Line Clipping | Set 2 (Cyrus Beck Algorithm)

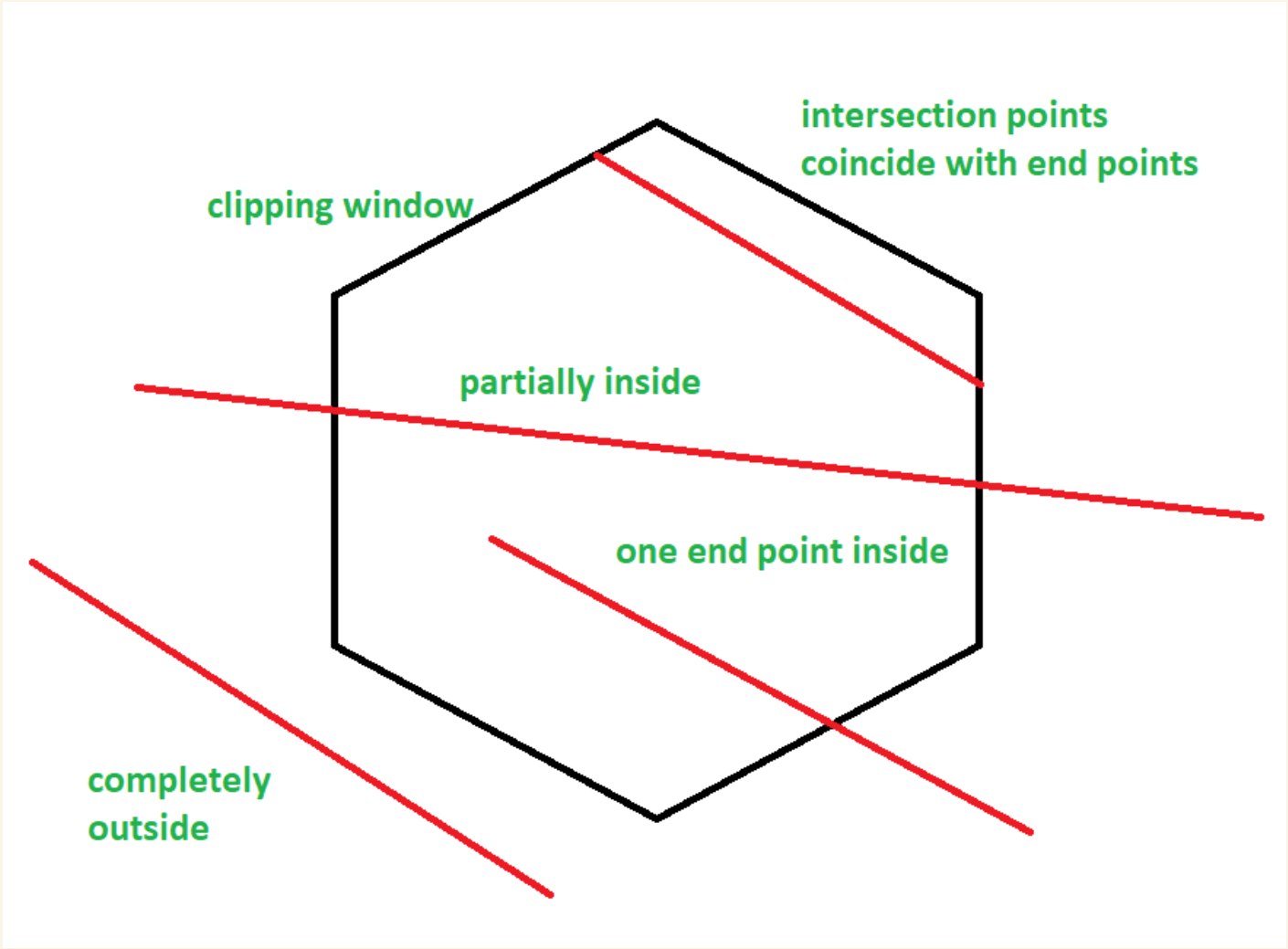
Background:
Cyrus Beck is a line clipping algorithm that is made for convex polygons. It allows line clipping for non-rectangular windows, unlike Cohen Sutherland or **Nicholl Le Nicholl**. It also removes the repeated clipping needed in Cohen Sutherland.

```
Input:
 1. Convex area of interest
    which is defined by a set of coordinates
    given in a clockwise fashion.
 2. vertices which are an array of coordinates:
    consisting of pairs (x, y)
 3. n which is the number of vertices
 4. A line to be clipped
    given by a set of coordinates.
 5. line which is an array of coordinates:
    consisting of two pairs, (x0, y0) and (x1, y1)
Output:
 1. Coordinates of line clipping which is the Accepted clipping
 2. Coordinates (-1, -1) which is the Rejected clipping
```

Algorithm:

- Normals of every edge is calculated.
- Vector for the clipping line is calculated.
- Dot product between the difference of one vertex per edge and one selected end point of the clipping line and the normal of the edge is calculated (for all edges).
- Dot product between the vector of the clipping line and the normal of edge (for all edges) is calculated.
- The former dot product is divided by the latter dot product and multiplied by -1. This is ‘t’.
- The values of ‘t’ are classified as entering or exiting (from all edges) by observing their denominators (latter dot product).
- One value of ‘t’ is chosen from each group, and put into the parametric form of a line to calculate the coordinates.
- If the entering ‘t’ value is greater than the exiting ‘t’ value, then the clipping line is rejected.

Cases:



1. **Case 1:** The **line is partially inside** the clipping window:

$0 < t_E < t_L < 1$

where t_E is 't' value for entering intersection point
t_L is 't' value for exiting intersection point

6. **Case 2:** The **line has one point inside or both sides inside** the window **or the intersection points are on the end points of the line**:

$$0 \leq t_E \leq t_L \leq 1$$

7. **Case 3:** The line is **completely outside** the window:

$$t_L < t_E$$

Pseudocode:

First, calculate the parametric form of the line to be clipped and then follow the algorithm.

- Choose a point called P_1 from the two points of the line (P_0P_1).

- Now for each edge of the polygon, calculate the normal pointing away from the centre of the polygon, namely N_1, N_2, etc.
- Now for each edge choose P_{Ei} (i -> i^{th} edge) (choose any of the vertices of the corresponding edge, eg.: For polygon ABCD, for side AB, P_{Ei} can be either point A or point B) and calculate

$P_0 - P_{Ei}$

- Then calculate

$P_1 - P_0$

- Then calculate the following dot products for each edge:

$N_i \cdot (P_0 - P_{Ei})$
$N_i \cdot (P_1 - P_0)$

where i -> i^{th} edge of the convex polygon

- Then calculate the corresponding 't' values for each edge by:

$$t = \frac{N_i \cdot (P_0 - P_{Ei})}{-(N_i \cdot (P_1 - P_0))}$$

- Then club the 't' values for which the $\mathbf{N_i \cdot (P_1 - P_0)}$ came out to be negative and take the minimum of all of them and 1.
- Similarly club all the 't' values for which the $N_i \cdot (P_1 - P_0)$ came out to be positive and take the maximum of all of the clubbed 't' values and 0.
- Now the two 't' values obtained from this algorithm are plugged into the parametric form of the 'to be clipped' line and the resulting two points obtained are the clipped points.

Implementation: Here is an implementation of the above steps in SFML C++ Graphics Library. You can also press any key to unclip the line and press any key to clip the line.

```
// C++ Program to implement Cyrus Beck

#include <SFML/Graphics.hpp>
#include <iostream>
#include <utility>
#include <vector>

using namespace std;
using namespace sf;

// Function to draw a line in SFML
void drawline(RenderWindow* window, pair<int, int> p0, pair<int, int> p1)
```

```
{
    Vertex line[] = {
        Vertex(Vector2f(p0.first, p0.second)),
        Vertex(Vector2f(p1.first, p1.second))
    };
    window->draw(line, 2, Lines);
}

// Function to draw a polygon, given vertices
void drawPolygon(RenderWindow* window, pair<int, int> vertices[], int n)
{
    for (int i = 0; i < n - 1; i++)
        drawline(window, vertices[i], vertices[i + 1]);
    drawline(window, vertices[0], vertices[n - 1]);
}

// Function to take dot product
int dot(pair<int, int> p0, pair<int, int> p1)
{
    return p0.first * p1.first + p0.second * p1.second;
}

// Function to calculate the max from a vector of floats
float max(vector<float> t)
{
    float maximum = INT_MIN;
    for (int i = 0; i < t.size(); i++)
        if (t[i] > maximum)
            maximum = t[i];
    return maximum;
}

// Function to calculate the min from a vector of floats
float min(vector<float> t)
{
    float minimum = INT_MAX;
    for (int i = 0; i < t.size(); i++)
        if (t[i] < minimum)
            minimum = t[i];
    return minimum;
}

// Cyrus Beck function, returns a pair of values
// that are then displayed as a line
pair<int, int>* CyrusBeck(pair<int, int> vertices[],
                          pair<int, int> line[], int n)
{

    // Temporary holder value that will be returned
    pair<int, int>* newPair = new pair<int, int>[2];

    // Normals initialized dynamically(can do it statically also, doesn't matter)
    pair<int, int>* normal = new pair<int, int>[n];

    // Calculating the normals
```

```
for (int i = 0; i < n; i++) {
    normal[i].second = vertices[(i + 1) % n].first - vertices[i].first;
    normal[i].first = vertices[i].second - vertices[(i + 1) % n].second;
}

// Calculating P1 - P0
pair<int, int> P1_P0
    = make_pair(line[1].first - line[0].first,
                line[1].second - line[0].second);

// Initializing all values of P0 - PEi
pair<int, int>* P0_PEi = new pair<int, int>[n];

// Calculating the values of P0 - PEi for all edges
for (int i = 0; i < n; i++) {

    // Calculating PEi - P0, so that the
    // denominator won't have to multiply by -1
    P0_PEi[i].first
        = vertices[i].first - line[0].first;

    // while calculating 't'
    P0_PEi[i].second = vertices[i].second - line[0].second;
}

int *numerator = new int[n], *denominator = new int[n];

// Calculating the numerator and denominators
// using the dot function
for (int i = 0; i < n; i++) {
    numerator[i] = dot(normal[i], P0_PEi[i]);
    denominator[i] = dot(normal[i], P1_P0);
}

// Initializing the 't' values dynamically
float* t = new float[n];

// Making two vectors called 't entering'
// and 't leaving' to group the 't's
// according to their denominators
vector<float> tE, tL;

// Calculating 't' and grouping them accordingly
for (int i = 0; i < n; i++) {

    t[i] = (float)(numerator[i]) / (float)(denominator[i]);

    if (denominator[i] > 0)
        tE.push_back(t[i]);
    else
        tL.push_back(t[i]);
}

// Initializing the final two values of 't'
float temp[2];
```

```
    // Taking the max of all 'tE' and 0, so pushing 0
    tE.push_back(0.f);
    temp[0] = max(tE);

    // Taking the min of all 'tL' and 1, so pushing 1
    tL.push_back(1.f);
    temp[1] = min(tL);

    // Entering 't' value cannot be
    // greater than exiting 't' value,
    // hence, this is the case when the line
    // is completely outside
    if (temp[0] > temp[1]) {
        newPair[0] = make_pair(-1, -1);
        newPair[1] = make_pair(-1, -1);
        return newPair;
    }

    // Calculating the coordinates in terms of x and y
    newPair[0].first
        t
        = (float)line[0].first
          + (float)P1_P0.first * (float)temp[0];
    newPair[0].second
        = (float)line[0].second
          + (float)P1_P0.second * (float)temp[0];
    newPair[1].first
        = (float)line[0].first
          + (float)P1_P0.first * (float)temp[1];
    newPair[1].second
        = (float)line[0].second
          + (float)P1_P0.second * (float)temp[1];
    cout << '(' << newPair[0].first << ", "
         << newPair[0].second << ") ("
         << newPair[1].first << ", "
         << newPair[1].second << ")";


    return newPair;
}

// Driver code
int main()
{

    // Setting up a window and loop
    // and the vertices of the polygon and line
    RenderWindow window(VideoMode(500, 500), "Cyrus Beck");
    pair<int, int> vertices[]
        = { make_pair(200, 50),
            make_pair(250, 100),
            make_pair(200, 150),
            make_pair(100, 150),
            make_pair(50, 100),
            make_pair(100, 50) };

    // Make sure that the vertices
    // are put in a clockwise order
    int n = sizeof(vertices) / sizeof(vertices[0]);
```

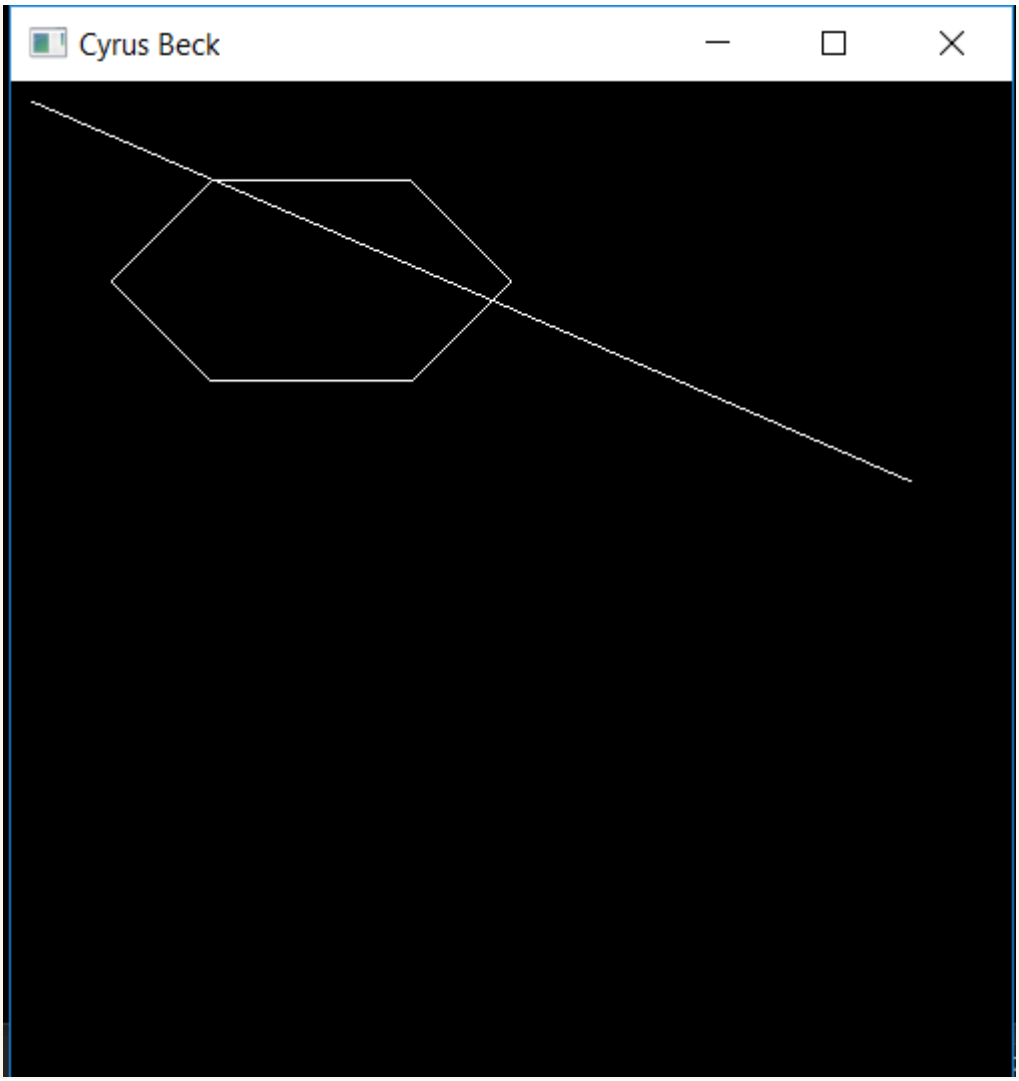
```
    pair<int, int> line[] = { make_pair(10, 10), make_pair(450, 200) };
    pair<int, int>* temp1 = CyrusBeck(vertices, line, n);
    pair<int, int> temp2[2];
    temp2[0] = line[0];
    temp2[1] = line[1];

    // To allow clipping and unclipping
    // of the line by just pressing a key
    bool trigger = false;
    while (window.isOpen()) {
        window.clear();
        Event event;
        if (window.pollEvent(event)) {
            if (event.type == Event::Closed)
                window.close();
            if (event.type == Event::KeyPressed)
                trigger = !trigger;
        }
        drawPolygon(&window, vertices, n);

        // Using the trigger value to clip
        // and unclip a line
        if (trigger) {
            line[0] = temp1[0];
            line[1] = temp1[1];
        }
        else {
            line[0] = temp2[0];
            line[1] = temp2[1];
        }
        drawline(&window, line[0], line[1]);
        window.display();
    }
    return 0;
}
```

Show less

Output:

```
(102, 50) (240, 109)
```

- **Before Clipping:**


Cyrus Beck

- **After Clipping:**