

**IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY
RECORD REPORT**

Submitted by
SRINIDHY P 2017506601

Under the supervision of
Dr D SANGEETHA



**DEPARTMENT OF INFORMATION TECHNOLOGY
MADRAS INSTITUTE OF TECHNOLOGY CAMPUS**

ANNA UNIVERSITY, CHENNAI – 600044

NOVEMBER 2020

BONAFIDE CERTIFICATE

Certified that the Bonafide record of the practical work done by Mr./Miss **SRINIDHY P (2017506601)** in the Academic Semester 7 for the **Laboratory of Graphics and Multimedia** (Subject Code **IT7711**) during the period from July 2020 to November 2020.

Date: 20-11-2020

Lab In-charge: Dr D SANGEETHA

LIST OF EXPERIMENTS

S.NO	DATE	TITLE OF EXPERIMENT	PAGE NO
1A	28/08/2020	Study of 2D-Primitives in Graphics	04
1B	28/08/2020	2D Primitives inbuilt functions in graphics	09
1C	28/08/2020	Simple Graphics programs	15
2	01/09/2020	DDA Algorithm	18
3	01/09/2020	Bresenham's algorithm	20
4	08/09/2020	Midpoint circle Algorithm	22
5	08/09/2020	Midpoint ellipse Algorithm	25
6	15/09/2020	Window to viewport transformation	28
7	22/09/2020	Liang barsky clipping Algorithm	31
8	22/09/2020	Cohen sutherland clipping Algorithm	35
9	07/10/2020	Perform 2D Transformations of an object	41
10	19/10/2020	Create 2D and 3D objects using OpenGL	44
11	29/10/2020	Create 3D Projections using OpenGL	48
12	20/11/2020	Mini Project Report - Path Finding using OPENGL	51

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 1A

DATE: 28/08/2020

Aim: To study about various 2D- graphical primitives involved in Computer graphics.

DEFINITION-

GRAPHICAL PRIMITIVE: In graphics, primitives are basic elements, such as lines, curves, and polygons, which you can combine to create more complex graphical images. In programming, primitives are the basic operations supported by the programming language.

In reality, Computer graphics refer different things in different contexts:

- Pictures, scenes that are generated by a computer.
- Tools used to make such pictures, software and hardware, input/output devices.
- The whole field of study that involves these tools and the pictures they produce.

Types of 2D graphics

2D graphics come in two flavours —

1. Raster - Raster graphics are the most common and are used for digital photos, Web graphics, icons, and other types of images. They are composed of a simple grid of pixels, which can each be a different colour.
2. Vector - made up of paths, which may be lines, shapes, letters, or other scalable objects. They are often used for creating logos, signs, and other types of drawings. Unlike raster graphics, vector graphics can be scaled to a larger size without losing quality.

Applications of Computer

Graphics Pictures, Drawings, etc. Mathematical or Geometrical Models
Computer Graphics Image Processing Overview of graphics system and output primitives Computer graphics is used today in many different areas of science, engineering, industry, business, education, entertainment, medicine, art and training. All of these are included in the following categories.

1. User interfaces
2. Plotting
3. Office automation and electronic publishing.
4. Computer Aided Drafting and Design
5. Scientific and business Visualization
6. Simulation and modelling.
7. Entertainment i.e. Disney movies such as Lion Kings and The Beauty of Beast, and other scientific movies like Jurassic Park, The lost world etc are the best example of the application of computer graphics in the field of entertainment.
8. Art and commerce
9. Cartography

Primitives

A basic non-divisible graphical element for input or output within a computer-graphics system.

Input primitive

- Typical output primitives are polyline, poly marker, and fill area.
Clipping of an output primitive cannot be guaranteed to produce another output primitive.
- Typical input primitives are locator, choice, and valuator.
- Input primitives often have a style of echoing associated with them.

Output primitives

- Output primitives are the geometric structures that has attributes such as straight line segments (pixel array) and patterns i.e.; polygon colour areas, used to describe the shapes and colours of the objects.
- Points and straight line segments are the simplest geometric components of pictures.
- Additional output primitive includes: circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon colour areas and character strings. This includes picture generation algorithm by examining device-level algorithms for displaying two-dimensional output primitives, with emphasis on scan-conversion methods for raster graphics system.

Different 2D graphical primitives

Point

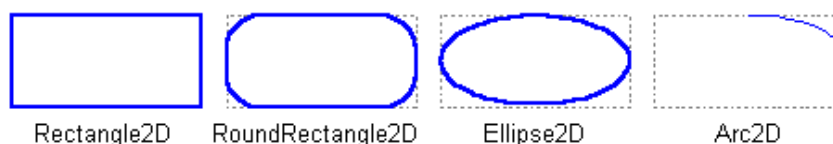
Point plotting is accomplished by converting a single coordinate position furnished by an application program into appropriate operations for the output device. With a CRT monitor, for example, the electron beam is turned on to illuminate the screen phosphor at the selected location.

Line

Line drawing is accomplished by calculating intermediate positions along the line path between two specified end points positions. An output device is then directed to fill in these positions between the end points.

Rectangular Shapes

The Rectangle2D, RoundRectangle2D, Arc2D, and Ellipse2D primitives are all derived from the **RectangularShape class**. This class defines methods for Shape objects that can be described by a rectangular bounding box. The geometry of a Rectangular Shaped object can be extrapolated from a rectangle that completely encloses the outline of the Shape.



Quadratic and cubic curves

The `QuadCurve2D` enables you to create quadratic parametric curve segments. A quadratic curve is defined by two endpoints and one control point.

The `CubicCurve2D` class enables you to create cubic parametric curve segments. A cubic curve is defined by two endpoints and two control points. The following are examples of quadratic and cubic curves. See [Stroking and Filling Graphics Primitives](#) for implementations of cubic and quadratic curves.

This figure represents a quadratic curve.

Control Point




This figure represents a cubic curve.

Control Point

Control Point

Bezier curve



Arbitrary Shapes

The **`GeneralPath`** class enables you to construct an arbitrary shape by specifying a series of positions along the shape's boundary. These positions can be connected by line segments, quadratic curves, or cubic (Bézier) curves. The following shape can be created with three line segments and a cubic curve.



Areas

With the Area class, you can perform boolean operations, such as union, intersection, and subtraction, on any two Shape objects. This technique, often referred to as *constructive area geometry*, enables you to quickly create complex Shape objects without having to describe each line segment or curve.

Line Drawing Algorithms

- Digital Differential Analyzer (DDA) Algorithm
- Bresenham's Line Algorithm
- Parallel Line Algorithm

Result: Thus the experiment has been successfully completed.

EXPERIMENT: 1B

DATE: 28/08/2020

Aim: To study about various graphics functions in C.

Functions with Syntax:

- ❖ arc - it used to draw an arc with center (x, y) and stangle specifies starting angle, endangle specifies the end angle and last parameter specifies the radius of the arc. arc function can also be used to draw a circle but for that starting angle and end angle should be 0 and 360 respectively.

Declaration: void arc(int x, int y, int stangle, int endangle, int radius);

- ❖ bar - Bar function is used to draw a 2-dimensional, rectangular filled in bar . Coordinates of left top and right bottom corner are required to draw the bar. Left specifies the X-coordinate of top left corner, top specifies the Y-coordinate of top left corner, right specifies the X-coordinate of right bottom corner, bottom specifies the Y-coordinate of right bottom corner.

Declaration: void bar(int left, int top, int right, int bottom);

- ❖ bar3d - bar3d function is used to draw a 2-dimensional, rectangular filled in bar . Coordinates of left top and right bottom corner of bar are required to draw the bar. left specifies the X-coordinate of top left corner, top specifies the Y-coordinate of top left corner, right specifies the X-coordinate of right bottom corner, bottom specifies the Y-coordinate of right bottom corner, depth specifies the depth of bar in pixels, topflag determines whether a 3 dimensional top is put on the bar or not (if it's non-zero then it's put otherwise not).

Declaration: void bar3d(int left, int top, int right, int bottom, int depth, int topflag);

- ❖ circle - Circle function is used to draw a circle with center (x,y) and third parameter specifies the radius of the circle.

Declaration: void circle(int x, int y, int radius);

- ❖ cleardevice - cleardevice function clears the screen in graphics mode and sets the current position to (0,0). Clearing the screen consists of filling the screen with current background color.

Declaration: void cleardevice();

- ❖ closegraph - closegraph function closes the graphics mode, deallocates all memory allocated by graphics system and restores the screen to the mode it was in before you called initgraph.

Declaration: void closegraph();

- ❖ drawpoly - Drawpoly function is used to draw polygons i.e. triangle, rectangle, pentagon, hexagon etc.

Declaration: void drawpoly(int num, int *polypoints);

- ❖ Ellipse - Ellipse is used to draw an ellipse (x,y) are coordinates of center of the ellipse, stangle is the starting angle, end angle is the ending angle, and fifth and sixth parameters specifies the X and Y radius of the ellipse.

Declarations: void ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);

- ❖ fillellipse - x and y are coordinates of center of the ellipse, xradius and yradius are x and y radius of ellipse respectively.

Declaration: void fillellipse(int x, int y, int xradius, int yradius);

- ❖ fillpoly - Fillpoly function draws and fills a polygon. It require same arguments as drawpoly.

Declaration: void drawpoly(int num, int *polypoints);

- ❖ floodfill - floodfill function is used to fill an enclosed area.

Declaration: void floodfill(int x, int y, int border);

- ❖ getarccords - getarccords function is used to get coordinates of arc which is drawn most recently.

Declaration: void getarccords(struct arccoordstype *var);

- ❖ getbkcolor - Function getbkcolor returns the current background-color.

Declaration: int getbkcolor();

- ❖ getcolor - getcolor function returns the current drawing color.

Declaration: int getcolor();

- ❖ getdrivename - Function getdrivename returns a pointer to the current graphics driver.

Declaration: char *name = getdrivename();

- ❖ getimage - getimage function saves a bit image of specified region into memory, region can be any rectangle.

Declaration: void getimage(int left, int top, int right, int bottom, void *bitmap);

- ❖ getmaxcolor - getmaxcolor function returns maximum color value for current graphics mode and driver. Total number of colors available for current graphics mode and driver are (getmaxcolor() + 1) as color numbering starts from zero.

Declaration: int getmaxcolor();

- ❖ getmaxx - getmaxx function returns the maximum X coordinate for current graphics mode and driver.

Declaration: int getmaxx();

- ❖ getmaxy - getmaxy function returns the maximum Y coordinate for current graphics mode and driver.

Declaration: int getmaxy();

- ❖ getpixel - getpixel function returns color of pixel present at location(x, y).

Declaration: int getpixel(int x, int y);

- ❖ getx - The function getx returns the X coordinate of the current position.

Declaration: int getx();

- ❖ gety - gety function returns the y coordinate of current position.

Declaration: int gety();

- ❖ graphdefaults - graphdefaults function resets all graphics settings to their defaults.

Declaration: void graphdefaults();

- ❖ grapherrormsg - grapherrormsg function returns an error message string.

Declaration: char *grapherrormsg(int errorcode);

- ❖ imagesize - imagesize function returns the number of bytes required to store a bitimage. This function is used when we are using [getimage](#).

Declaration:- unsigned int imagesize(int left, int top, int right, int bottom);

- ❖ line - line function is used to draw a line from a point(x1,y1) to point(x2,y2) i.e. (x1,y1) and (x2,y2) are end points of the line.

Declaration: void line(int x1, int y1, int x2, int y2);

- ❖ moveto -moveto function changes the current position (CP) to (x, y)

Declaration: void moveto(int x, int y);

- ❖ moverel - Function moverel moves the current position to a relative distance.

Declaration: void moverel(int x, int y);

- ❖ outtext - outtext function displays text at current position.

Declaration: void outtext(char *string);

- ❖ outtextxy - outtextxy function display text or string at a specified point(x,y) on the screen.

Declaration: void outtextxy(int x, int y, char *string);

- ❖ putimage - putimage function outputs a bit image onto the screen.

Declaration: void putimage(int left, int top, void *ptr, int op);

- ❖ **putpixel** - putpixel function plots a pixel at location (x, y) of specified color.

Declaration: void putpixel(int x, int y, int color);

- ❖ **Rectangle** - rectangle function is used to draw a rectangle. Coordinates of left top and right bottom corner are required to draw the rectangle. left specifies the X-coordinate of top left corner, top specifies the Y-coordinate of top left corner, right specifies the X-coordinate of right bottom corner, bottom specifies the Y-coordinate of right bottom corner.

Declaration: void rectangle(int left, int top, int right, int bottom);

- ❖ **Sector** -Sector function draws and fills an elliptical pie slice.

Declaration: void sector (int x, int y, int stangle, int endangle, int xradius, int yradius);

- ❖ **setbkcolor** - setbkcolor function changes current background color

Declaration: void setbkcolor(int color);

- ❖ **setcolor** – setcolor function is used to change the current drawing color.

Declaration: void setcolor(int color);

- ❖ **setfillstyle** -setfillstyle function sets the current fill pattern and fill color.

Declaration: void setfillstyle(int pattern, int color);

- ❖ **setlinestyle** – used to declare style of a line

Declaration: void setlinestyle(int linestyle, unsigned pattern, int thickness);

- ❖ **settextstyle** -Settextstyle function is used to change the way in which text appears, using it we can modify the size of text, change direction of text and change the font of text.

Declaration: void settextstyle(int font, int direction, int charsize);

- ❖ **setviewport** -setviewport function sets the current viewport for graphics output.

Declaration: void setviewport(int left, int top, int right, int bottom, int clip);

- ❖ textheight -textheight function returns the height of a string in pixels.

Declaration: int textheight(char *string);

- ❖ textwidth -Textwidth function returns the width of a string in pixels.

Declaration: int textwidth(char *string);

Result: Thus the experiment has been successfully completed and various graphics functions has been studied.

EXPERIMENT: 1C

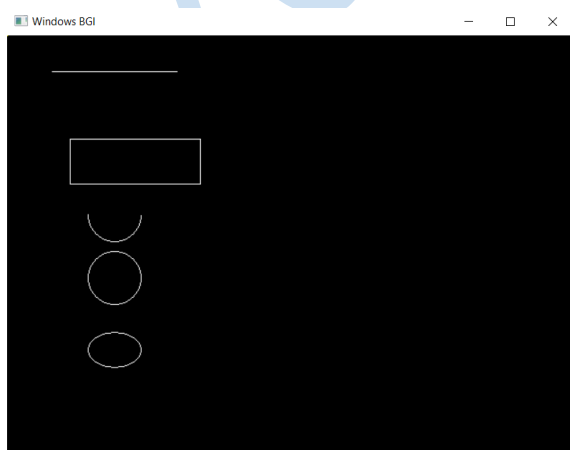
DATE: 28/08/2020

Aim: To write simple graphics program in C using basic graphics.h library functions.

1. Source Code:

```
#include<graphics.h>
#include<conio.h>
main(){
    int gd=DETECT,gm;
    initgraph (&gd,&gm,"c:\\tc\\bgi");
    setbkcolor(BLACK);
    line(50,40,190,40);
    rectangle(70,115,215,165);
    arc(120,200,180,0,30);
    circle(120,270,30);
    ellipse(120,350,0,360,30,20);
    getch();
}
```

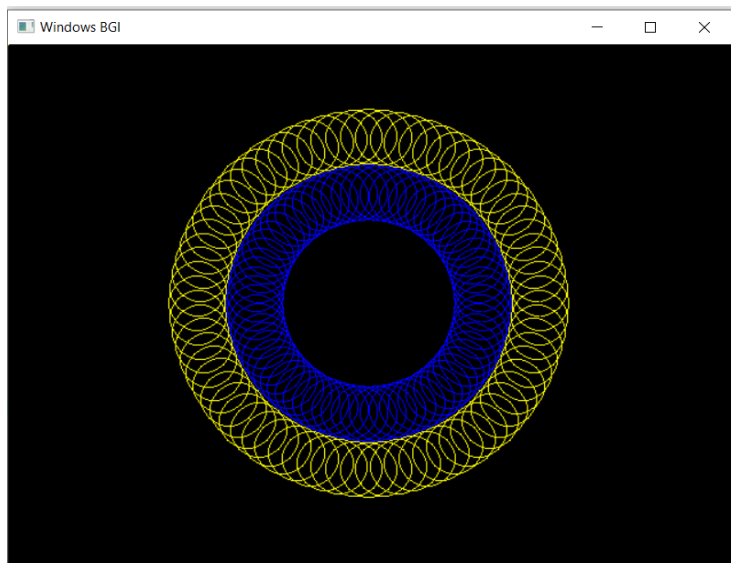
Output:



2. Source Code:

```
#include<graphics.h>
#include<conio.h>
#include<dos.h>
main()
{
    int gd = DETECT, gm, x, y, color, angle = 0;
    struct arccoordstype a, b;
    initgraph(&gd, &gm, "C:\\TC\\BGI");
    delay(2000);
    while(angle<=360)
    {
        setcolor(BLACK);
        arc(getmaxx()/2,getmaxy()/2,angle,angle+2,100);
        setcolor(BLUE);
        getarccoords(&a);
        circle(a.xstart,a.ystart,25);
        setcolor(BLACK);
        arc(getmaxx()/2,getmaxy()/2,angle,angle+2,150);
        getarccoords(&a);
        setcolor(YELLOW);
        circle(a.xstart,a.ystart,25);
        angle = angle+5;
        delay(50);
    }
    getch();
    closegraph();
}
```


Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 2

DATE: 01/09/2020

Aim: To write a program to implement line drawing Digital Differential Algorithm (DDA) in C++.

Algorithm:

Step 1: Get two endpoint pixel positions as Input.

Step 2: Horizontal and vertical differences between the endpoint positions are assigned to parameters dx and dy. (Calculate $dx = x_b - x_a$ and $dy = y_b - y_a$).

Step 3: The difference with the greater magnitude determines the value of parameter steps.

Step 4: Starting with pixel position (xa, ya), determine the offset needed at each step to generate the next pixel position along the line path.

Step 5: loop the following process for a number of times. Use a unit of increment or decrement in the x and y direction b. if xa is less than xb the values of increment in the x and y directions are 1 and m c. if xa is greater than xb then the decrements -1 and -m are used.

Source Code:

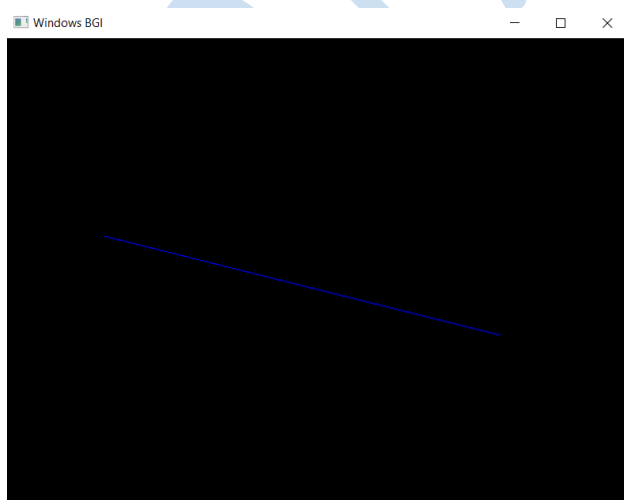
```
#include<graphics.h>
#include<conio.h>
#include<stdio.h>
main() {
    int gd = DETECT ,gm, i;
    float x, y,dx,dy,steps;
    int x0, x1, y0, y1;
    initgraph(&gd, &gm, "C:\\\\TC\\\\BGI");
    setbkcolor(WHITE);
    x0 = 100 , y0 = 200, x1 = 500, y1 = 300;
```

```

dx = (float)(x1 - x0);
dy = (float)(y1 - y0);
if(dx>=dy){
    steps = dx; }
else{
    steps = dy; }
dx = dx/steps;
dy = dy/steps;
x = x0; y = y0; i = 1;
while(i<= steps){
    putpixel(x, y, BLUE);
    x += dx; y += dy;
    i=i+1;
}
getch();
closegraph();
}

```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

EXPERIMENT: 3

DATE: 01/09/2020

Aim: To write a program to implement Bresenham's line drawing algorithm in C++.

Algorithm:

1. Input the two line endpoints and store the left end point in (x0, y0).
2. Load (x0, y0) into frame buffer, ie. Plot the first point.
3. Calculate the constants Δx , Δy , $2\Delta y$ and obtain the starting value for the decision parameter as
 $P_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, starting at $k=0$ perform the following test If $P_k < 0$, the next point to plot is (x_{k+1}, y_k) and $P_{k+1} = P_k + 2\Delta y$ otherwise, the next point to plot is (x_{k+1}, y_{k+1}) and $P_{k+1} = P_k + 2\Delta y - 2\Delta x$
5. Perform step4 Δx times.

Source Code:

```
#include<iostream.h>
#include<graphics.h>
void drawline(int x0, int y0, int x1, int y1{
    int dx, dy, p, x, y;
    dx =x1-x0;
    dy =y1-y0;
    x=x0;
    y=y0;
    p=2*dy-dx
    while(x<x1){
        if(p>=0){
            putpixel(x,y,7);
```

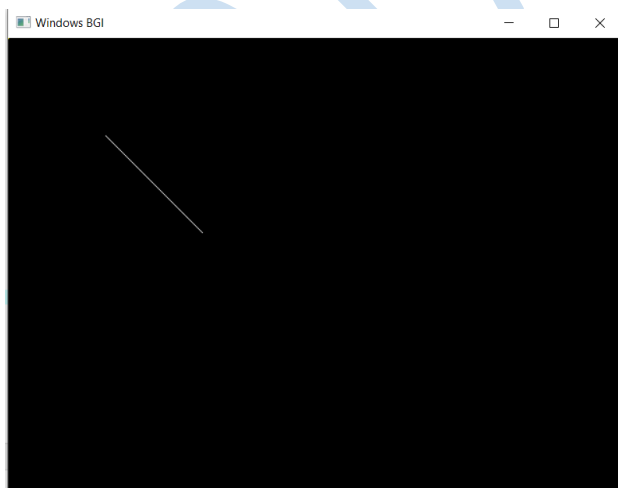
```

        y=y+1;
        p=p+2*dy-2*dx;
    }
    else{
        putpixel(x,y,7);
        p=p+2*dy;
    }
    x=x+1;
}
}

int main(){
    int gdriver=DETECT, gmode, error, x0=100, y0=100, x1=200, y1=200;
    initgraph(&gdriver, &gmode, "c:\\turbo3\\bgi");
    drawline(x0, y0, x1, y1);
    return 0;
}

```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 4

DATE: 08/09/2020

Aim: To implement Mid-Point Circle Algorithm in C++.

Algorithm:

Step1: Initially assume $x = 0$, $y = r$
we have $p = 1 - r$

Step2: Repeat the steps while $x \leq y$

Plot (x, y)

If $(p < 0)$

Then have $p = p + 2x + 3$

Else

$p = p + 2(x - y) + 5$

$y = y - 1$ (end if)

$x = x + 1$ (end loop)

Step3: End

Source Code:

```
#include<graphics.h>
#include<stdio.h>
void pixel(int xc,int yc,int x,int y);
int main(){
    int gd,gm,xc,yc,r,x,y,p;
    detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"C://TurboC3//BGI");

    printf("Enter center of circle :");
    scanf("%d%d",&xc,&yc);
    printf("Enter radius of circle :");
```

```

scanf("%d",&r);
x=0;
y=r;
p=1-r;
pixel(xc,yc,x,y);

while(x<y)
{
    if(p<0) {
        x++;
        p=p+2*x+1;
    }
    else{
        x++;
        y--;
        p=p+2*(x-y)+1;
    }
    pixel(xc,yc,x,y);
}
getch();
closegraph();
return 0;
}

```

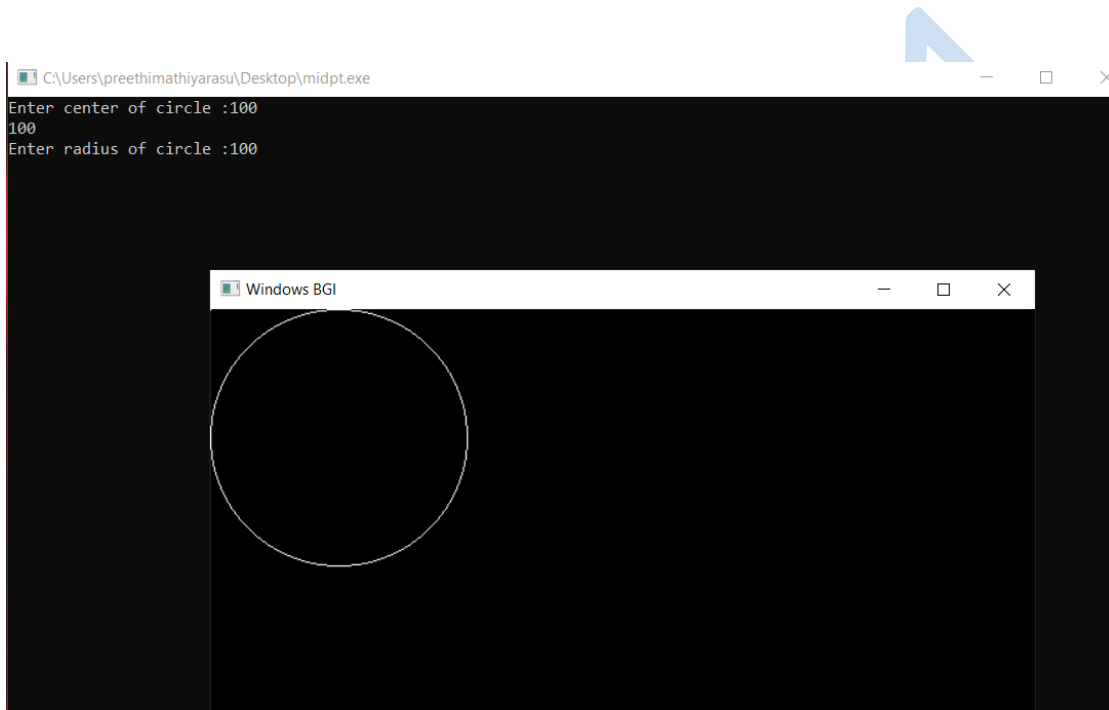
```

void pixel(int xc,int yc,int x,int y) {
    putpixel(xc+x,yc+y,WHITE);
    putpixel(xc+x,yc-y,WHITE);
    putpixel(xc-x,yc+y,WHITE);
    putpixel(xc-x,yc-y,WHITE);
    putpixel(xc+y,yc+x,WHITE);
}

```

```
    putpixel(xc+y,yc-x,WHITE);  
    putpixel(xc-y,yc+x,WHITE);  
    putpixel(xc-y,yc-x,WHITE);  
}
```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 5

DATE: 08/09/2020

Aim: To implement Mid-Point Ellipse Algorithm in C++.

Algorithm:

1. Get input radius along x axis and y axis and find centre of ellipse.
2. We assume the centre of ellipse to be at origin and the first point as:
 $(x, y_0) = (0, r_y)$.
3. Find the initial decision parameter for region 1.
4. For every x_k position in region 1 :
If $p1_k < 0$ then the next point along the is (x_{k+1}, y_k)
Else, the next point is (x_{k+1}, y_{k-1}) .
5. Obtain the initial value in region 2 using the last point (x_0, y_0) of region 1.
6. At each y_k in region 2 starting at $k = 0$ check the condition.
If $p2_k > 0$ the next point is (x_k, y_{k-1})
Else, the next point is (x_{k+1}, y_{k-1})
7. Now obtain the symmetric points in the three quadrants and plot the coordinate value as: $x = x + x_c$, $y = y + y_c$
8. Repeat the steps for region 1 until $2r_y^2x \geq 2r_x^2y$

Source Code:

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

main () {
    float x,y,a, b,r,p,h,k,p1,p2;
    h=319;
    k=239;
```

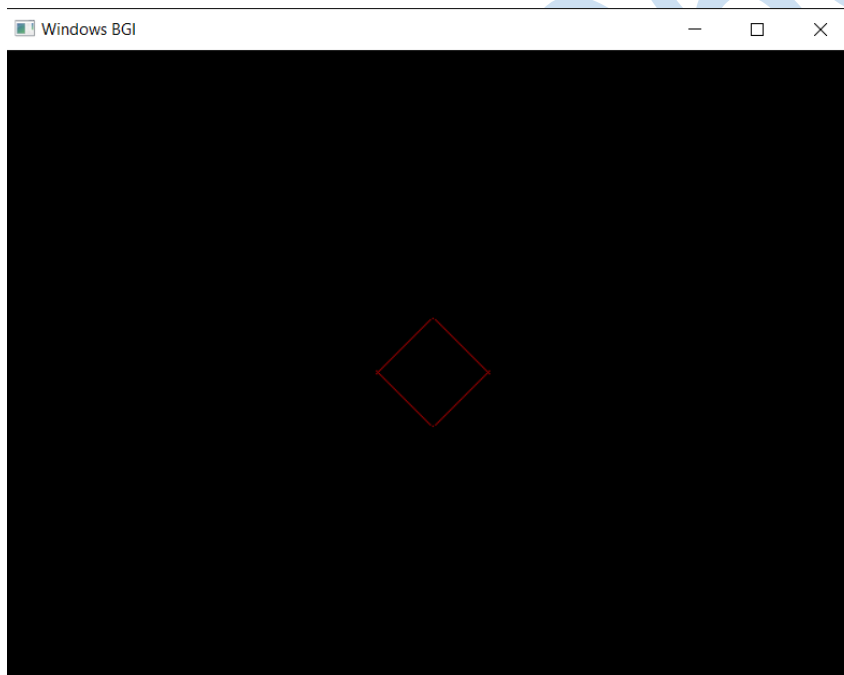
```

    a=50;
    b=40;
int gdriver = DETECT,gmode, errorcode;
int midx, midy, i;
initgraph (&gdriver, &gmode, " ");
x=0;
y=b;
p1 = ((b * b)-(a * a * b) + (a * a)/4);
{
    putpixel (x+h, y+k, RED);
    putpixel (-x+h, -y+k, RED);
    putpixel (x+h, -y+k, RED);
    putpixel (-x+h, y+k, RED);
    if (p1 < 0)
        p1 += ((2 * b * b) * (x+1)) - ((2 * a * a) * (y-1)) + (b * b);
    else
    {
        p1 += ((2 * b * b) * (x+1)) - ((2 * a * a) * (y-1)) - (b * b);
        y--;
    }
    x++;
}
p2 = ((b * b) * (x + 0.5)) + ((a * a) * (y-1) * (y-1)) - (a * a * b * b);
while (y >= 0){
    if (p2 > 0)
        p2 = p2 - ((2 * a * a) * (y-1)) + (a * a);
    else{
        p2 = p2 - ((2 * a * a) * (y-1)) + ((2 * b * b) * (x+1)) + (a * a);
        x++;
    }
}

```

```
    }  
    y--;  
    putpixel (x+h, y+k, RED);  
    putpixel (-x+h, -y+k, RED);  
    putpixel (x+h, -y+k, RED);  
    putpixel (-x+h, y+k, RED);  
}  
getch();  
}
```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 6

DATE: 15/09/2020

Aim: To implement window to viewport transformation in C.

Source Code:

```
#include <graphics.h>
#include<conio.h>
#include <stdio.h>
main(){
int wxmax,wymax,wxmin,wymn;
int vxmax,vymax,vxmin,vymn;
float sx,sy;
int x,x1,x2,y,y1,y2;
int gr=DETECT ,gm;
initgraph (&gr,&gm,"C:\\TURBOC3\\BGI");

printf("Enter coordinates for triangle x,y");
scanf("%d %d",&x,&y);
printf("\n x1 and y1");
scanf("%d %d",&x1,&y1);
printf("\n x2 and y2");
scanf("%d %d",&x2,&y2);

printf("Enter window coordinates for triangle wxmax,wymax");
scanf("%d %d",&wxmax,&wymax);
```

```
printf("Enter window coordinates for triangle wxmin,wymin");  
scanf("%d %d",&wxmin,&wymin);  
cleardevice();  
delay(50);  
rectangle(wxmin,wymin,wxmax,wymax);  
outtextxy(wxmin,wymin-10,"Window");
```

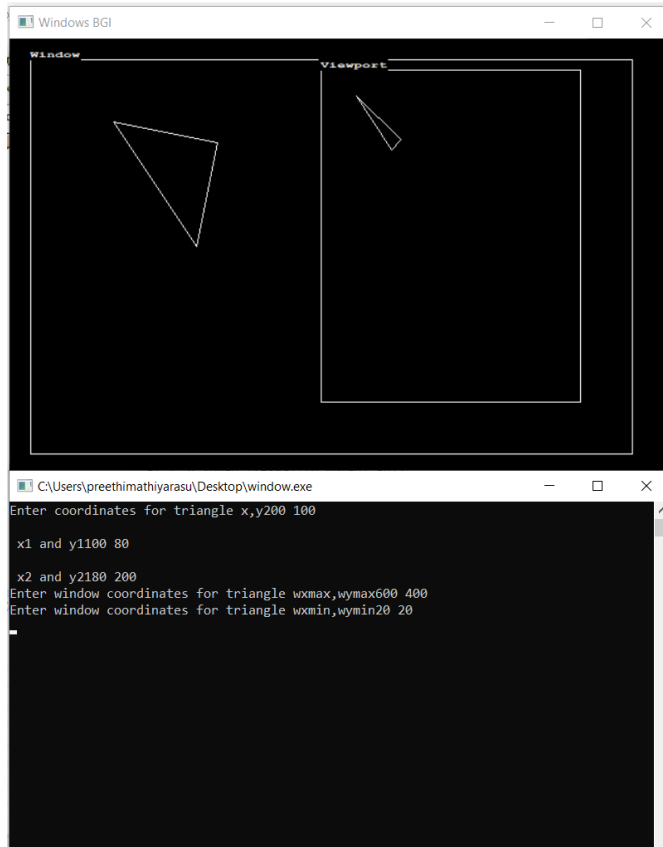
```
line(x,y,x1,y1);  
line(x1,y1, x2,y2);  
line(x2,y2,x,y);  
vxmin=300; vymin=30; vxmax=550; vymax=350;  
rectangle(vxmin,vymin,vxmax,vymax);  
outtextxy(vxmin,vymin-10,"Viewport");
```

```
sx=(float)(vxmax-vxmin)/(wxmax-wxmin);  
sy=(float)(vymax-vymin)/(wymax-wymin);  
x=vxmin+(float)((x-wxmin)*sx);  
y=vymin+(float)((y-wymin)*sy);  
x1=vxmin+(float)((x1-wxmin)*sx);  
x2=vxmin+(float)((x2-wxmin)*sx);  
y1=vymin+(float)((y1-wymin)*sx);  
y2=vymin+(float)((y2-wymin)*sx);
```

```
line(x,y,x1,y1);  
line(x1,y1, x2,y2);  
line(x2,y2,x,y);  
getch();
```

```
closegraph();  
}
```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 7

DATE: 22/09/2020

Aim: To implement liang barsky line clipping algorithm in C++.

Algorithm:

1. Assume $t_{\min}=0$ and $t_{\max}=1$
2. Calculate the values of t_L, t_R, t_T and t_B .
If $t < t_{\min}$ or $t > t_{\max}$ ignore it and go to the next edge
Otherwise classify the t value as entering or exiting value
If t is entering value set $t_{\min} = t$ if t is exiting value set $t_{\max} = t$
3. If $t_{\min} < t_{\max}$ then draw a line from $(x_1 + dx * t_{\min}, y_1 + dy * t_{\min})$ to $(x_1 + dx * t_{\max}, y_1 + dy * t_{\max})$
4. If the line crosses over the window, you will see $(x_1 + dx * t_{\min}, y_1 + dy * t_{\min})$ and $(x_1 + dx * t_{\max}, y_1 + dy * t_{\max})$ are intersection between line and edge.

Source Code:

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
#include<dos.h>
main(){
    int i,gd=DETECT,gm;
    int x1,y1,x2,y2,xmin,xmax,ymin,ymax,xx1,xx2,yy1,yy2,dx,dy;
    float t1,t2,p[4],q[4],temp;
    x1=120;    y1=120;
    x2=300;    y2=300;
    xmin=100;  ymin=100;
```

```

xmax=250; ymax=250;
initgraph(&gd,&gm,"c:\\turbo3\\bgi");
rectangle(xmin,ymin,xmax,ymax);
dx=x2-x1;  dy=y2-y1;
p[0]=-dx;  p[1]=dx;
p[2]=-dy;  p[3]=dy;
q[0]=x1-xmin;
q[1]=xmax-x1;
q[2]=y1-ymin;
q[3]=ymax-y1;
for(i=0;i<4;i++){
    if(p[i]==0){
        printf("line is parallel to one of the clipping boundary");
        if(q[i]>=0){
            if(i<2) {
                if(y1<ymin) {
                    y1=ymin;
                }
                if(y2>ymax) {
                    y2=ymax;
                }
                line(x1,y1,x2,y2);
            }
            if(i>1) {
                if(x1<xmin) {
                    x1=xmin;
                }
            }
        }
    }
}

```



```

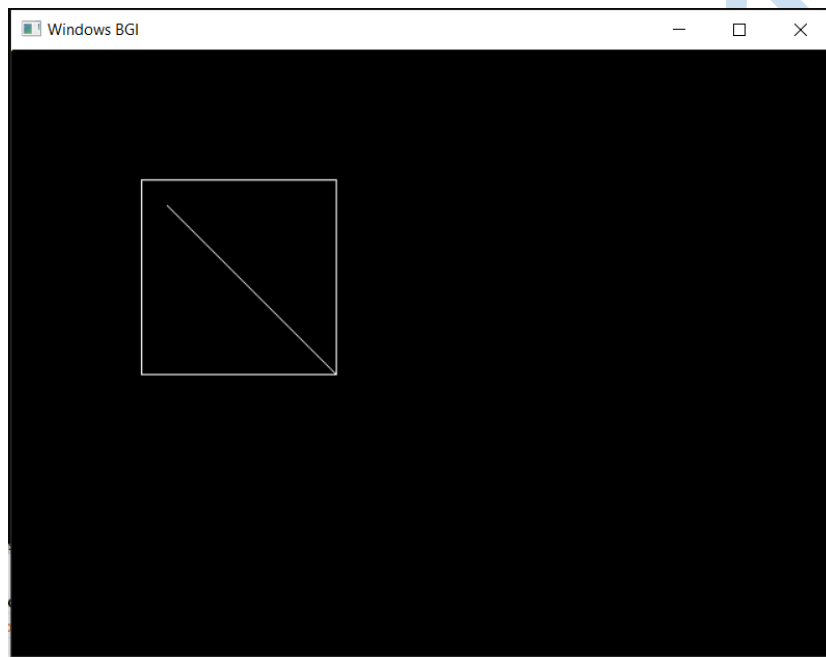
        if(x2>xmax) {
            x2=xmax;
        }
        line(x1,y1,x2,y2);
    }
}

t1=0;
t2=1;
for(i=0;i<4;i++){
    temp=q[i]/p[i];
    if(p[i]<0) {
        if(t1<=temp)
            t1=temp;
    }
    else{
        if(t2>temp)
            t2=temp;
    }
}
if(t1<t2) {
    xx1 = x1 + t1 * p[1];
    xx2 = x1 + t2 * p[1];
    yy1 = y1 + t1 * p[3];
    yy2 = y1 + t2 * p[3];
}

```

```
        line(xx1,yy1,xx2,yy2);  
    }  
    delay(3000);  
    closegraph();  
}
```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 8

DATE: 22/09/2020

Aim: To implement cohen sutherland line clipping algorithm in C++.

Algorithm:

Step 1 : Assign a region code for two endpoints of given line

Step 2 : If both endpoints have a region code 0000 then given line is completely inside and we will keep this line

Step 3 : If step 2 fails, perform the logical AND operation for both region codes.

Step 3.a : If the result is not 0000, then given line is completely outside.

Step 3.b : Else line is partially inside.

Step 3.b.a : Choose an endpoint of the line that is outside the given rectangle.

Step 3.b.b : Find the intersection point of the rectangular boundary (based on region code).

Step 3.b.c : Replace endpoint with the intersection point and update the region code.

Step 3.b.d : Repeat step 2 until we find a clipped line either trivially accepted or rejected.

Step 4 : Repeat step 1 for new inputs.

Source Code:

```
#include <stdio.h>

class CohenSutherLandAlgo {
private:
    double x1,y1,x2,y2;
    double x_max,y_max,x_min,y_min;
    const int INSIDE = 0;
    const int LEFT  = 1;
    const int RIGHT = 2;
```

```

const int BOTTOM = 4;
const int TOP = 8;
public:
    CohenSutherlandAlgo() {
        x1 = 0.0;
        x2 = 0.0;
        y1 = 0.0;
        y2 = 0.0;
    }
    void getCoordinates();
    void getClippingRectangle();
    int generateCode(double x, double y);
    void cohenSutherland();
};

void CohenSutherlandAlgo::getCoordinates(){
    printf("\nEnter Co-ordinates of P1(X1,Y1) of Line Segment : ");
    printf("\nEnter X1 Co-ordinate : ");
    scanf( "%lf",&x1);
    printf( "\nEnter Y1 Co-ordinate : ");
    scanf( "%lf",&y1);
    printf("\nEnter Co-ordinates of P2(X2,Y2) of Line Segment : ");
    printf( "\nEnter X2 Co-ordinate : ");
    scanf( "%lf",& x2);
    printf( "\nEnter Y2 Co-ordinate : ");
    scanf( "%lf",& y2);
}

void CohenSutherlandAlgo::getClippingRectangle(){

```

```

printf( "\nEnter the Co-ordinates of Interested Rectangle.");
printf( "\nEnter the X_MAX : ");
scanf( "%lf",&x_max);
printf( "\nEnter the Y_MAX : ");
scanf( "%lf",& y_max);
printf( "\nEnter the X_MIN : ");
scanf( "%lf",& x_min);
printf( "\nEnter the Y_MIN : ");
scanf( "%lf",& y_min);
}

int CohenSutherLandAlgo::generateCode(double x, double y) {
    int code = INSIDE;    // intially initializing as being inside
    if (x < x_min)         // lies to the left of rectangle
        code |= LEFT;
    else if (x > x_max)    // lies to the right of rectangle
        code |= RIGHT;
    if (y < y_min)        // lies below the rectangle
        code |= BOTTOM;
    else if (y > y_max)    // lies above the rectangle
        code |= TOP;
    return code;
}

void CohenSutherLandAlgo::cohenSutherland() {
    int code1 = generateCode(x1, y1);
    int code2 = generateCode(x2, y2);
    bool accept = false;

    while (true) {

```

```

if ((code1 == 0) && (code2 == 0)) {
    accept = true;
    break;
}
else if (code1 & code2) {
    break;
}
else {
    int code_out;
    double x, y;
    if (code1 != 0)
        code_out = code1;
    else
        code_out = code2;
    if (code_out & TOP) {
        x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
        y = y_max;
    }
    else if (code_out & BOTTOM) {
        x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
        y = y_min;
    }
    else if (code_out & RIGHT) {
        y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
        x = x_max;
    }
    else if (code_out & LEFT) {

```

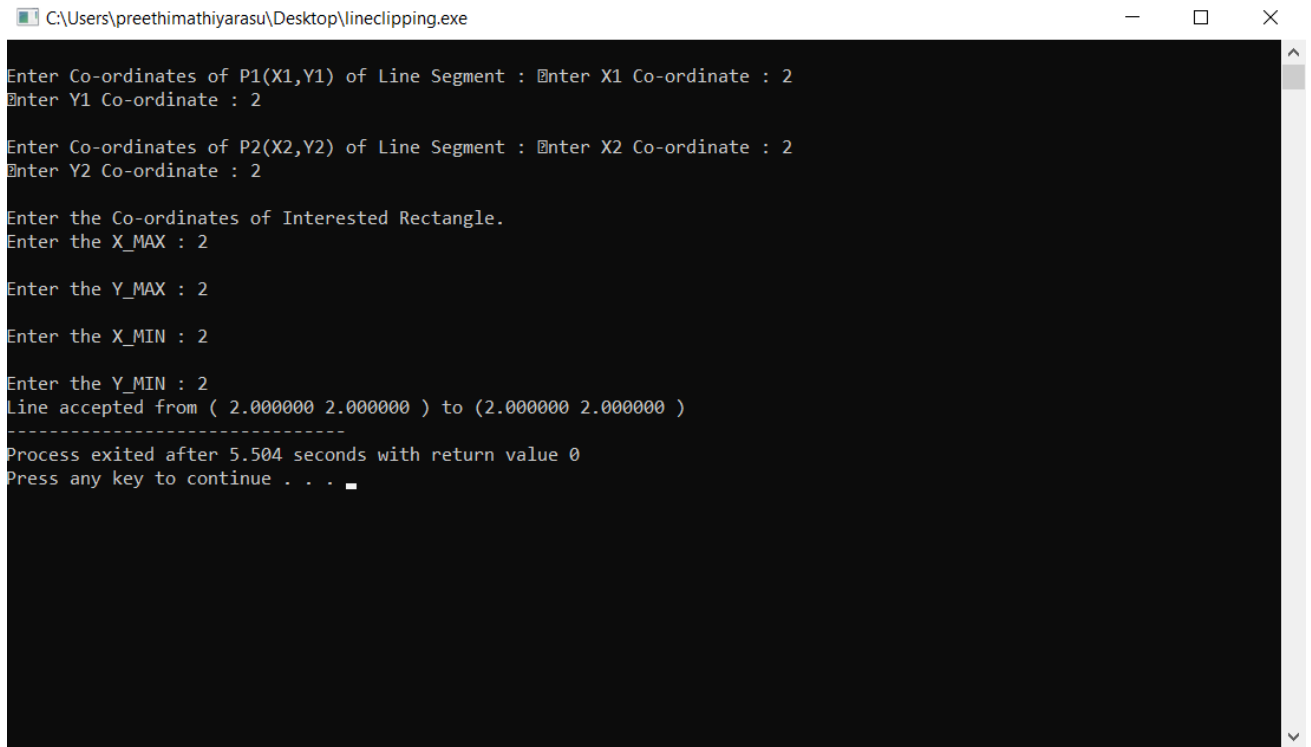
```

        y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
        x = x_min;
    }
    if (code_out == code1) {
        x1 = x;
        y1 = y;
        code1 = generateCode(x1, y1);
    }
    else{
        x2 = x;
        y2 = y;
        code2 = generateCode(x2, y2);
    }
}
}
if (accept) {
    printf("Line accepted from ( %lf %lf ) to (%lf %lf )",x1 ,y1,x2,y2 );
}
else
    printf("Line can't be drawn");
}
int main() {
    CohenSutherlandAlgo c;
    c.getCoordinates();
    c.getClippingRectangle();
    c.cohenSutherland();
    return 0;
}

```

}

Output:



```
C:\Users\preethimathiyarasu\Desktop\lineclipping.exe

Enter Co-ordinates of P1(X1,Y1) of Line Segment : Enter X1 Co-ordinate : 2
Enter Y1 Co-ordinate : 2

Enter Co-ordinates of P2(X2,Y2) of Line Segment : Enter X2 Co-ordinate : 2
Enter Y2 Co-ordinate : 2

Enter the Co-ordinates of Interested Rectangle.
Enter the X_MAX : 2

Enter the Y_MAX : 2

Enter the X_MIN : 2

Enter the Y_MIN : 2
Line accepted from ( 2.000000 2.000000 ) to (2.000000 2.000000 )
-----
Process exited after 5.504 seconds with return value 0
Press any key to continue . . .
```

Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 9

DATE: 07/10/2020

Aim: To perform 2D Transformations of an object in C++.

Algorithm: 1. Take 2*2 matrix

2. For each point of (1) make a 2*1 matrix P where P[0][0] equals to x coordinate of the point and P[1][0] equals to y coordinate of the point.

(2) Multiply the matrix with the point P to get new co-ordinate.

3. Draw the final result using the new points.

Source Code:

```
#include<stdio.h>
#include<graphics.h>
void findNewCoordinate(int s[][2], int p[][1])
{
    int temp[2][1] = { 0 };
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 1; j++)
            for (int k = 0; k < 2; k++)
                temp[i][j] += (s[i][k] * p[k][j]);
    p[0][0] = temp[0][0];
    p[1][0] = temp[1][0];
}
void scale(int x[], int y[], int sx, int sy)
{
    line(x[0], y[0], x[1], y[1]);
    line(x[1], y[1], x[2], y[2]);
}
```

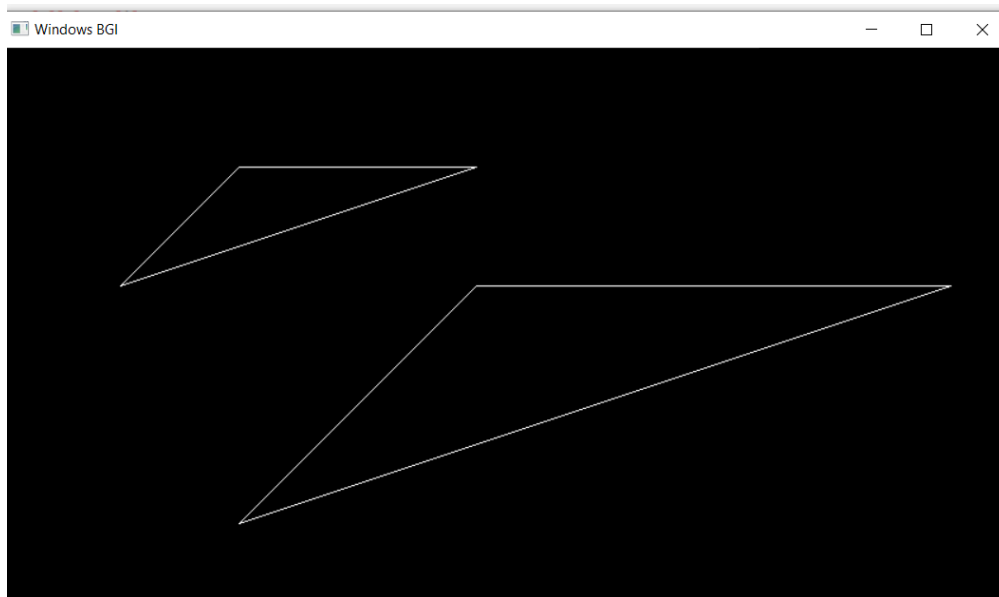
```

    line(x[2], y[2], x[0], y[0]);
    int s[2][2] = { sx, 0, 0, sy };
    int p[2][1];
    for (int i = 0; i < 3; i++)
    {
        p[0][0] = x[i];
        p[1][0] = y[i];
        findNewCoordinate(s, p);
        x[i] = p[0][0];
        y[i] = p[1][0];
    }
    line(x[0], y[0], x[1], y[1]);
    line(x[1], y[1], x[2], y[2]);
    line(x[2], y[2], x[0], y[0]);
}

int main()
{
    int x[] = { 100, 200, 400 };
    int y[] = { 200, 100, 100 };
    int sx = 2, sy = 2;
    int gd, gm;
    detectgraph(&gd, &gm);
    initgraph(&gd, &gm, " ");
    scale(x, y, sx, sy);
    getch();
    return 0;
}

```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 10

DATE: 19/10/2020

Aim: To create 2D and 3D objects using OpenGL.

Source Code:

```
#include <windows.h>
#include <GL/glut.h>
char title[] = "3D Shapes";
void initGL() {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glShadeModel(GL_SMOOTH);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(1.5f, 0.0f, -7.0f);
    glBegin(GL_QUADS);
        glColor3f(0.0f, 1.0f, 0.0f);    // Green
        glVertex3f( 1.0f, 1.0f, -1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glVertex3f(-1.0f, 1.0f,  1.0f); glVertex3f( 1.0f, 1.0f,  1.0f);
```

```

glColor3f(1.0f, 0.5f, 0.0f); // Orange
glVertex3f( 1.0f, -1.0f, 1.0f);    glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glColor3f(1.0f, 0.0f, 0.0f); // Red
glVertex3f( 1.0f, 1.0f, 1.0f);    glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);    glVertex3f( 1.0f, -1.0f, 1.0f);
glColor3f(1.0f, 1.0f, 0.0f); // Yellow
glVertex3f( 1.0f, -1.0f, -1.0f);    glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);    glVertex3f( 1.0f, 1.0f, -1.0f);
glColor3f(0.0f, 0.0f, 1.0f); // Blue
glVertex3f(-1.0f, 1.0f, 1.0f);    glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);    glVertex3f(-1.0f, -1.0f, 1.0f);
// Right face (x = 1.0f)
glColor3f(1.0f, 0.0f, 1.0f); // Magenta
glVertex3f(1.0f, 1.0f, -1.0f);    glVertex3f(1.0f, 1.0f, 1.0f);
glVertex3f(1.0f, -1.0f, 1.0f);    glVertex3f(1.0f, -1.0f, -1.0f);
glEnd();
glLoadIdentity();
glTranslatef(-1.5f, 0.0f, -6.0f);
glBegin(GL_TRIANGLES); // Front
glColor3f(1.0f, 0.0f, 0.0f); // Red
glVertex3f( 0.0f, 1.0f, 0.0f);
glColor3f(0.0f, 1.0f, 0.0f); // Green
glVertex3f(-1.0f, -1.0f, 1.0f);
glColor3f(0.0f, 0.0f, 1.0f); // Blue
glVertex3f(1.0f, -1.0f, 1.0f);
// Right

```

```

glColor3f(1.0f, 0.0f, 0.0f); // Red
glVertex3f(0.0f, 1.0f, 0.0f);
glColor3f(0.0f, 0.0f, 1.0f); // Blue
glVertex3f(1.0f, -1.0f, 1.0f);
glColor3f(0.0f, 1.0f, 0.0f); // Green
glVertex3f(1.0f, -1.0f, -1.0f);
// Back
glColor3f(1.0f, 0.0f, 0.0f); // Red
glVertex3f(0.0f, 1.0f, 0.0f);
glColor3f(0.0f, 1.0f, 0.0f); // Green
glVertex3f(1.0f, -1.0f, -1.0f);
glColor3f(0.0f, 0.0f, 1.0f); // Blue
glVertex3f(-1.0f, -1.0f, -1.0f);
// Left
glColor3f(1.0f,0.0f,0.0f); // Red
glVertex3f( 0.0f, 1.0f, 0.0f);
glColor3f(0.0f,0.0f,1.0f); // Blue
glVertex3f(-1.0f,-1.0f,-1.0f);
glColor3f(0.0f,1.0f,0.0f); // Green
glVertex3f(-1.0f,-1.0f, 1.0f);
glEnd(); // Done drawing the pyramid
glutSwapBuffers(); // Swap the front and back frame buffers (double
buffering)
}

void reshape(GLsizei width, GLsizei height) { // GLsizei for non-negative
integer
if (height == 0) height = 1;

GLfloat aspect = (GLfloat)width / (GLfloat)height;

```

```

glViewport(0, 0, width, height);

glMatrixMode(GL_PROJECTION); // To operate on the Projection matrix
glLoadIdentity();           // Reset
gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}

int main(int argc, char** argv) {

    glutInit(&argc, argv); glutInitDisplayMode(GLUT_DOUBLE);
    glutInitWindowSize(640, 480); glutInitWindowPosition(50, 50);
    glutCreateWindow(title); glutDisplayFunc(display);
    glutReshapeFunc(reshape);

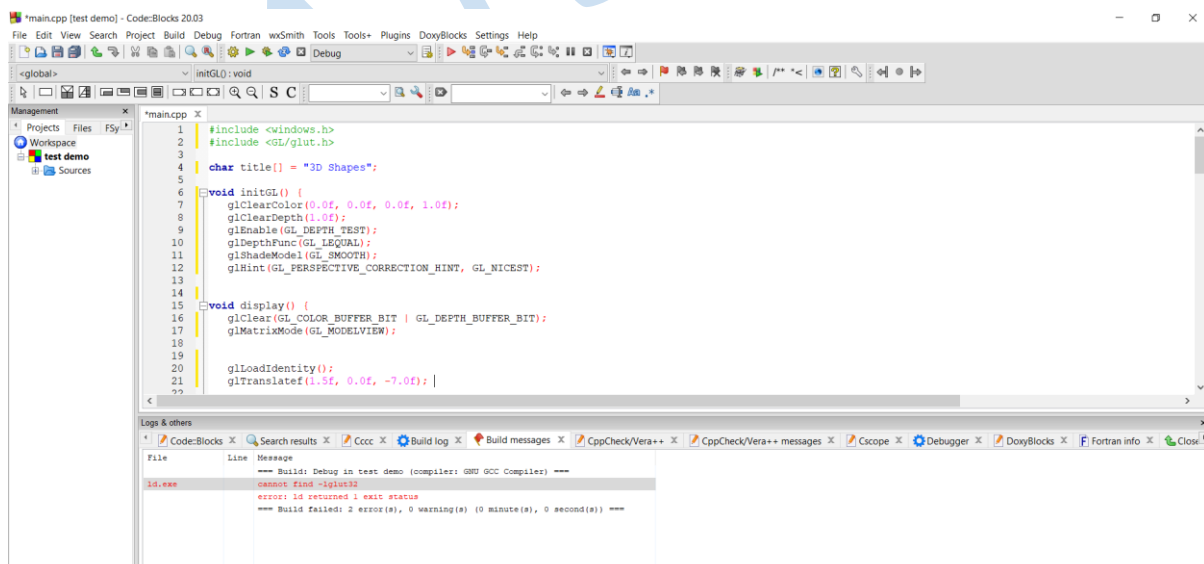
    initGL();

    glutMainLoop();

    return 0;
}

```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

EXPERIMENT: 11

DATE: 29/10/2020

Aim: To create 3D Projections using OpenGL.

Source Code:

```
#ifndef
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

static bool spinning = true;
static const int FPS = 60;
static GLfloat currentAngleOfRotation = 0.0;
void reshape(GLint w, GLint h) {
    glViewport(0, 0, w, h);
    GLfloat aspect = (GLfloat)w / (GLfloat)h;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h) {
        glOrtho(-50.0, 50.0, -50.0/aspect, 50.0/aspect, -1.0, 1.0);
    }
    else {
        glOrtho(-50.0*aspect, 50.0*aspect, -50.0, 50.0, -1.0, 1.0);
    }
}
```



```

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(currentAngleOfRotation, 0.0, 0.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glFlush();
    glutSwapBuffers();
}

void timer(int v) {
    if (spinning) {
        currentAngleOfRotation += 1.0;
        if (currentAngleOfRotation > 360.0) {
            currentAngleOfRotation -= 360.0;
        }
        glutPostRedisplay();
    }
    glutTimerFunc(1000/FPS, timer, v);
}

void mouse(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        spinning = true;
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
        spinning = false;
    }
}

```

```
}  
  
int main(int argc, char** argv) {  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);  
    glutInitWindowPosition(80, 80);  
    glutInitWindowSize(800, 500);  
    glutCreateWindow("Spinning Square");  
    glutReshapeFunc(reshape);  
    glutDisplayFunc(display);  
    glutTimerFunc(100, timer, 0);  
    glutMouseFunc(mouse);  
    glutMainLoop();  
}
```

Output:



Result: Thus the experiment has been successfully completed and the output is verified.

PATH FINDING GAME USING OPENGL

IT7711 COMPUTER GRAPHICS AND MULTIMEDIA LABORATORY

MINI PROJECT REPORT

Submitted by

M.PREETHI 2017506578

M.POOJA 2017506573

P.SRINIDHY 2017506601

Under the supervision of

Dr D SANGEETHA

In partial fulfilment for the award of the degree

Of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY



DEPARTMENT OF INFORMATION TECHNOLOGY

MADRAS INSTITUTE OF TECHNOLOGY CAMPUS

ANNA UNIVERSITY, CHENNAI – 600044

NOVEMBER 2020

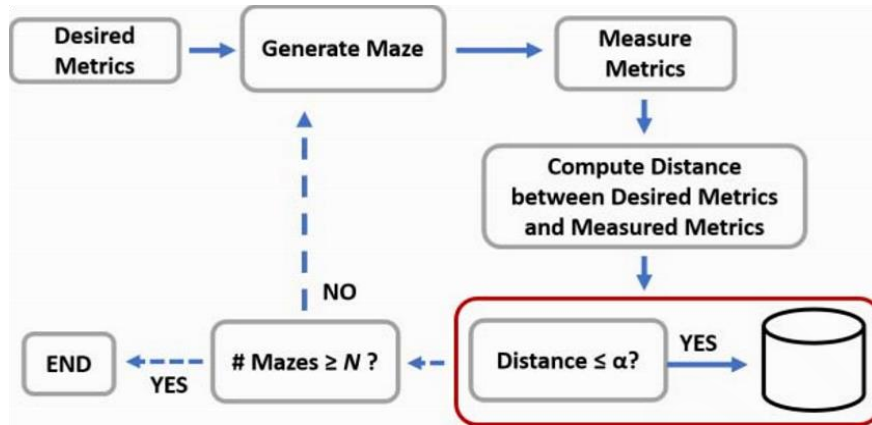
1. ABSTRACT

Game level design is one of the most important element of developing an enjoyable video game. Besides, game with difficult and dynamic level can make players more exciting. Maze is a puzzle, in which players need to find a path from an entrance to a goal on the maze. Although maze is a puzzle itself, it is also used as a tool in different fields, such as computer game and robotics. Since maze users in different fields may have different purposes of using a maze, different desired properties may be wanted on the maze. More specifically, the maze built is a 2-Dimensional of fixed shape and size, in which the horizontal and vertical walls are connected in such a way, so that the point can move from given starting point to the ending point through the spaces formed by connecting walls, but point should never cross the wall. Our research focuses on a perfect maze, and existing perfect maze generation algorithms can be used in the SBPCG approach. When the algorithms are used, to obtain a desired maze with a higher probability, our method chooses the best algorithm amongst intelligently. Lastly, we provide several use cases and demonstrate that our method generates desired mazes effectively for each use case. . In a field of a computer game, it can be used as a basis for a game level. In a field of robotics, it can be used as a platform to demonstrate robot's path learning ability. Also, in a field of architecture, its pattern can be used to decorate a building. As we've talked previously, the maze can be used as a tool in various fields, and maze users in different fields may desire different properties on a maze. These days, a maze is not only used as a puzzle but also adopted in many different fields For example, a user may want a maze with numerous turns and branches in a robot contest, while others may want a maze with long straight passages and fewer dead-ends in a building decoration.

2. INTRODUCTION

Puzzle that has a simple goal: “Find a path from an entrance to a goal of the maze.” Because of its simple and intuitive goal, children can play mazes without a deep understanding of its rules. But, for a certain maze, it can also have much difficult level so that even adults need to spend several hours to solve it. Due to its simple goal and challenging aspects, maze puzzles are being enjoyed regardless of one’s age. Hence, this work focuses on the research problem: in different fields, when users have different desired properties on a maze, how can we generate a maze that best fits their own desired properties. It does not have a loop and an inaccessible area. To generate the desired perfect maze, we apply search-based procedural content generation (SBPCG) approach. In this approach it has been implemented using existing maze generation algorithms, the process repeatedly generates mazes and evaluate them to check whether they have properties close to desired properties. However, some maze generation algorithms have different characteristics over sampled mazes. One algorithm has a higher probability to have a maze with long straight ways, and another algorithm has a higher probability to have a maze with many branches. Thus, applying several maze generation algorithms in this approach is not trivial as simply choosing any algorithm. In this paper, our research shows how the maze generation algorithms can be used appropriately in this approach to generate a maze efficiently when desired properties are given by users.

3. ARCHITECTURE



The approach in this architecture searches for contents meeting a given desired criteria by iteratively sampling input parameters used in procedural content generation (PCG) algorithms. In the searching process, when a content is generated by PCG algorithm, the process scores the content using a fitness function. After the searching process stops, the content with the highest score is returned as the best one. The flow chart of SBPCG approach is to find a maze with desired properties. When desired metric values are given as desired properties, the approach samples input parameters of maze generation algorithms, pseudo-random numbers, to generate a maze. Then, the maze's distance to the desired properties is evaluated. To evaluate the distance, we measure metrics from the maze and calculate the distance between the measured metrics and desired metrics. For example, let's assume that we desire 5 turns and 11 straights on a maze. When the process obtains a maze, if the maze has 6 turns and 15 straights, the distance of the maze to desired properties is given by $4.12 = (6 - 5) + (15 - 11)$. If the distance is less than given threshold α , the maze is stored. The process repeats until it meets a given termination criterion. After the process stops, users can choose the best maze among the stored maze.

3.1 PERFECT MAZE

Our research domain is a rectangular perfect maze. As shown in Figure 3.1, the perfect maze has no loop and no inaccessible area. In this section, we explain our research domain, perfect maze, with more detail.

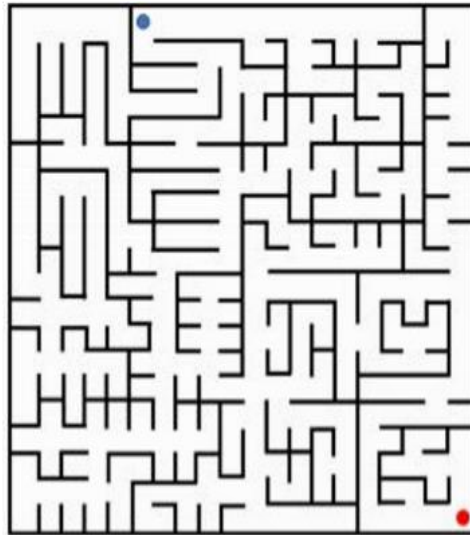


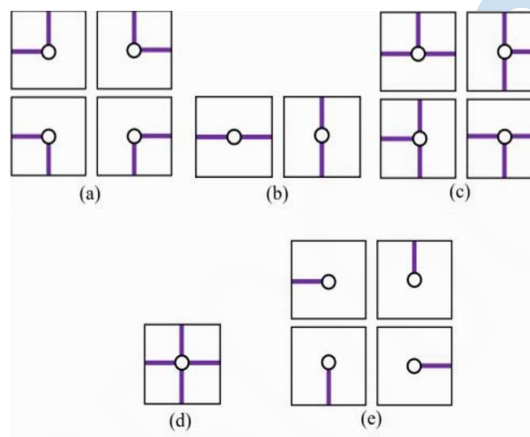
Figure 3.1 An example of a rectangular perfect maze with the start point (blue point) and the end point (red point).

3.2 PERFECT MAZE COMPONENTS

Here, several components of a perfect maze has been listed.

- a) Start point and End Point: The start point is a point where a player starts to solve a maze. The end point is a point where a player needs to arrive to finish a maze puzzle.
- b) Solution Path: The solution path is a path between the start point and the end point. Since a perfect maze has no loop, there is unique solution path.
- c) Dead-End Tree: Paths on a perfect maze except the solution path are dead-end trees. One maze can have several dead-end trees, and they are branched from the solution path. In our research, we define three types of dead-end tree.

- d) Forward Dead-End: The forward dead-end is a dead-end tree forwarding to the end point. It provides illusion that this dead-end tree is leading to the end point.
- e) Backward Dead-End: The backward dead-end is a dead-end tree that turns away from the end point. It has opposite directional property compared to forward dead-end.
- f) Alcove: The alcove is single dead-end only with straight path without any turn.



4. WORKING

The main working of path finding game is to find out the path from given place to another place by using the movement of point. We use the special key button for the movement of point. The left key button is used to movement of point along the X-axis as the value decreases, The right key button is used to movement of point along the X-axis as the value increases, The up key button is used to movement of point along the y-axis as the value increases, the down key button is used to movement of point along the y-axis as the value decreases. There is also be given the time limitation, so it is necessary to find out the path within a given time interval.

5. IMPLEMENTATION

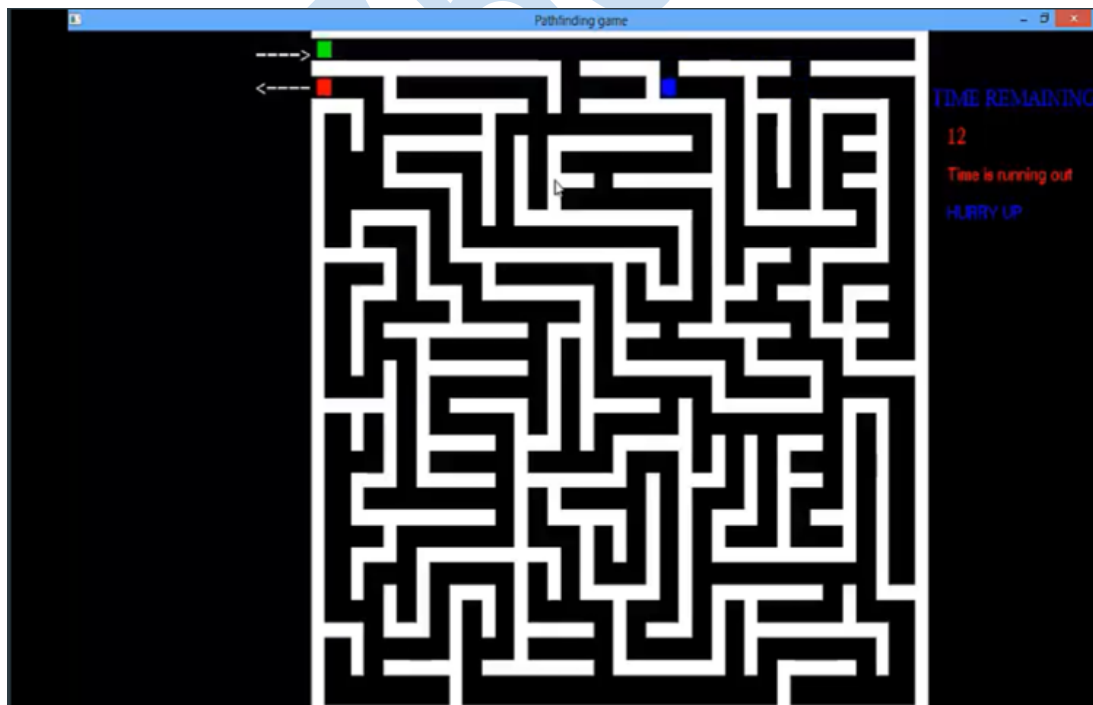
The implementation has been carried using

- Language used : C++
- Platform : Dev-c++, visual studio
- Library : OpenGL

OPENGL - OpenGL (Open Graphics Library) - a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics

The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

5.1 OUTPUT SCREENSHOT



6. CONCLUSION AND FUTURE WORKS

Thus our work provides a method to generate desired mazes, when desired properties are given by users. For users, they are allowed to set up desired metric values for maze topology and also insert topological constraints directly over a maze. Existing maze generation algorithms are utilized with SBPCG approach, and our method chooses the best algorithm amongst using histograms of the algorithms. Then, the chosen algorithm is used in SBPCG approach to generate desired mazes effectively.

REFERENCES

1. Christopher Berg. AmazeingArt. <http://amazeingart.com/>
2. Adrian Fisher. Maze Maker. <http://mazemaker.com>
3. Walter D. Pullen. ThinkLabyrinth! <http://www.astrolog.org/labyrnth.htm>
4. Jamis Buck. 2015. Mazes for Programmers, Code Your Own Twisty Little Passages. The Pragmatic Bookshelf.
5. Jamis Buck: HTML 5 Presentation with Demos of Maze Generation Algorithms. www.jamisbuck.org/presentations/rubyconf2011/index.html
6. Steve LaValle. TheRRTPage. <http://msl.cs.uiuc.edu/rrt/index.html>
7. Jie Xu and Craig S. Kaplan. 2006. Vortex maze construction. Journal of Mathematics and the Arts 1(November 2006).
8. Jie Xu and Craig S. Kaplan. 2007. Image-guided maze construction. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2007 26,3(July 2007).
9. Hans Pedersen and Karan Singh. 2006. Organic Labyrinths and Mazes. In Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering (NPAR '06). ACM, New York, NY, USA, 79– 86.

10. Wen-Shou Chou. 2016. Rectangular Maze Construction by Combining Algorithms and Designed Graph Patterns. GSTF Journal on Computing (JOC) 5 (August 2016),35–39.

2017506601