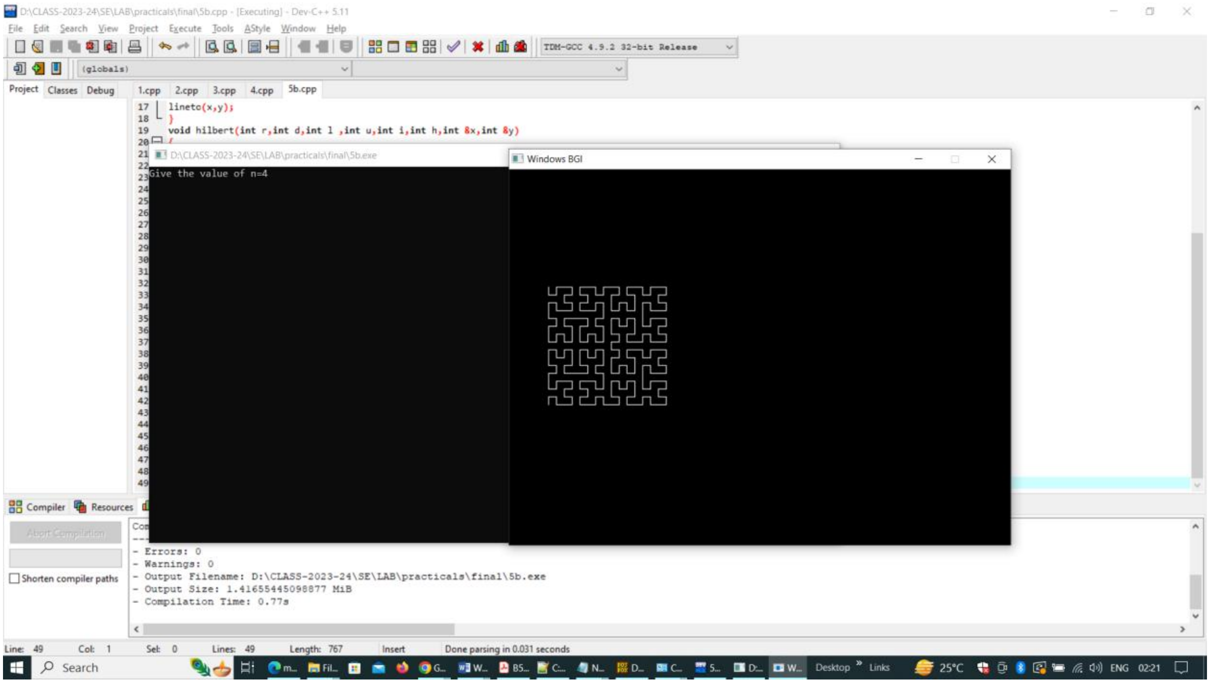
/* 5 b)
Title: - Write C++ program to generate Hilbert curve using concept of fractals.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

Program-

```
#include<iostream>
#include<graphics.h>
#include<math.h>
#include<cstdlib>
using namespace std;
void move(int j, int h, int &x,int &y)
{
if(j==1)
y-=h;
else
if(j==2)
x+=h;
else if(j==3)
y+=h;
else if(j==4)
x-=h;
lineto(x,y);
}
void hilbert(int r,int d,int l ,int u,int i,int h,int &x,int &y)
{
if(i>0)
{
i--;
hilbert(d,r,u,l,i,h,x,y);
move(r,h,x,y);
hilbert(r,d,l,u,i,h,x,y);
move(d,h,x,y);
hilbert(r,d,l,u,i,h,x,y);
move(l,h,x,y);
hilbert(u,l,d,r,i,h,x,y);
}
}
int main()
{
int n,x1,y1;
int x0=50,y0=150,x,y,h=10,r=2,d=3,l=4,u=1;
cout<<"Give the value of n=";
cin>>n;
x=x0;
y=y0;
int driver=DETECT,mode=0;
initgraph(&driver,&mode,NULL);
moveto(x,y);
hilbert(r,d,l,u,n,h,x,y);
```

```
delay(10000);
closegraph();
return 0;
}
```

/*Output:-



This C++ program uses the graphics library to draw a Hilbert curve. A Hilbert curve is a space-filling fractal curve that visits every point in a square grid exactly once. Let's break down the code:

Functions:

1. **move Function:**
    - Takes parameters **j** (direction), **h** (step size), **x**, and **y**.
    - Moves the current position **(x, y)** based on the specified direction.
    - Draws a line to the new position.
2. **hilbert Function:**
    - Implements the Hilbert curve recursively using four directions: **r** (right), **d** (down), **l** (left), and **u** (up).
    - Parameters include the current recursion depth **i**, step size **h**, and current position **(x, y)**.
    - The function is called recursively to draw each part of the Hilbert curve, and it moves to new positions using the **move** function.

main Function:

1. **Variable Initialization:**

- Initializes variables **n** (order of the Hilbert curve), **x1**, **y1**, **x0**, **y0**, **x**, **y**, **h**, **r**, **d**, **l**, and **u**.
- The starting point is set to **(x0, y0)**.

2. **Graphics Initialization:**
   - Initializes the graphics mode using the **initgraph** function.
3. **Input:**
   - Asks the user to input the order **n** of the Hilbert curve.
4. **Drawing the Hilbert Curve:**
   - Calls the **hilbert** function to draw the Hilbert curve.
   - The initial direction is set to **r** (right).
   - The **delay(10000)** function introduces a delay to keep the graphics window open for a while.
5. **Graphics Cleanup:**
   - Closes the graphics window using the **closegraph** function.

How the Hilbert Curve is Drawn:

- The **hilbert** function is a recursive function that draws a Hilbert curve by subdividing each segment into smaller segments.
- At each recursion level, the curve is drawn by making four recursive calls in the order **d, r, l, u** (down, right, left, up), and the **move** function is used to update the current position and draw lines.

Example:

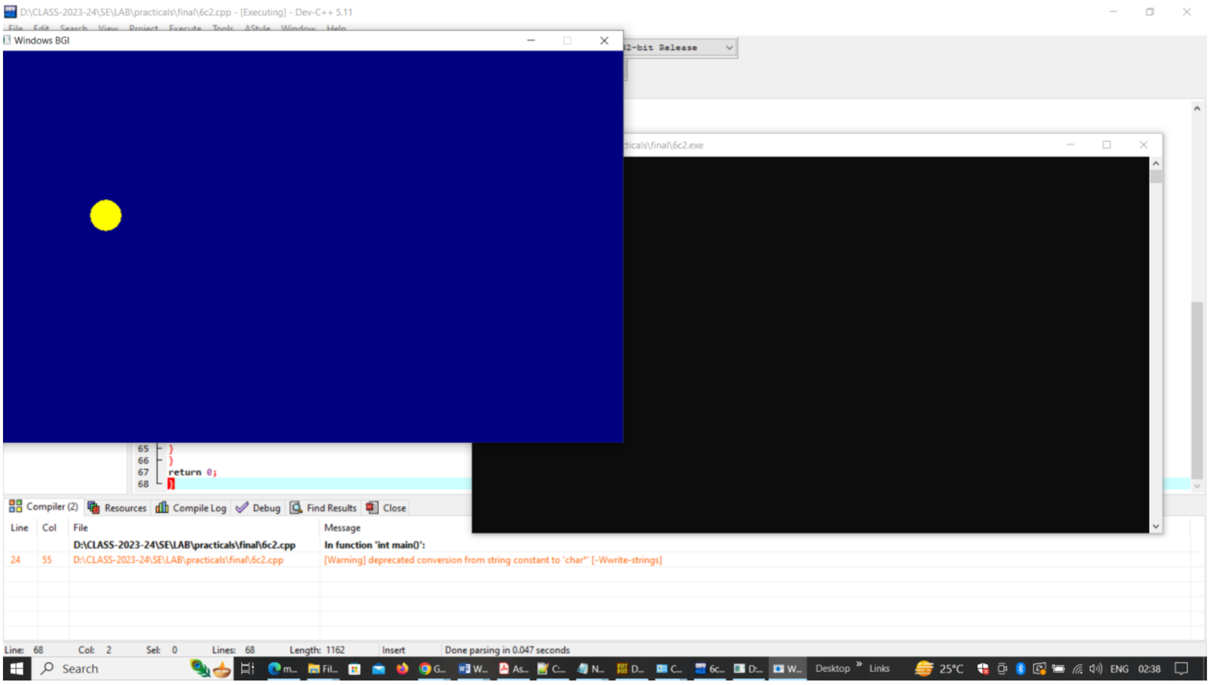
Suppose the user inputs **n = 3**. The Hilbert curve will be drawn with a third-order curve, and the order of recursive calls will be based on the **d, r, l, u** pattern. The final result is displayed in a graphics window.

/* 6 c)
Title: - Write OpenGL Program to draw Sunrise and Sun Set.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

```
#include<iostream>

#include<graphics.h>

#include<cstdlib>

#include<dos.h>

#include<cmath>

using namespace std;

int main()

{

initwindow(800,500);

int x0,y0;

int gdriver = DETECT,gmode,errorcode;

int xmax,ymax;

errorcode=graphresult();

if(errorcode!=0)

{

cout<<"Graphics error:"<<grapherrormsg(errorcode);

cout<<"Press any ket to halt";

exit(1);

}

int i,j;

setbkcolor(BLUE);

setcolor(RED);

rectangle(0,0,getmaxx(),getmaxy());

outtextxy(250,240,"::::PRESS ANY KEY TO CONTINUE::::");

while(!kbhit());

for(i=50,j=0;i<=250,j<=250;i+=5,j+=5)

{
```

```
delay(120);
cleardevice();
if(i<=150)
{
setcolor(YELLOW);
setfillstyle(1,YELLOW);
fillellipse(i,300-j,20,20);
}
else
{
setcolor(GREEN^RED);
setfillstyle(1,GREEN^RED);
fillellipse(i,300-j,20,20);
}
}
delay(1000);
cleardevice();
setcolor(RED);
setfillstyle(1,RED);
fillellipse(300,50,20,20);
delay(150);
int k,l;
for(k=305,l=55;k<=550,l<=300;k+=5,l+=5)
{
delay(120);
cleardevice();
if(k<=450)
{
setcolor(GREEN^RED);
setfillstyle(1,GREEN^RED);
fillellipse(k,l,20,20);
}
```

```
else
{
setcolor(YELLOW);
setfillstyle(1,YELLOW);
fillellipse(k,l,20,20);
}
}
return 0;
}
```



This C++ program uses the graphics library to create a simple animation. Below is an explanation of the code:

main Function:

1. **Graphics Initialization:**
    - **initwindow(800, 500)** creates a graphics window of size 800x500 pixels.
    - **gdriver**, **gmode**, and **errorcode** are variables for graphics driver, graphics mode, and error code.
    - **errorcode = graphresult()** checks for errors during graphics initialization.
    - If there is an error, an error message is displayed, and the program exits.
2. **Drawing the Initial Rectangle and Text:**
    - **setbkcolor(BLUE)** sets the background color of the window to blue.
    - **setcolor(RED)** sets the drawing color to red.

- **rectangle(0, 0, getmaxx(), getmaxy())** draws a red rectangle around the window.
- **outtextxy(250, 240, "::::PRESS ANY KEY TO CONTINUE:::::")** displays text in the middle of the window.
- The program waits for a keypress (**while (!kbhit())**) before proceeding.

3. **Animating Ellipses:**
    - Two loops animate ellipses moving diagonally.
    - The first loop (**for(i=50, j=0; i<=250, j<=250; i+=5, j+=5)**) animates yellow and red ellipses from bottom-left to top-right.
        - The **delay(120)** introduces a delay for each frame.
        - Ellipses are drawn using **fillellipse** and change color as they move.
    - After a delay, a single red ellipse appears at a specific location (**fillellipse(300, 50, 20, 20)**).
    - The second loop (**for(k=305, l=55; k<=550, l<=300; k+=5, l+=5)**) animates green and yellow ellipses from top-right to bottom-left.
4. **Closing the Graphics Window:**
    - After the animation, the program returns 0, effectively ending the program.

How the Animation Works:

- The program initializes a graphics window, draws a rectangle, and displays a text message.
- It waits for a keypress before animating ellipses.
- Ellipses move diagonally, changing colors, and then a single red ellipse appears.
- Another set of ellipses moves diagonally, changing colors.
- The graphics window closes when the program ends.

Note:

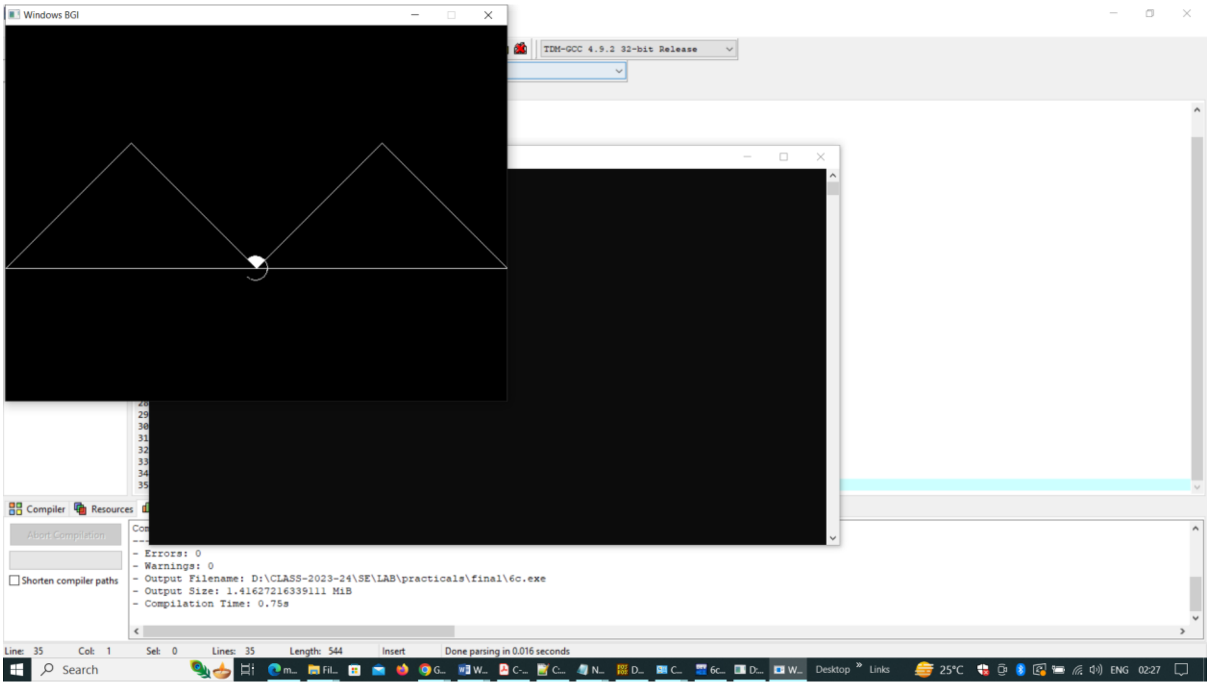
- The program uses the Turbo C++ graphics library, and the functions like **initwindow**, **rectangle**, **outtextxy**, and **fillellipse** are specific to this library.
- The use of graphics libraries and functions like **fillellipse** may not be supported in some modern IDEs or compilers.

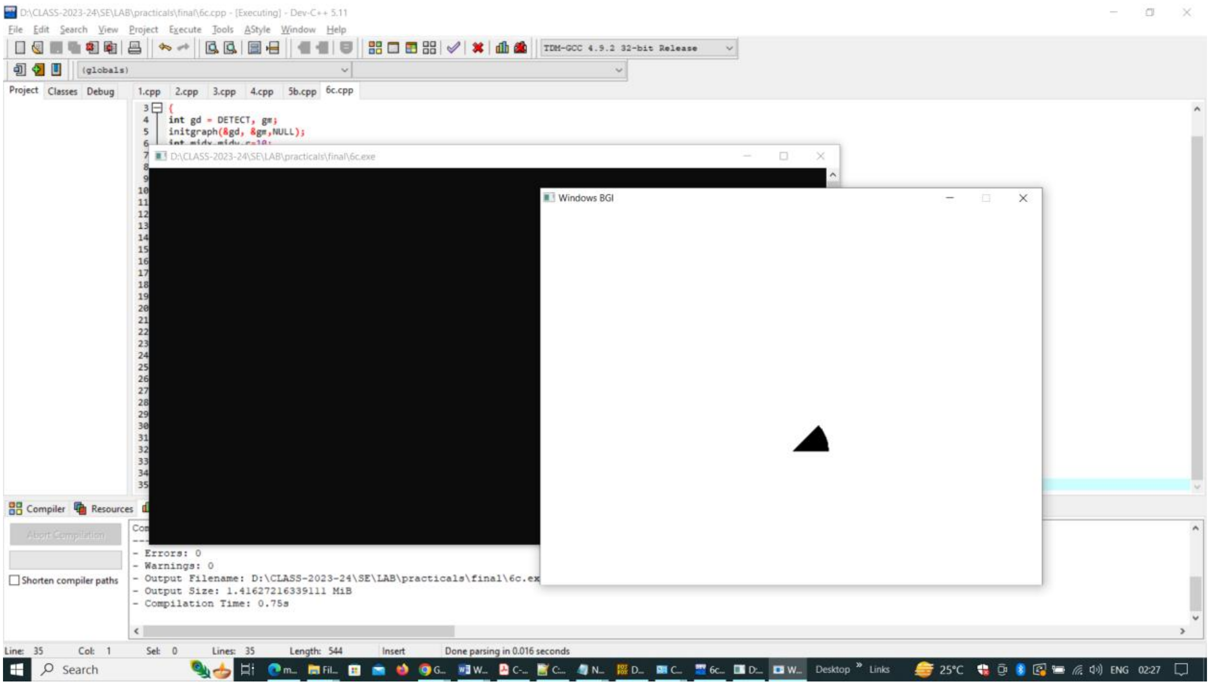
/* 6 c2)
Title: - Write OpenGL Program to draw Sunrise and Sun Set.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

Program-

```
#include<graphics.h>
int main()
{
int gd = DETECT, gm;
initgraph(&gd, &gm,NULL);
int midx,midy,r=10;
midx=getmaxx()/2;
while(r<=50)
{
cleardevice();
setcolor(WHITE);
line(0,310,160,150);
line(160,150,320,310);
line(320,310,480,150);
line(480,150,640,310);
line(0,310,640,310);
arc(midx,310,225,133,r);
floodfill(midx,300,15);
if(r>20)
{
setcolor(7);
floodfill(2,2,15);
setcolor(6);
floodfill(150,250,15);
floodfill(550,250,15);
setcolor(2);
floodfill(2,450,15);
}
delay(50);
r+=2;
}
getch();
closegraph();
}
```

/*Output:-





Here's a breakdown of the code:

main Function:

1. **Graphics Initialization:**

    - Initializes the graphics mode using the **initgraph** function.
    - **gd** is the graphics driver, and **gm** is the graphics mode.
    - **getmaxx()** is used to find the maximum x-coordinate of the graphics window.
    - **midx** is set to the midpoint of the graphics window.

- A loop is set up to create the animation.

2. **Drawing and Animation Loop:**
    - The loop continues until the radius **r** of the ball is greater than 50.
    - **cleardevice()** clears the graphics window in each iteration.
    - Lines are drawn to create a triangular shape resembling a roof.
    - **arc()** draws an arc representing the bouncing ball.
    - **floodfill()** is used to fill the ball with a color. The color changes as the ball reaches different positions.
        - Initially (when **r <= 20**), the ball is filled with color 15 (white).
        - When **r > 20**, additional colors (7, 6, 2) are used to fill specific areas, creating a colorful animation.
    - The **delay(50)** introduces a short delay to control the speed of the animation.
    - The radius **r** is incremented by 2 in each iteration to make the ball grow.
3. **Graphics Cleanup:**
    - **getch()** waits for a key press before closing the graphics window.
    - **closegraph()** closes the graphics mode.

How the Animation Works:

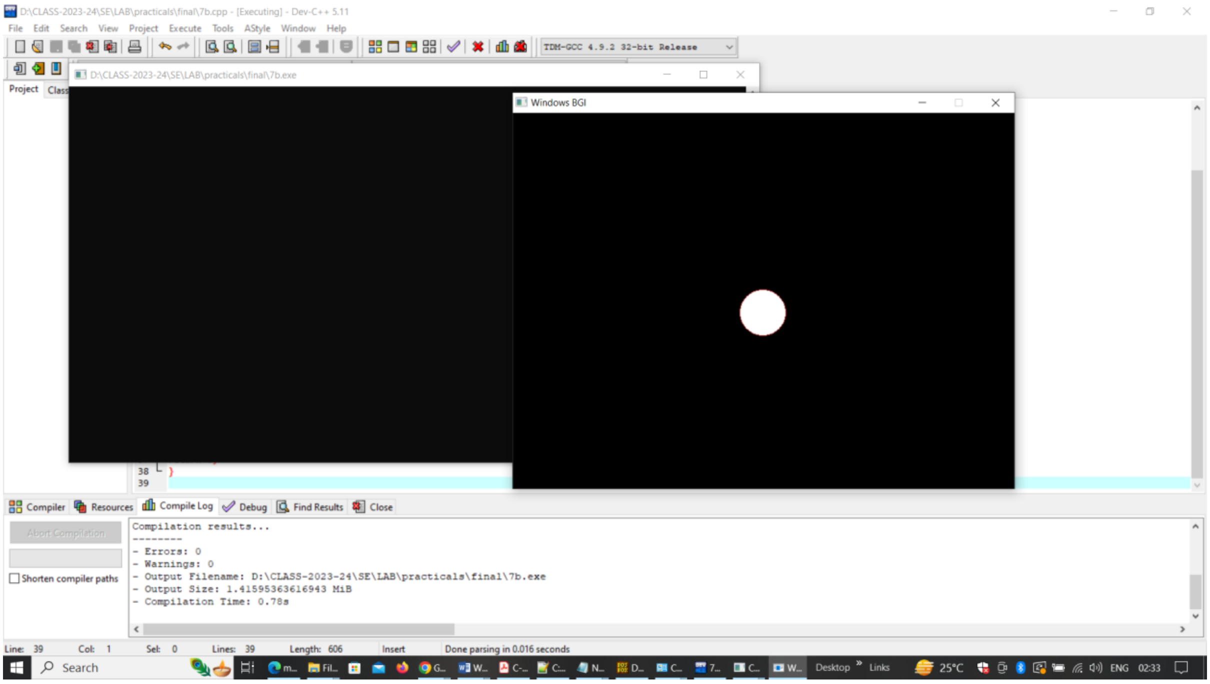
- The program draws a bouncing ball inside a triangular shape.
- The ball's position and appearance change during the animation.
- As the ball grows (radius **r** increases), different areas of the ball are filled with different colors.
- The triangular shape is redrawn in each iteration of the loop to create the appearance of the ball bouncing off the walls.

/* 7 b)
Title: - Write C++ program to generate Bouncing ball. Apply the concept of polymorphism.
Roll No:-
Class:-SE Computer
Sub:-OOPL & CGL
Date:-
****
*/

Program-

```
#include <iostream>
#include <cstdlib>
#include <graphics.h>
using namespace std;
int main()
{
int gd = DETECT, gm;
int i, x, y, flag=0;
initgraph(&gd, &gm,NULL);
/* get mid positions in x and y-axis */
x = getmaxx()/2;
y = 30;
while (1)
{
if(y >= getmaxy()-30 || y <= 30)
flag = !flag;
/* draws the gray board */
setcolor(RED);
//setfillstyle(SOLID_FILL, RED);
circle(x, y, 30);
floodfill(x, y, RED);
/* delay for 50 milli seconds */
delay(50);
/* clears screen */
cleardevice();
if(flag)
{
y = y + 5;
}
else
{
y = y - 5;
}
}
delay(5000);
closegraph();
return 0;
}
```

/*Output:-



This C++ program uses the graphics library to create a simple animation of a bouncing ball within a window. Below is an explanation of the code:

main Function:

1. **Graphics Initialization:**
    - **gd** is the graphics driver, and **gm** is the graphics mode.
    - **initgraph(&gd, &gm, NULL)** initializes the graphics mode using the Turbo C++ graphics library.
2. **Variables:**
    - **i**, **x**, **y**: Variables to control the position of the bouncing ball.
    - **flag**: A flag variable to determine the direction of the ball's movement.
3. **Bouncing Ball Animation Loop:**
    - The **while(1)** loop runs indefinitely for continuous animation.
    - Inside the loop:
        - The condition **y >= getmaxy()-30 || y <= 30** checks whether the ball has reached the bottom or top of the window. If so, it toggles the **flag** variable, changing the direction of the ball.
        - **setcolor(RED)** sets the color of the circle to red.
        - **circle(x, y, 30)** draws a circle at coordinates (x, y) with a radius of 30 pixels.
        - **floodfill(x, y, RED)** fills the circle with the red color.
        - **delay(50)** introduces a short delay to control the speed of the animation.
        - **cleardevice()** clears the graphics window in each iteration to erase the previously drawn circle.

- The position **y** of the circle is updated based on the **flag** variable, causing the ball to move up or down.

4. **Graphics Cleanup:**

    - **delay(5000)** introduces a delay of 5000 milliseconds (5 seconds) before closing the graphics window.
    - **closegraph()** closes the graphics mode.

How the Animation Works:

- The program creates a graphics window and draws a red circle representing a bouncing ball.
- The ball moves up and down within the window.
- When the ball reaches the top or bottom of the window, its direction is reversed (toggled by the **flag** variable).
- The animation continues indefinitely until the user manually closes the graphics window.

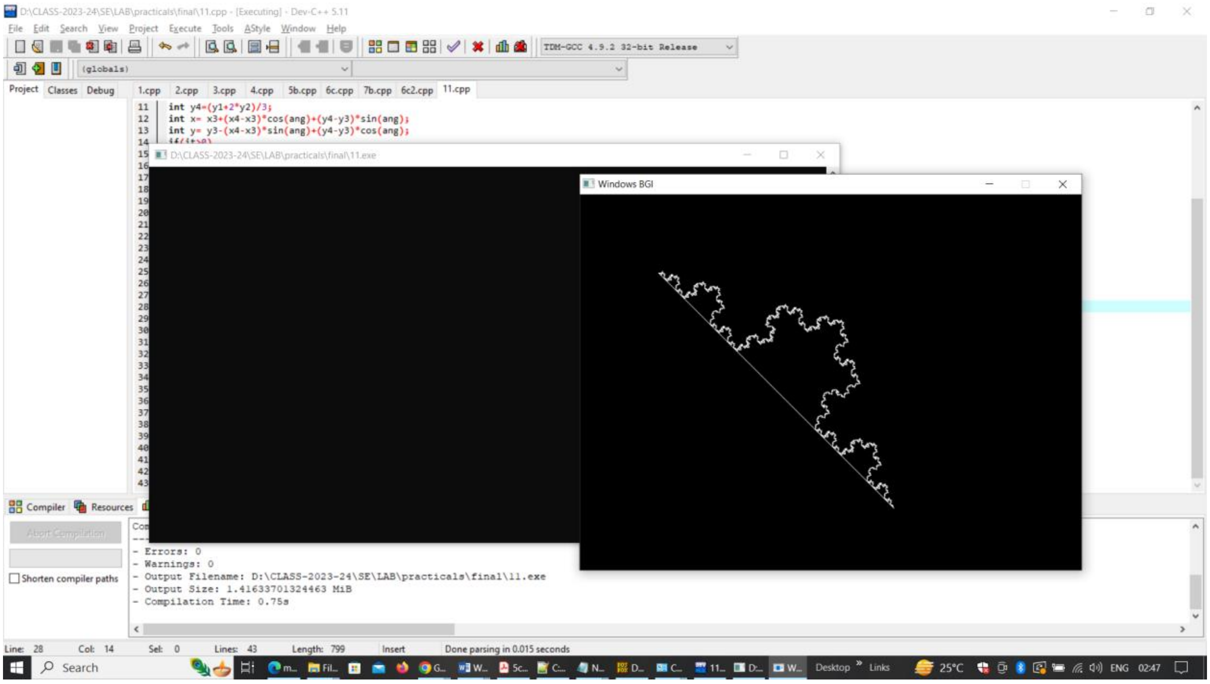
Note:

- This code is dependent on the Turbo C++ graphics library, and it might not work in modern compilers or environments.
- The use of graphics libraries and functions like **circle** and **floodfill** is specific to graphics programming and may not be supported in some modern IDEs or compilers.

5 C) Write C++ program to generate fractal patterns by using Koch curves.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#include<dos.h>
void koch(int x1,int y1,int x2,int y2,int it){
float ang=60*M_PI/180;
int x3=(2*x1+x2)/3;
int y3=(2*y1+y2)/3;
int x4=(x1+2*x2)/3;
int y4=(y1+2*y2)/3;
int x= x3+(x4-x3)*cos(ang)+(y4-y3)*sin(ang);
int y= y3-(x4-x3)*sin(ang)+(y4-y3)*cos(ang);
if(it>0)
{
koch(x1,y1,x3,y3,it-1);
koch(x3,y3,x,y,it-1);
koch(x,y,x4,y4,it-1);
koch(x4,y4,x2,y2,it-1);
}
else{
//delay(100);
line(x1,y1,x3,y3);
//delay(100);
line(x3,y3,x,y);
//delay(100);
line(x,y,x4,y4);
//delay(100);
line(x4,y4,x2,y2);
```

```
//delay(100);
}
}
int main()
{
int gd = DETECT,gm;
initgraph(&gd,&gm,NULL);
int x1=100,y1=100,x2=400,y2=400;
line(100,100,400,400);
//delay(50);
koch(x1,y1,x2,y2,5);
getch();
return 0;
}
```



This C code generates a Koch snowflake fractal using the graphics library. Below is an explanation of the code:

koch Function:

cCopy code

void koch(int x1, int y1, int x2, int y2, int it)

This recursive function generates the Koch curve (a specific type of fractal) between the points **(x1, y1)** and **(x2, y2)**.

- **x1**, **y1**: Coordinates of the starting point.
- **x2**, **y2**: Coordinates of the ending point.
- **it**: Number of iterations or recursion depth.

main Function:

cCopy code

int main()

- **Graphics Initialization:**
    - **int gd = DETECT, gm;**: Initializes the graphics driver and mode.
    - **initgraph(&gd, &gm, NULL);**: Initializes the graphics system.
- **Drawing a Line:**
    - **int x1 = 100, y1 = 100, x2 = 400, y2 = 400;**: Defines the coordinates of the initial line.
    - **line(100, 100, 400, 400);**: Draws the initial line.
- **Generating Koch Snowflake:**
    - **koch(x1, y1, x2, y2, 5);**: Calls the **koch** function to generate the Koch snowflake.
        - The last parameter (**5**) specifies the recursion depth (number of iterations).
        - The depth determines the level of detail in the fractal.

koch Function Details:

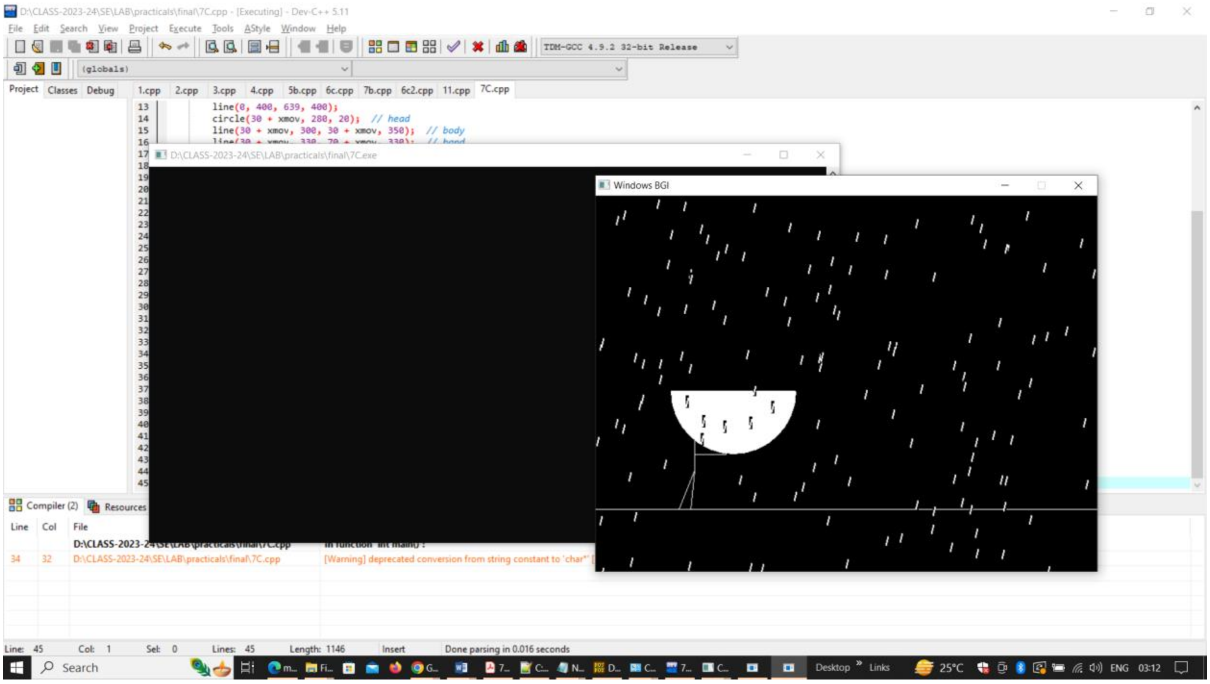
- **Base Case:**
    - If **it** (iteration) is zero, the function draws lines between the given points.
    - Lines are drawn with the **line** function.
- **Recursive Case:**
    - If **it** is greater than zero, the function calculates intermediate points (**x3**, **y3**, **x**, **y**, **x4**, **y4**) and calls itself recursively for four segments.
    - The angle **ang** is 60 degrees in radians.
    - The recursive calls generate smaller segments, creating the Koch curve.

Notes:

- The **delay** function is commented out. Uncommenting it introduces delays between drawing segments, providing a visual effect.
- The program uses the Turbo C graphics library functions like **initgraph** and **line**, which may not be supported in modern IDEs or compilers.
- The Koch snowflake is a well-known fractal, and the recursive process creates self-similarity at different scales.

7 c) Write C++ program to draw man walking in the rain with an umbrella. Apply the concept ofpolymorphism.

```
#include <iostream>
#include <graphics.h>
#include <cstdlib>
#include <ctime>
#include <dos.h>

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

   for (xmov = 1; xmov < 200; xmov = xmov + 5) {
      line(0, 400, 639, 400);
      circle(30 + xmov, 280, 20);  // head
      line(30 + xmov, 300, 30 + xmov, 350);  // body
      line(30 + xmov, 330, 70 + xmov, 330);  // hand

      if (xmov % 2 == 0) {
         line(30 + xmov, 350, 25 + xmov, 400);  // left leg
         line(30 + xmov, 350, 10 + xmov, 400);  // right leg
      } else {
         line(30 + xmov, 350, 25 + xmov, 400);
         delay(25);
      }

      line(70 + xmov, 250, 70 + xmov, 330);  // umbrella
      pieslice(80 + xmov, 250, 180, 0, 80);

      srand(time(NULL));  // Seed for rand() based on current time

      for (int i = 0; i <= 300; i++) {
         x = rand() % 800;
         y = rand() % 800;
         outtextxy(x, y, "/");
      }

      delay(600);
      cleardevice();
   }
   getch();
   closegraph();
   return 0;
}
```



This C++ program uses the graphics library to create a simple animation. It draws a figure moving horizontally, simulating walking, and displays a raining effect using randomly generated "/" characters. Here's a breakdown of the code:

1. **Include Headers:**

#include <iostream> #include <graphics.h> #include <cstdlib> #include <ctime> #include <dos.h>

These are standard C++ and graphics library headers.

2. **Main Function:**

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

Initializes the graphics system.

3. **Variable Declarations:**

int xmov, x, y;

Variables for animation control and raindrop positions.

4. **Animation Loop:**

for (xmov = 1; xmov < 200; xmov = xmov + 5) {

Loop for the horizontal movement of the walking figure.

5. **Drawing Figure:**

line(0, 400, 639, 400); circle(30 + xmov, 280, 20); // head line(30 + xmov, 300, 30 + xmov, 350); // body line(30 + xmov, 330, 70 + xmov, 330); // hand

Draws the figure (head, body, hand).

6. **Leg Movement:**

if (xmov % 2 == 0) { line(30 + xmov, 350, 25 + xmov, 400); // left leg line(30 + xmov, 350, 10 + xmov, 400); // right leg } else { line(30 + xmov, 350, 25 + xmov, 400); delay(25); }

Simulates the walking motion with alternating leg movements.

7. **Umbrella and Raindrops:**

line(70 + xmov, 250, 70 + xmov, 330); // umbrella pieslice(80 + xmov, 250, 180, 0, 80); srand(time(NULL)); // Seed for rand() based on current time for (int i = 0; i <= 300; i++) { x = rand() % 800; y = rand() % 800; outtextxy(x, y, "/"); }

Draws an umbrella and simulates raindrops with randomly placed "/" characters.

8. **Delay, Clear Screen:**

delay(600); cleardevice();

Delays for a short period and clears the screen for the next frame.

9. **Graphics Cleanup:**

getch(); closegraph(); return 0;

Waits for a key press, closes the graphics system, and returns 0.

Note: The **rand()** function is used to generate pseudo-random numbers. The **srand(time(NULL))** call seeds the random number generator based on the current time, ensuring a different sequence on each run.