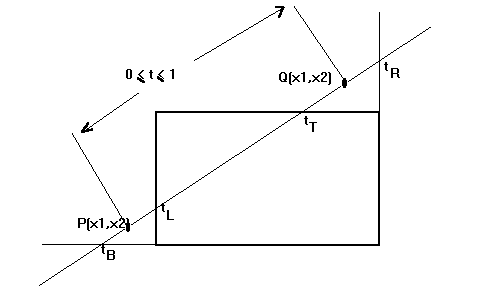
**Experiment- 8**

**Aim**

To implement Liang-Barsky Line Clipping Algorithm

**Description/Theory**

Liang and Barsky have created an algorithm that uses floating-point arithmetic but finds the appropriate end points with at most four computations. This algorithm uses the parametric equations for a line and solves four inequalities to find the range of the parameter for which the line is in the viewport. This algorithm is significantly more efficient than Cohen–Sutherland. The idea of the Liang–Barsky clipping algorithm is to do as much testing as possible before computing line intersections.



Let P(x1,y1) Q(x2,y2) be the line which we want to study. The **parametric equation of the line segment** from gives x-values and y-values for every point in terms of a **parameter t**that ranges from 0 to 1. The equations are:

x = x1 + (x2-x1) \* t = x1 + dx\*t

y = y1 + (y2-y1)\*t = y1 + dy\*t

We can see that when t = 0, the point computed is P(x1,y1); and when t = 1, the point computed is Q(x2,y2).

**Algorithm**

The algorithm for Liang-Barsky Line Clipping Algorithm is given below:

1. Set tmin = 0 and tmax = 1
2. Calculate the values of tL, tR, tT, and tB (tvalues).
   * if t < tmin or t > tmax ignore it and go to the next edge
   * otherwise classify the **t**value as entering or exiting value (using inner product to classify)
   * if **t** is entering value set tmin = t; if **t** is exiting value set tmax = t
3. If https://www.cs.helsinki.fi/group/goa/viewing/leikkaus/l9.giftmin < tmax then **draw a line** from (x1 + dx\*tmin, y1 + dy\*tmin) to (x1 + dx\*tmax, y1 + dy\*tmax)
4. If the line crosses over the window, you will see (x1 + dx\*tmin, y1 + dy\*tmin) and (x1 + dx\*tmax, y1 + dy\*tmax) are intersection between line and edge.

**Code**

#include<stdio.h>

#include<graphics.h>

#include<math.h>

#include<conio.h>

#include<dos.h>

int main()

{

int i, gd = DETECT, gm;

int x1 = 120, y1 = 120, x2 = 300, y2 = 300, xmin = 100, xmax = 250, ymin = 100, ymax = 250;

int xx1, xx2, yy1, yy2, dx, dy;

float t1, t2, p[4], q[4], temp;

initgraph(&gd, &gm, "");

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);

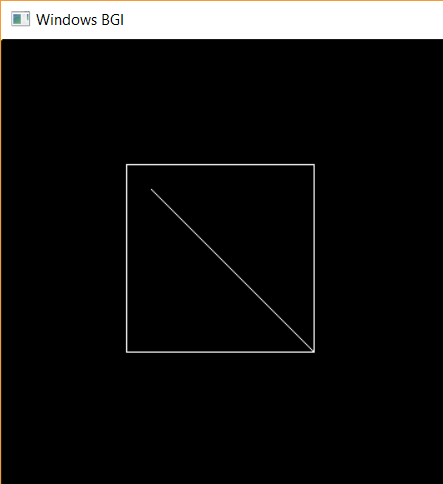
}

getch();

closegraph();

}

**Result**



**Discussion**

Liang-Barsky algorithm calculates two values of Parameter t : t1 and t2 that define that part of the line that lies within the clip rectangle. The value of t1 is determined by checking the rectangle edges for which the line proceeds from the outside to the inside (p <0). The value of t1 is taken as a largest value amongst various values of intersections with all edges. On the other hand, the value of t2 is determined by checking the rectangle edges for which the line proceeds from the inside to the outside (p > 0). The minimum of the calculated value is taken as a value for t2.

Now, if t1 > t2, the line is completely outside the clipping window and it can be rejected. Otherwise the values of t1 and t2 are substituted in the parametric equations to get the end points of the clipped line.

**Finding and Learning**

Upon studying the Liang-Barsky Line Clipping Algorithm, we get to learn about two of its advantages, given below:

1. More efficient than other algorithms as line intersection with boundaries calculations are reduced.
2. Intersections of line are computed only once.

**Experiment- 9**

**Aim**

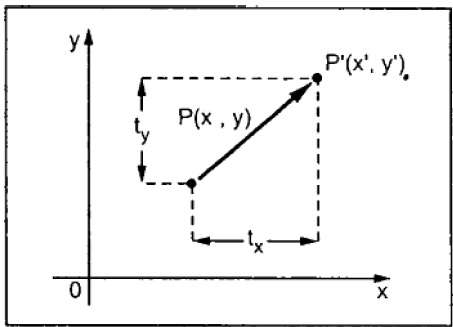
To perform Translation on a given object.

**Description/Theory**

In computer graphics, we have seen how to draw some basic figures like line and circles. In this post we will discuss on basics of an important operation in computer graphics as well as 2-D geometry, which is **transformation**.  
In computer graphics, transformation of the coordinates consists of three major processes:

* Translation
* Rotation
* Scaling

A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (tx, ty) to the original coordinate (X, Y) to get the new coordinate (X’, Y’).



**Algorithm**

If point (X, Y) is to be translated by amount Dx and Dy to a new location (X’, Y’) then new coordinates can be obtained by adding Dx to X and Dy to Y as:

X' = Dx + X

Y' = Dy + Y

or P' = T + P where

P' = (X', Y'),

T = (Dx, Dy ),

P = (X, Y)

Here, P(X, Y) is the original point. T(Dx, Dy) is the **translation factor**, i.e. the amount by which the point will be translated. P'(X’, Y’) is the coordinates of point P after translation.

**Code**

/\*The code below performs translation on rectangular structure\*/

#include<bits/stdc++.h>

#include<graphics.h>

#include<conio.h>

using namespace std;

void translateRectangle ( int P[][2], int T[])

{

int gd = DETECT, gm;

initgraph (&gd, &gm, "");

setcolor (2);

rectangle (P[0][0], P[0][1], P[1][0], P[1][1]);

P[0][0] = P[0][0] + T[0];

P[0][1] = P[0][1] + T[1];

P[1][0] = P[1][0] + T[0];

P[1][1] = P[1][1] + T[1];

setcolor(3);

rectangle (P[0][0], P[0][1], P[1][0], P[1][1]);

}

int main()

{

int P[2][2] = { {150, 80} , {225, 180}};

int T[] = {30, 50};

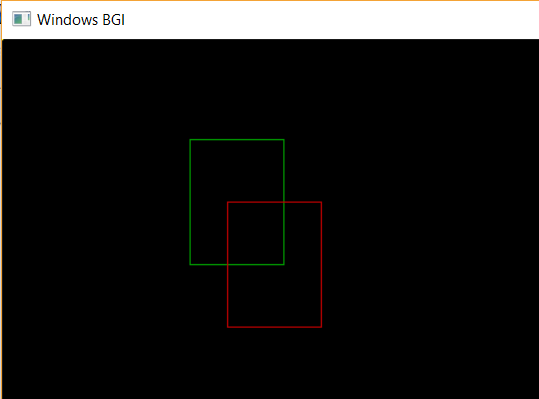
translateRectangle (P, T);

getch();

return 0;

}

**Result**



**Discussion**

A translation process is the simplest operation among transformation of coordinates. It moves every point a constant distance in a specified direction. It can be described as a rigid motion. A translation can also be interpreted as the addition of a constant vector to every point, or as shifting the origin of the coordinate system.

**Finding and Learning**

Whenever we perform translation of any object we simply translate its each and every point. Some of basic objects along with their translation can be drawn as:

* **Point Translation P(X, Y) :** Here we only translate the x and y coordinates of given point as per given translation factor dx and dy respectively.
* **Line Translation:** The idea to translate a line is to translate both of the end points of the line by the given translation factor(dx, dy) and then draw a new line with inbuilt graphics function.
* **Rectangle Translation :** Here we translate the x and y coordinates of both given points A(top left ) and B(bottom right) as per given translation factor dx and dy respectively and then draw a rectangle with inbuilt graphics function