PES University

Cloud Computing and Big Data

POLYGON GENERATOR

GitHub: https://github.com/Dhruval360/Random-Polygon-Generator

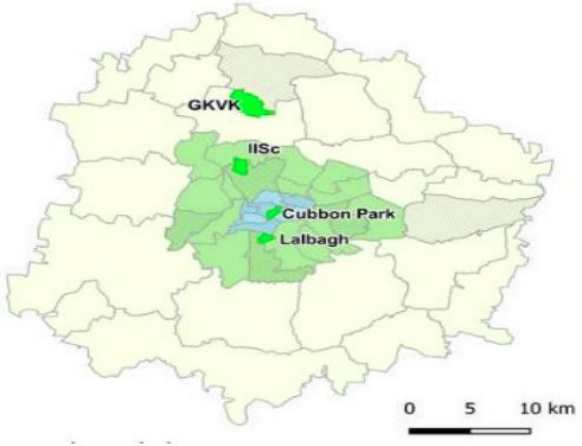
TEAM DETAILS

PESU CCBD

SRN	NAME	SEMESTER	SECTION	BRANCH	CAMPUS
PES1UG19CS123	CHANDRADHAR RAO	4	B	CSE	RR
PES1UG19CS272	MIHIR MADHUSUDAN KESTUR	4	E	CSE	RR
PES1UG19CS313	P B DHRUVAL	4	E	CSE	RR

OBJECTIVES

- Generate random polygons (non intersecting and without holes) using different algorithms
- Analyse the timing of different algorithms
- Analyse the size of WKT file storing the polygons



map.wkt

```
POLYGON ((3.000000 3.000000, 3.000000 3.000000, 3.000000 3.000000, 3.000000 3.000000, 3.000000 3.000000, 3.000000 3.000000, 3.000000
POLYGON ((6.016495 5.662020, 5.866161 5.706738, 5.880958 5.738233, 6.534220 5.587014, 6.656834 5.527001, 6.153739 5.860202, 6.068405
POLYGON ((4.050073 4.295337, 4.161233 4.258084, 4.061256 4.267348, 4.068874 4.179169, 4.020436 4.174149, 4.054114 4.164256, 4.180512
POLYGON ((1.269102 3.944558, 1.051136 3.831131, 1.271583 3.708642, 1.102496 3.632396, 1.234552 3.575039, 1.250266 3.597622, 1.315973
POLYGON ((6.000000 6.000000, 6.000000 6.000000, 6.000000 6.000000, 6.000000 6.000000, 6.000000 6.000000, 6.000000 6.000000, 6.000000
POLYGON ((3.645478 2.574716, 3.484147 2.966303, 3.226194 3.637231, 3.430134 3.804988, 3.562211 2.722423, 3.495646 2.891761, 3.174326
POLYGON ((6.819064 6.748944, 6.731159 6.607250, 6.396247 6.720903, 5.557959 6.936055, 5.333042 6.937072, 5.413842 6.831517, 6.222337
POLYGON ((4.601198 4.858209, 4.517963 4.905807, 4.094531 4.862153, 4.100474 4.417200, 4.375969 4.351378, 4.047889 4.102855, 4.428218
POLYGON ((6.668432 6.256512, 6.202387 6.744470, 5.568880 6.183278, 5.665567 6.028719, 5.003452 5.799153, 5.745844 5.471411, 6.229668
POLYGON ((5.698564 5.381813, 5.589810 5.566075, 5.176775 5.150106, 4.984689 5.132500, 5.450324 5.561799, 5.499614 5.652387, 5.183745
POLYGON ((2.547590 2.305417, 2.599972 2.318905, 2.565708 2.285605, 2.647358 2.282142, 2.652512 2.255288, 2.603121 2.253795, 2.607093
POLYGON ((4.346180 4.928948, 3.501299 4.931143, 3.952465 4.942447, 4.262262 4.958766, 3.921970 4.958097, 3.788600 4.973923, 3.996615
POLYGON ((5.920400 7.150467, 5.738882 7.041210, 5.710901 7.018229, 6.515934 7.233451, 6.327970 7.164103, 5.655576 6.990081, 6.367360
POLYGON ((4.577381 4.585475, 4.578696 4.607094, 4.509944 4.593044, 4.485175 4.585867, 4.454347 4.625699, 4.510353 4.657525, 4.528510
POLYGON ((5.413683 5.469432, 5.869431 5.699743, 5.764378 5.654165, 5.412444 5.501509, 5.317239 5.490523, 5.182331 5.393747, 4.986599
POLYGON ((5.815971 5.314485, 5.811034 5.320623, 5.893037 5.342424, 5.921187 5.360315, 5.939770 5.369902, 5.991300 5.389470, 5.938973
POLYGON ((5.000000 5.000000, 5.000000 5.000000, 5.000000 5.000000, 5.000000 5.000000, 5.000000 5.000000, 5.000000 5.000000, 5.000000
POLYGON ((3.131680 4.204995, 3.135251 3.861739, 3.023223 4.239276, 3.083972 3.011818, 3.132808 3.137274, 3.171994 3.272822, 3.184007
POLYGON ((5.221175 5.738131, 5.390305 5.462525, 5.398714 5.392472, 5.121495 5.878615, 5.119256 5.826915, 5.110218 5.787521, 5.012906
POLYGON ((4.839733 4.714834, 4.691576 4.741833, 4.873422 4.755387, 4.925864 4.864539, 4.910778 4.889437, 4.930733 4.878579, 4.950242
POLYGON ((3.719523 4.414508, 3.709864 4.396293, 3.644482 4.400635, 4.065413 3.748374, 4.053667 3.714054, 3.930893 3.796596, 3.871110
POLYGON ((5.703848 6.346062, 5.958159 6.097382, 5.951052 6.147283, 6.027563 6.074836, 6.038290 6.037444, 6.236490 5.869557, 6.084408
POLYGON ((4.222158 4.009841, 4.258248 4.047883, 4.336661 4.102545, 4.468419 4.023914, 4.312592 4.007067, 4.561538 4.009306, 4.539534
POLYGON ((4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000
POLYGON ((4.872015 4.685429, 5.104323 4.823209, 4.989517 4.895902, 5.007543 4.935266, 4.884771 4.987826, 5.344780 4.973142, 5.296534
POLYGON ((4.011593 4.212391, 3.293233 4.913380, 3.310905 4.825663, 3.306174 4.413737, 3.560809 4.304825, 4.193526 3.827415, 4.275336
POLYGON ((4.553581 3.103048, 4.514343 3.230258, 4.579088 3.223679, 4.873499 3.156775, 4.567633 3.292666, 5.138670 3.247474, 5.280661
POLYGON ((4.094358 4.575940, 4.446750 4.042960, 4.361538 4.508307, 4.623705 4.105657, 5.368680 5.104590, 4.961767 4.431918, 5.357315
POLYGON ((4.227998 4.292259, 4.332369 4.372598, 4.409075 4.396394, 4.454563 4.455942, 4.778874 4.617695, 4.775208 4.654160, 4.972003
POLYGON ((5.072918 4.327086, 5.183295 4.419047, 4.987029 4.205920, 5.721485 4.889281, 5.710159 4.864480, 5.440057 4.544036, 5.616742
POLYGON ((4.368512 5.596766, 4.013548 5.569669, 4.037952 5.434188, 4.133618 5.457747, 4.068043 5.317913, 4.061124 5.309611, 4.379768
POLYGON ((4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000 4.000000, 4.000000
```

LITERATURE SURVEY

PESU CCBD

1. **Thomas Auer and Martin Held. 1996. Heuristics for the Generation of Random Polygons. In Proceedings of the 8th Canadian Conference on Computational Geometry. Carleton University Press, 38–43.**
    - The authors explore five algorithms to generate random polygons namely 2-opt moves, Steady Growth, Space partition, Quick star and Star universe. They conclude three algorithms 2-opt moves, Steady Growth and Space partition are suitable for practical purposes. For this project, Space Partition algorithm is implemented as the authors mention that it offers the best speed performance compared to the rest.
2. **Victor Hua, Generation of Simple Polygons, 2020, Final Project in Computational Geometry.**
    - The researchers present two algorithms for generating simple random non-intersecting polygons namely The “naive” algorithm and “improved” algorithm. Both need a convex hull of the random points to generate the polygon. For this project, the “naive” solution was implemented in C++ and the “improved” solution is kept for future work.

FEATURES

PESU CCBD

Our program allows the user to:

- Select the number of polygons to be generated
- Select the algorithm to be used (We have implemented 3 of them)
- Specify the file to which the polygons will be written to in WKT format
- Graph the generated polygons onto a single canvas (using OpenGL)
- View the status of program while running
- Graph the distribution of the generated polygons
- Select the length of the side of canvas
- A dedicated profiling mode wherein the program prints out metrics in the format:
    - Number of polygons generated, time taken, file size, algorithm used

We also provide a Profiler to Profile the algorithms and display graph of metrics like:

- **Number of polygons generated VS Size of WKT file**
- **Number of polygons generated VS Time taken**

Compilation and Features

PESU CCBD

```
dhruval@spyder:~/Desktop/Projects/CCBD/Random-Polygon-Generator$ make polygonGenerator -j7
g++ -O3 -DNDEBUG -c src/Graphics.cpp -o obj/Graphics.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/WKT_writer.cpp -o obj/WKT_writer.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/Driver.cpp -o obj/Driver.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/Polygon.cpp -o obj/Polygon.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/NaivePolygonGenerator.cpp -o obj/NaivePolygonGenerator.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/Polar.cpp -o obj/Polar.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -c src/SpacePartition.cpp -o obj/SpacePartition.o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread
g++ -O3 -DNDEBUG -o bin/polygonGenerator obj/Graphics.o obj/WKT_writer.o obj/Driver.o obj/Polygon.o obj/NaivePolygonGenerator.o obj/Polar.o obj/SpacePartition.
o -lm -lpopt -lglut -lGLU -lGL -fopenmp -lpthread && echo "Compiled Successfully!! Run the program using ./bin/polygonGenerator"
Compiled Successfully!! Run the program using ./bin/polygonGenerator
dhruval@spyder:~/Desktop/Projects/CCBD/Random-Polygon-Generator$ ./bin/polygonGenerator -?
Usage: [OPTION...]
  -n, --number_of_polygons=NUM     Set n = number of polygons that needs to be generated. [Default : n = 1]
  -v, --verbose=NUM                Set v = 1 for verbose output (will slow down the program by some time)
  -a, --algorithm=STR              Set a = polar or spacePartition or naivePoly to select the algorithm used to generate the polygons
  -g, --graph=NUM                  Set g = 1 to graph the generated polygons onto a single canvas (using OpenGL)
  -p, --profiling=NUM              Set p = 1 for profiling mode
  -f, --filename=STR               Enter the filename to which the generated polygons are to be written to in WKT format. [Default : map.wkt]
  -d, --distribution=NUM           Set d = 1 for obtaining an analysis of the distribution of the polygons generated
  -c, --canvas_size=NUM            Set the canvas size within which all the polygons will be generated. [Default : c = 1000]

Help options:
  -?, --help                       Show this help message
      --usage                      Display brief usage message
dhruval@spyder:~/Desktop/Projects/CCBD/Random-Polygon-Generator$
```

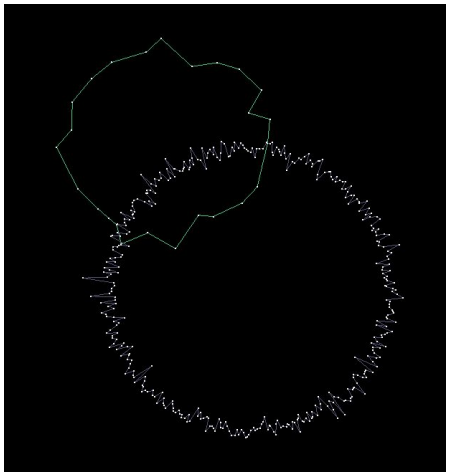
Sample commands

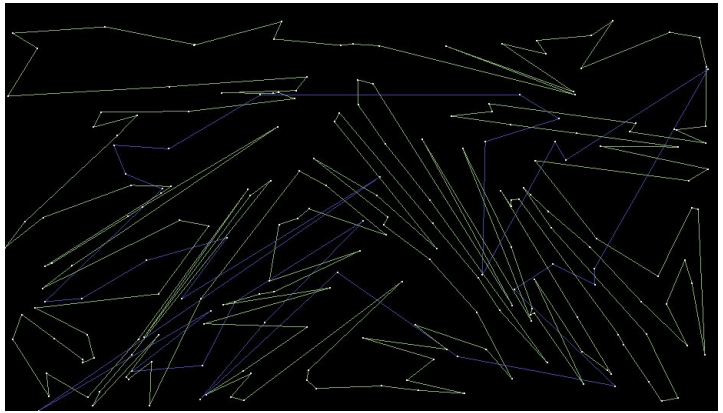
PESU CCBD

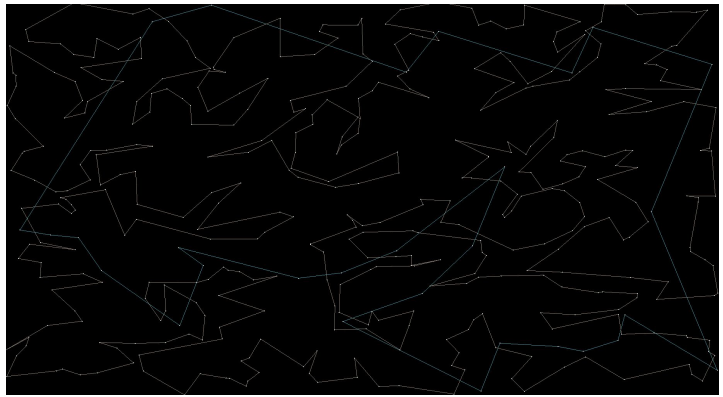
```
./bin/polygonGenerator -n 2 -v 1 -a polar -g 1 -p 1 -f polar.wkt -d 1
```

```
./bin/polygonGenerator -n 2 -v 1 -a spacePartition -g 1 -p 1 -f spacePartition.wkt -d 1
```

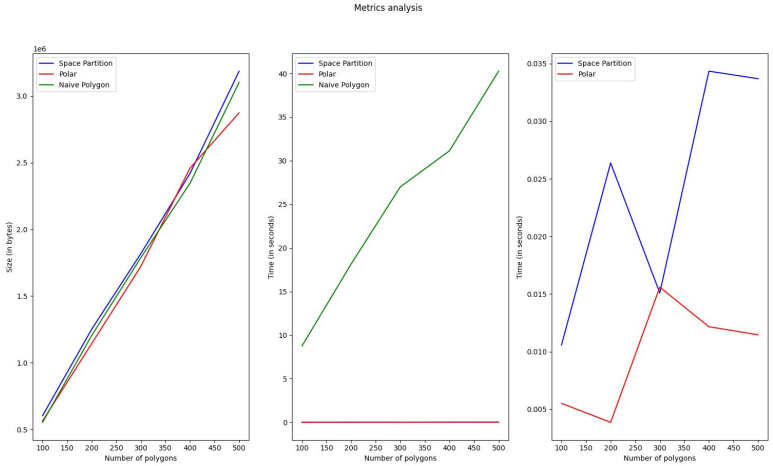
```
./bin/polygonGenerator -n 2 -v 1 -a naivePoly -g 1 -p 1 -f naivePoly.wkt -d 1
```







Sample Profiling




```
mihir@mihir-ThinkPad-T450s:~/PROJECTS/Random-Polygon-Generator$ ./Profiler.sh 5
Profiling started...
Iteration 1 (Generating 100 polygons using each algorithm)
        Executing Polar algorithm...
        Executing Space Partition algorithm...
        Executing Naive Polygon algorithm...
Finished generating 100 polygons using every algorithm. [20% complete]
-------------------------------------------------------------------------------------------
Iteration 2 (Generating 200 polygons using each algorithm)
        Executing Polar algorithm...
        Executing Space Partition algorithm...
        Executing Naive Polygon algorithm...
Finished generating 200 polygons using every algorithm. [40% complete]
-------------------------------------------------------------------------------------------
Iteration 3 (Generating 300 polygons using each algorithm)
        Executing Polar algorithm...
        Executing Space Partition algorithm...
        Executing Naive Polygon algorithm...
Finished generating 300 polygons using every algorithm. [60% complete]
-------------------------------------------------------------------------------------------
Iteration 4 (Generating 400 polygons using each algorithm)
        Executing Polar algorithm...
        Executing Space Partition algorithm...
        Executing Naive Polygon algorithm...
Finished generating 400 polygons using every algorithm. [80% complete]
-------------------------------------------------------------------------------------------
Iteration 5 (Generating 500 polygons using each algorithm)
        Executing Polar algorithm...
        Executing Space Partition algorithm...
        Executing Naive Polygon algorithm...
Finished generating 500 polygons using every algorithm. [100% complete]
-------------------------------------------------------------------------------------------
Profiling complete. Plotting the graphs...
```

DISTRIBUTION ANALYSIS

PESU CCBD

To calculate the distribution of polygons we,

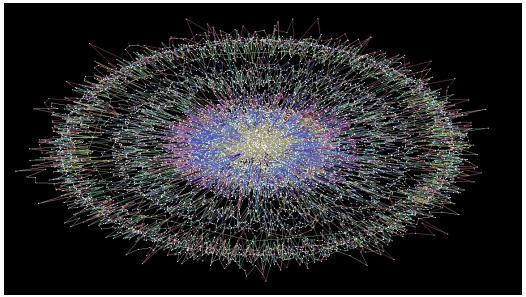
- Calculate the average values of x-coordinates and y-coordinates for each polygon to find the “average” center
- The distance from the origin to the coordinates of the “average” center is calculated for each polygon
- Plot the histogram of distances obtained for all the polygons on the map

Observation:

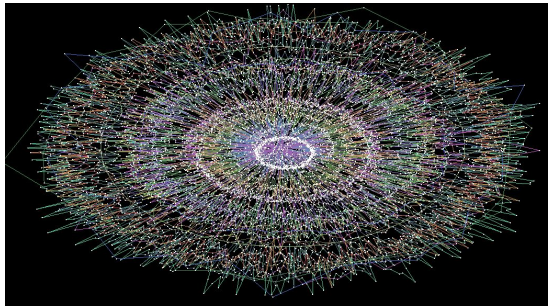
The distribution of the map of polygons depends on the pseudo-random generator function that generates random numbers, for various parameters of the polygons. Hence, multiple pseudo-random generators that sample from different distributions have been used.

Maps of polygons sampled from different distributions

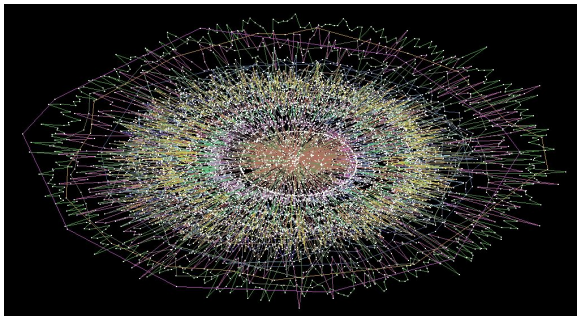
PESU CCBD



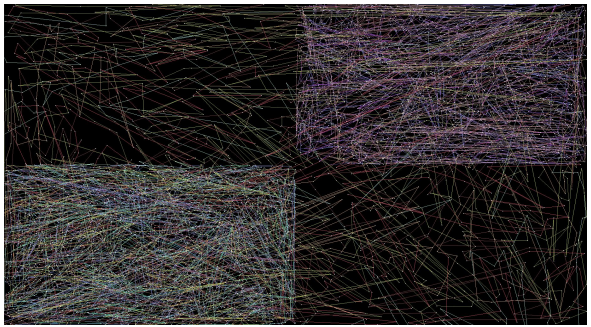
Geometric distribution



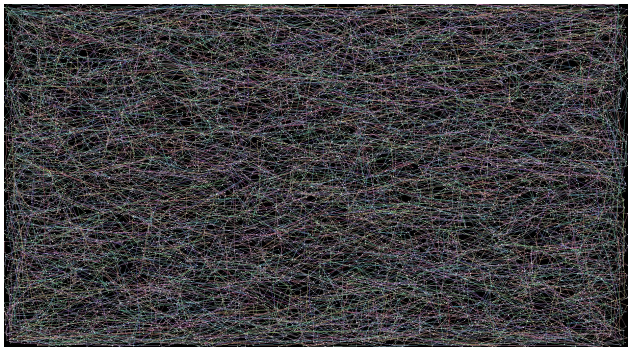
Poisson distribution



Binomial distribution



Normal distribution



Uniform distribution

Naive Polygon Generator From Convex Hull

PESU CCBD

Generating the convex hull

Convex Hull is a bounding shape that makes sure all the points are contained within the polygon and it has the ***smallest surface area***.
The algorithm is implemented here to generate the convex hull coordinates.
It is called the ***Graham Scan.***
Steps:

1. Find the bottom most point (in terms of y coordinates). Let that be the first point in the convex hull and called ***p0***.
2. The remaining points (n-1, if initially we had finite set of n points) are sorted in accordance with their polar angle from the anchor point ***p0***. If multiple points have same polar angle, then the one with the largest distance is considered since we want an inclusive bound.
3. If after sorting, the array size (let size be ***m***) is less than ***3,*** then no polygon can be formed.
4. If not, we create a stack ***stk*** and push the points with index ***0, 1, 2*** into the stack.
5. Now for the remaining ***m-3*** points, we consider the orientation of the top of the stack point, the point just below the top of the stack and the i^{th} point.
6. If the orientation is clockwise, then we reject the i^{th} point because we need a "***Convex***" hull and not a "***concave***" hull. Otherwise we keep the point and push it to the stack and repeat the process.

Naive Polygon Generator From Convex Hull

PESU CCBD

Algorithm to generate the polygons

This is the Brute Force Approach:

Steps:

1. We iterate over all the edges (let the starting vertex be ***A*** and end vertex be ***B***) and points (let one of them be ***P***) of the convex hull and find the pair with ***shortest*** point to line distance.
2. Consider removing the Edge ***AB*** and adding the edge ***AP*** and ***PB***. We check its validity (weather this pair would intersect with the already existing edges).
3. If its valid we include this point into the points of the polygon. If not we repeat the loop again and look for the next ***min distance***.
4. At the end of this loop, we would have constructed all the points of a ***valid Polygon*** without intersections.

Naive Polygon Generator From Convex Hull

PESU CCBD

Complexity Analysis

Graham Scan:

Sorting the points based on the polar angle would be the limiting factor. At best we could do it in ***O(nlogn)***. The loop is O(n) as a point is looked at only twice, once for right turn and once for left turn.

The Algorithm:

Finding distance between each edge and each point - $O(n^2)$

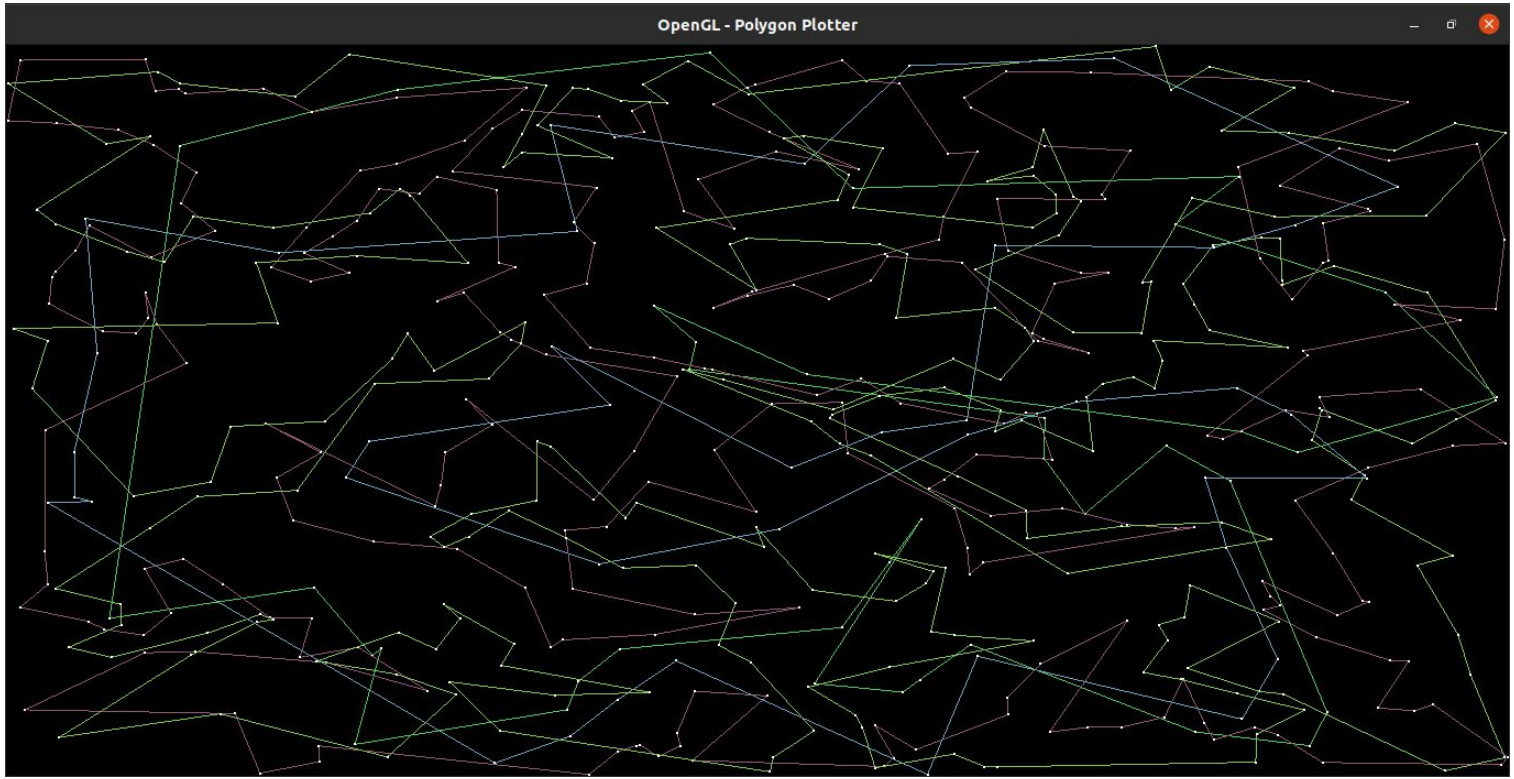
Checking for validity of the pair of edges against n edges - O(n)

Number of interior points - O(n)

Since each of this is nested inside the while loop, the overall complexity is ***$O(n^4)$***

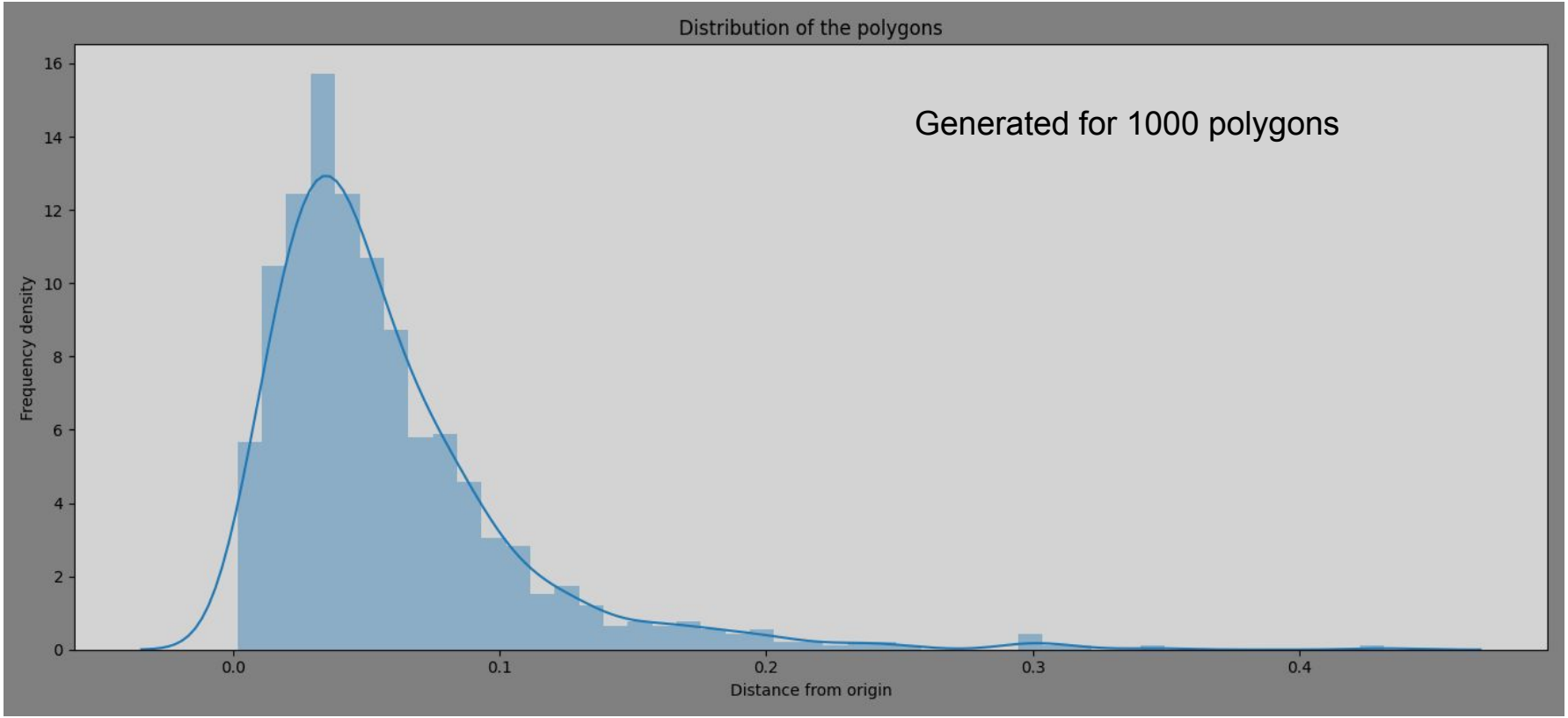
Naive Polygon Generator From Convex Hull

PESU CCBD



Generated for 4 polygons

Observed to follow Poisson distribution



Polar Generator

PESU CCBD

This algorithm takes 5 random parameters as input:

- A coordinate in the canvas (which denotes the average centre of the polygon)
- Average radius of the polygon
- Spike factor
- Irregularity factor
- Number of vertices of the polygon, **N**

Algorithm:

1. Initially a circle with a random radius, centred at a random location in the canvas is chosen.
2. Then **N** points on the circle's circumference are sampled. These points make up the polygon.
3. The Polar angles of these points is randomised based on the irregularity factor.
4. The polygon's vertices are then displaced from the circle's circumference based on the spike factor.

Note:

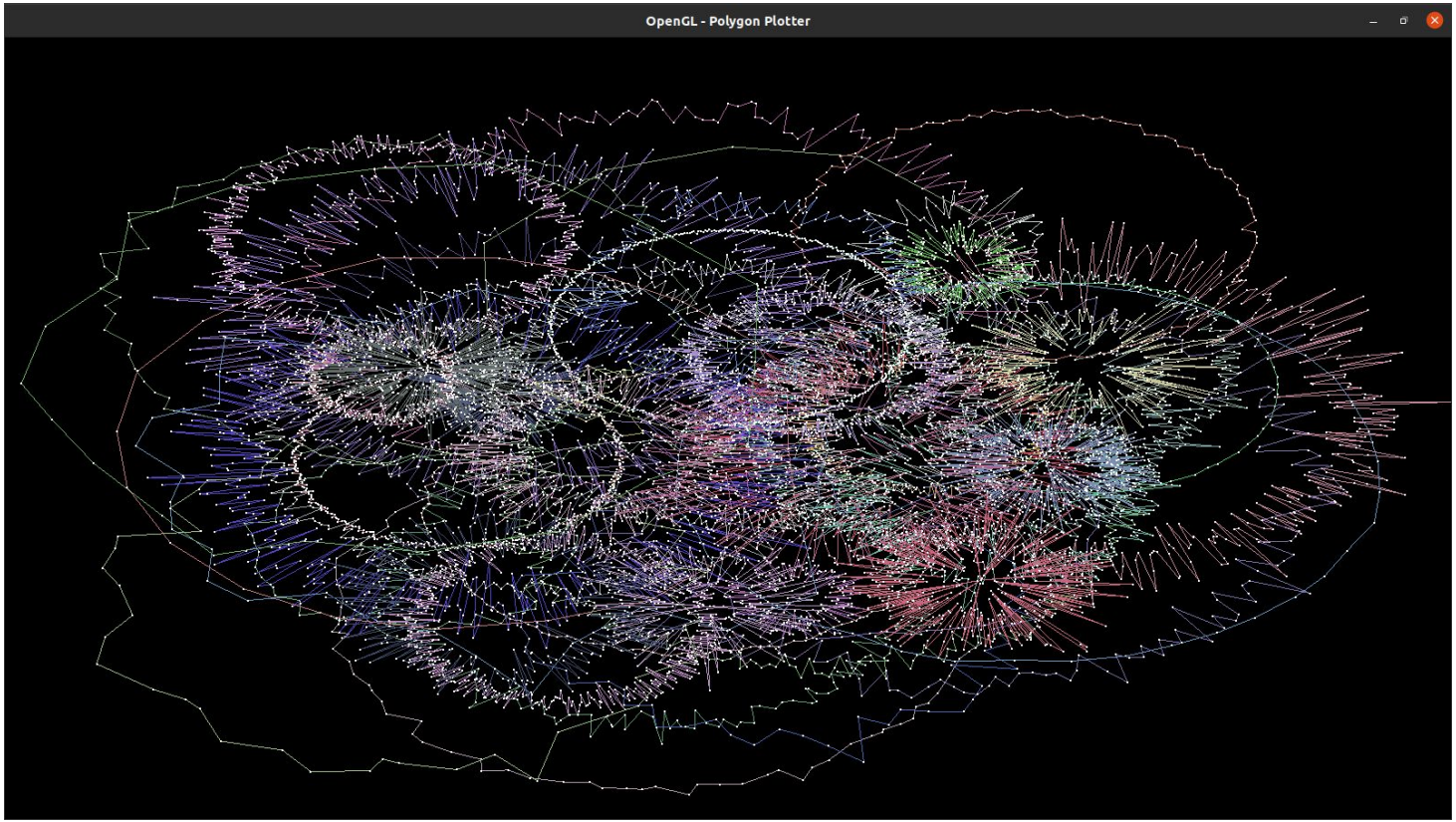
- If the irregularity factor and the spike factor are 0, a regular polygon is obtained
- As polar coordinates have been used while generating the polygon, this algorithm has been called as the Polar generator. It usually generates star like polygons.

Polar Generator

Time Complexity: O(n)

Explanation:
Generating **n** angle steps requires a for loop - O(n)
Normalizing the **n** angle steps requires a for loop - O(n)
Displacing the points based on spike value requires a for loop - O(n)
Hence the Overall Time Complexity of the algorithm is O(n)

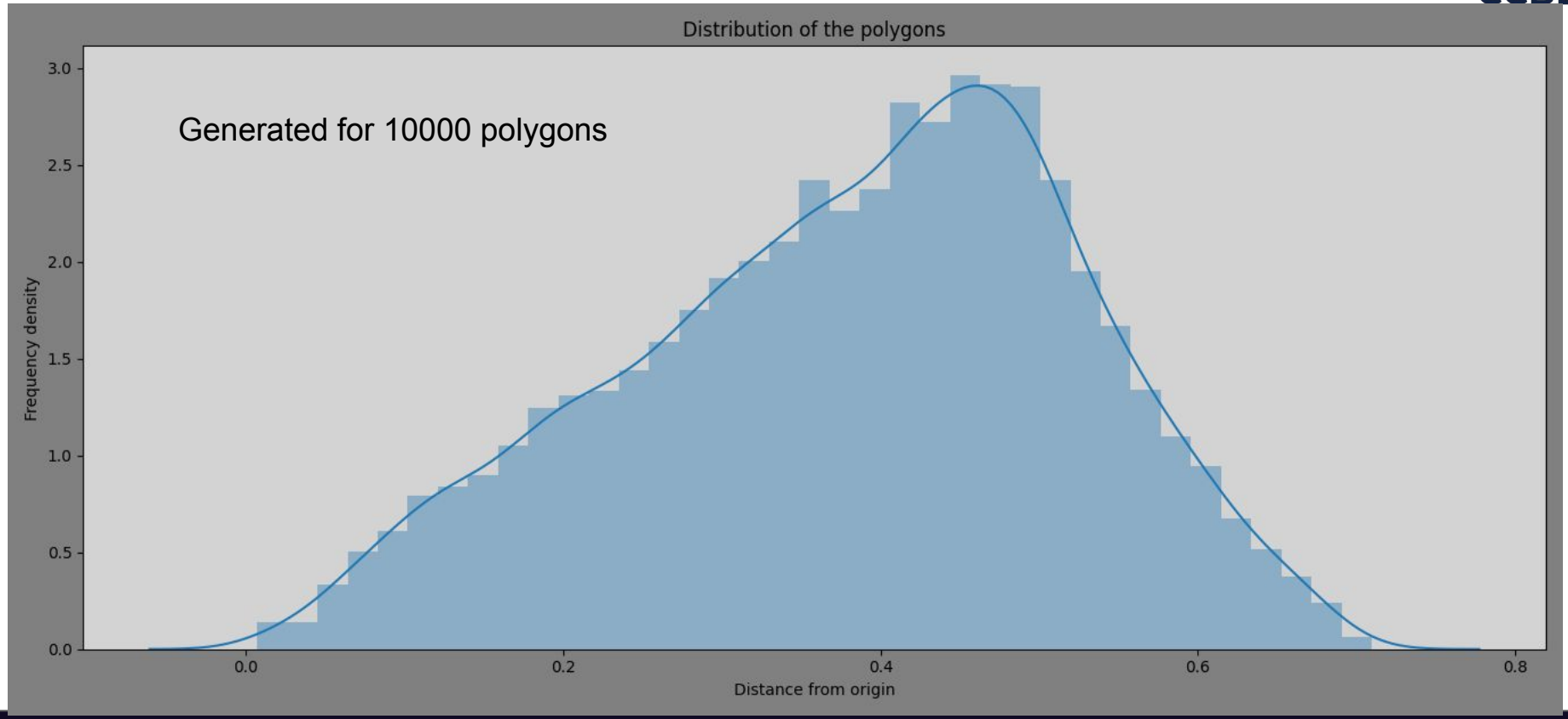
Polar Generator



Generated for 50 polygons

Observed to follow Skewed Gaussian distribution

PESU CCBD



Space Partition Algorithm

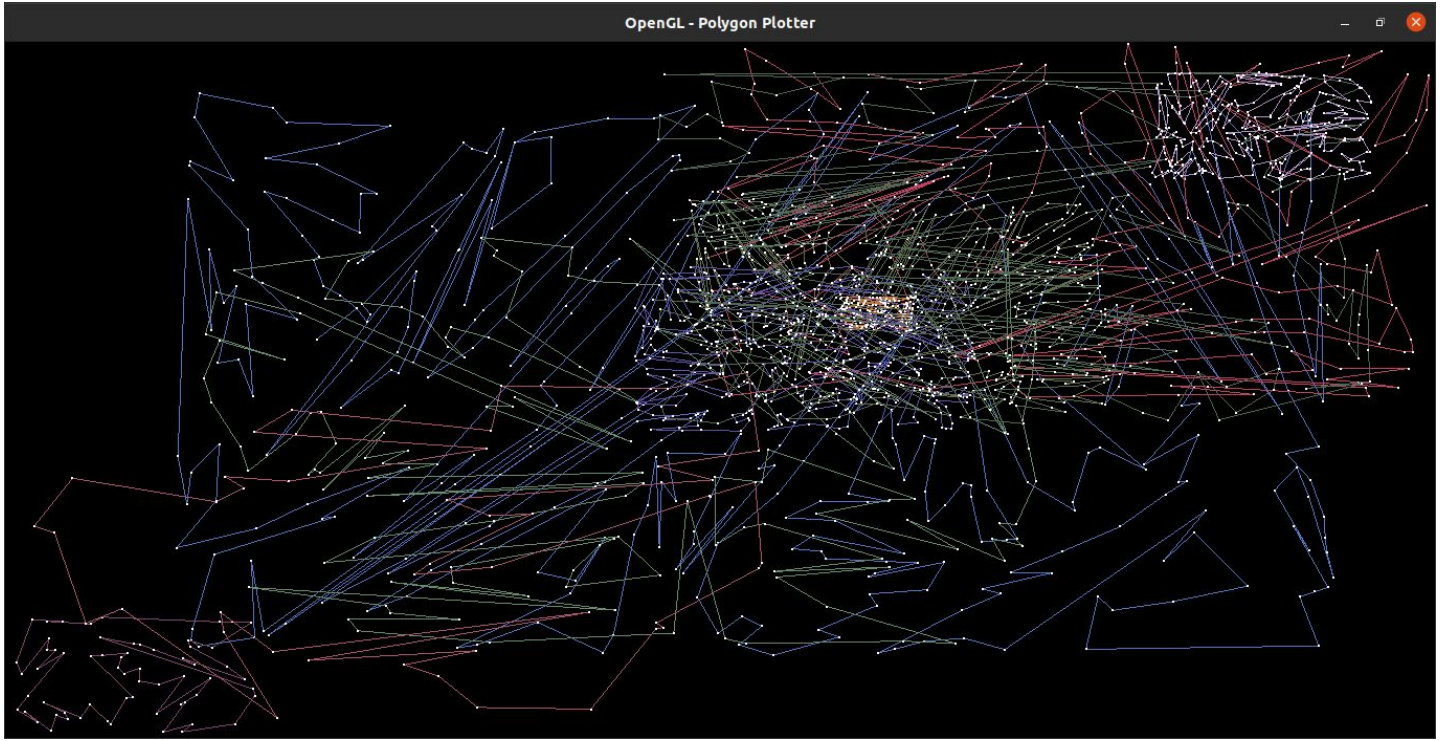
PESU CCBD

1. **S** is a set of randomly generated coordinates
2. Choose start point **s** and end point **e** from **S** at random
3. Split **S** into two subsets such that each subset consist of disjoint convex hulls, points are separated to left and right subset by the line **l(s,e)**
4. The termination condition for the recursion of a subset **S'** is if it has only two points, a start point **a** and end point **b** and the line segment **ab** is the output.
5. If there are more than two points in subset **S'**:
   a. Pick any point **s'** from **S'** at random
   b. Select a random line **l** through **s'** such that it intersects **ab** line segment
   c. Line **l** splits **S'** to two subsets **S''** (point **a** is the start point and **s'** as its last point) and **S'''** (point **s'** is the start point and **b** as its last point)
6. Go to step 4

Time Complexity :

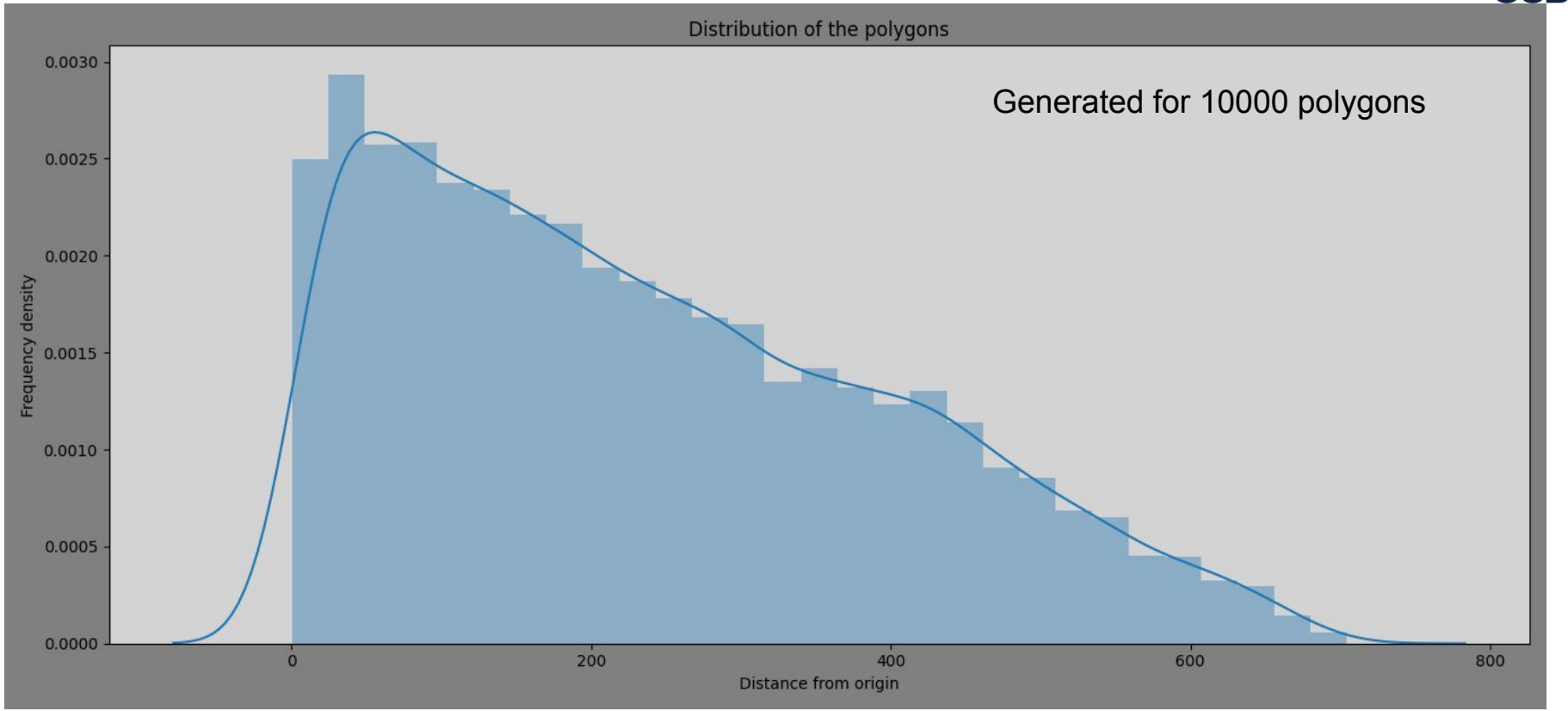
- Worst case - $O(n^2)$
- Best case - O(nlogn)

Space Partition Algorithm

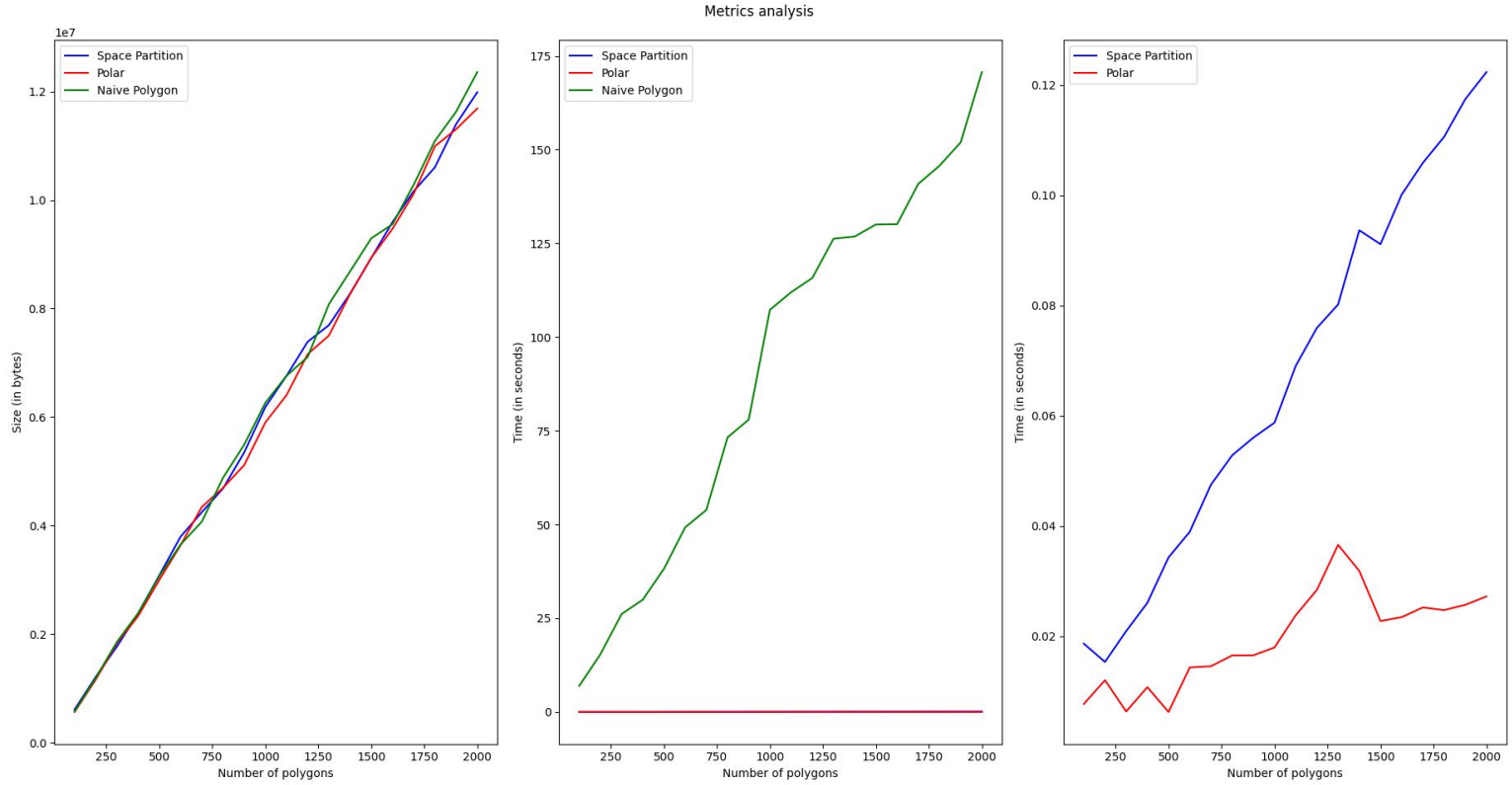


Generated for 10 polygons

Observed to follow Weibull Distribution



SIZE AND TIME ANALYSIS



Observation:

- Naive approach takes longest time
- Polar takes shortest time
- Approximately, the size of the WKT file increases linearly as the number of polygons stored increases

Additional Remarks

PESU CCBD

- Initially, we used matplotlib for plotting the polygons. However, it was very slow and used to consume a lot memory for plotting large number of polygons. Hence we looked into **OpenGL**. The plotting now is optimised and uses very less resources compared to matplotlib while being visually more appealing to the user. The polygons were also given random colours so that they can be distinguished better when plotted together.
- Multithreading using **OpenMP** gave us the ability to generate multiple polygons at the same time in parallel
- We found data visualisation (graphing) using C++ to be very challenging and tedious, hence we exploited the beauty and simplicity of python for data visualization (graphing)
- We found **GDB** very useful while debugging our code to identify various errors like segmentation faults, infinite loops, etc and correct the logic by stepping through each instruction.
- We wrote a **shell script** that could profile our program.
- The usage of makefiles made programing easier as we did not have to type lengthy commands for compilation. It also gave us the flexibility to have separate recipes for production and debugging. We also used automatic variables inside our makefile which allowed us to add new source files without needing to update the makefile

Project Structure

README.md

Polygon Generator

A Program to generate Random Polygons using three different algorithms, write them to a file in WKT format and visualise them using OpenGL.

Project Structure

The `src` folder contains all the source code for this project. It consists of the following files:

1. **Driver.cpp** which contains the main function
2. **Graphics.cpp** which contains the OpenGL graphing routines
3. **Polygon.hpp** which contains the Polygon class definition
4. **Polygon.cpp** which contains the definitions of the Polygon class member functions
5. The files **NaivePolygonGenerator.cpp, SpacePartition.cpp and Polar.cpp** contain the three random polygon generation algorithms
6. **WKT_writer.cpp** contains the routines to write the generated Polygons to a file in the `WKT` format

The `Profiler.sh` file contains a shell script that can be used to profile our program.

The files `Distribution.py` and `Metrics.py` are used for graphing the Distribution of the generated polygons and visualising the metrics generated after profiling respectively.

The `Images` folder contains some screenshots of the generated polygon maps.

Dependencies

- A C++ compiler like g++ or clang
- popt.h (for command line input)
- OpenGL (for visualising the generated polygons)

README.md

The `Profiler.sh` file contains a shell script that can be used to profile our program.

The files `Distribution.py` and `Metrics.py` are used for graphing the Distribution of the generated polygons and visualising the metrics generated after profiling respectively.

The `Images` folder contains some screenshots of the generated polygon maps.

Dependencies

- A C++ compiler like g++ or clang
- popt.h (for command line input)
- OpenGL (for visualising the generated polygons)

Compiling and running

Compile using the make utility:

```
$ make polygonGenerator -j$(($(nproc)+1))
```

Run the following command to print the help doc for the program:

```
$ ./bin/polygonGenerator -?
```

Run the Profiler using:

```
$ ./Profiler.sh 5
```

Here, the 5 represents the number of iterations the Profilers runs the program. In any iteration `i`, `100i` polygons are generated using each of the three algorithms.

LEARNING OUTCOME

PESU CCBD

We learnt many things while working on this project such as:

- Makefile
- C++
- OpenGL
- Multithreading (pthread and OpenMP)
- Debugging using GDB
- Shell scripting
- Many different ways to generate Convex Hulls and random polygons
- Different data distributions
- Collaborating on GitHub
- Team work
- Time management

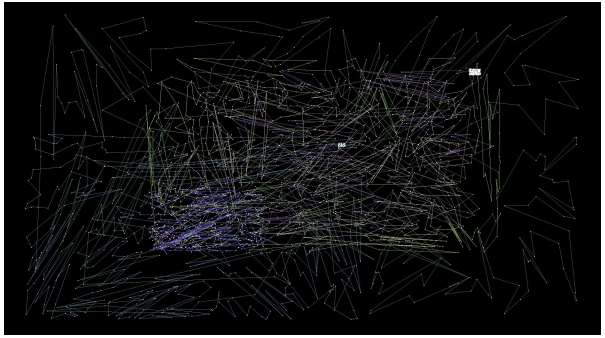
REFERENCES

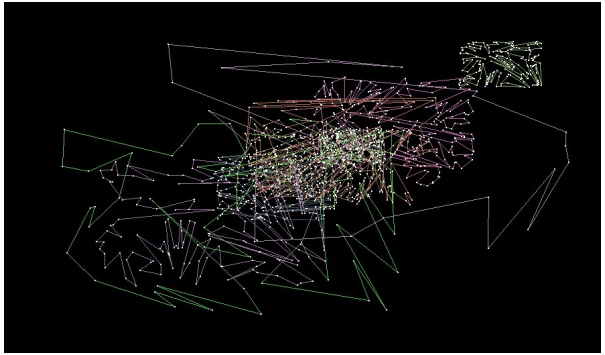
PESU CCBD

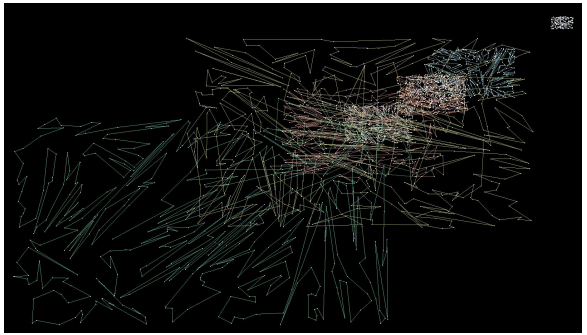
1. Heuristics for the Generation of Random Polygons
2. GenerationOfSimplePolygons.pdf
3. Algorithm to generate random 2D polygon
4. Well-known text representation of geometry
5. https://www.cplusplus.com/reference/random/
6. C/C++ Program for How to check if two given line segments intersect?
7. Orientation of 3 ordered points
8. Random Polygon Generator (RPG)
9. https://github.com/BichengLUO/random_polygon
10. <vector> - C++ Reference
11. terminate called after throwing an instance of 'std::out_of_range'
12. c++ type/value mismatch at argument 1 in template parameter list
13. Graham scan
14. INFINITY

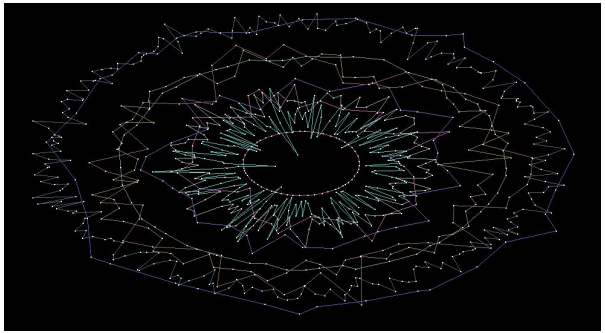
Pictures of few more maps

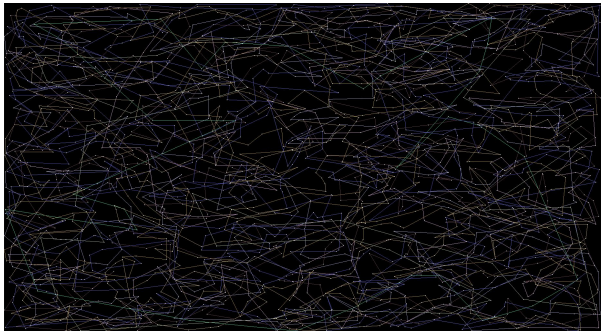
PESU CCBD

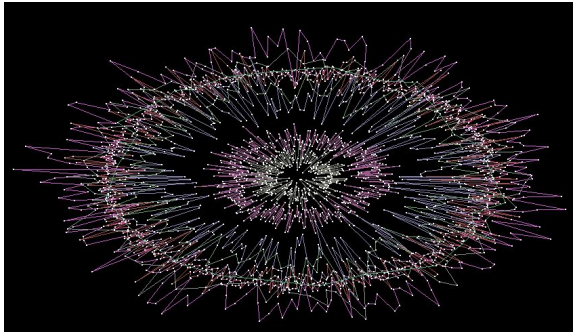












PES University

Cloud Computing and Big Data

THANK YOU!