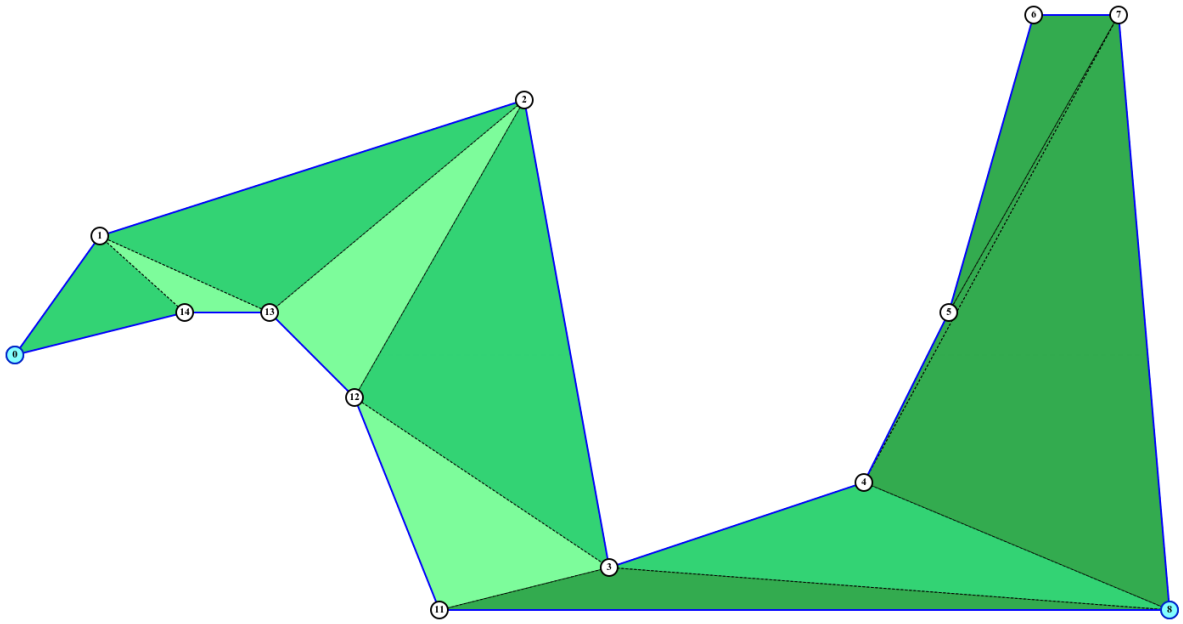
Implementace
Planární Polygonové Triangulace
v jazyce C++



Jonáš Sanislo

Praha 4 Nusle, 2024

Obsah

Úvod

Následující text popisuje implementaci planární polygonové triangulace (dále jen code), která dle daného vstupu provádí výpočet obsahu zadaného polygonu. Implementace obsahuje dvě hlavní třídy, Graph a ThreadGraph. Implementace těchto třídy, jak již název napovídá je rozdílná v právě v počtu využitých vláken. Pro usnadnění popisu si zavedeme slovník pojmů a představíme si použité třídy.

Co to to trianguace?

Planární triangulace je proces rozdělení (triangulace) planárního polygonu (uzavřené oblasti na rovině) na trojúhelníky tak, aby žádné dva trojúhelníky neměly společný bod (kromě vrcholů polygonu) a žádné dvě hrany trojúhelníků se nekřížily. Triangulace planárních polygonů má široké využití, například v počítačové grafice, algoritmice nebo v geografických informačních systémech. Triangulací můžeme rozumět jako rozdělení plochy polygonu na jednodušší a lépe zpracovatelné části.

Třídy

Graph

- Představuje celkovou datovou strukturu polygonu vlastnící veškeré atributy polygonu

ThreadGhraph

- Vícevláknová verze třídy Graph

Vertex

- Vytvořený objekt představuje bod v rovině, drží informace o svých souřadnicích a unikátním jméně (name)

Triangle

- Obsahuje 3 vrcholy polygonu, je produktem procesu triangulace

DataHelper

- Pomocná třída pro spouštění porovnávání codů a výpisu do Console

Slovník

- Pivot

Pivotní vrcholy jsou takové vertexy, které mají největší rozdíl x souřadnic. Jinak řečeno jsou to vrcholy, který se nachází na nejkrajnějších bodech polygonu. V případě konfliktu je jako pivot vždy vybrán ten s nižší y souřadnicí.

- Chain

V ThreadGraph se nachází dva chainy, higher_chain a lower_chain. Tyto dva chainy jsou posloupností vrcholů, ve směru hodinových ručiček napínajících se mezi pvotními body.

- Area

Obsah polygonu či trojúhelníku

Podmínky

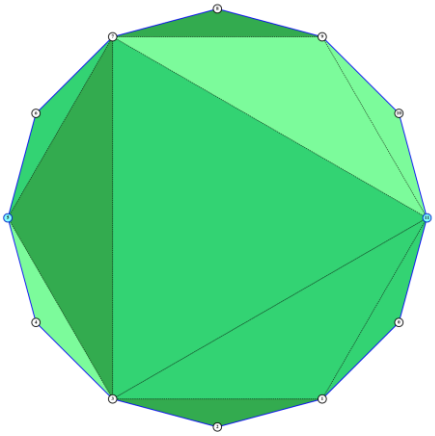
Code není schopen zpracovat ledajaký vstup, je tedy nutnost splnit pár podmínek.

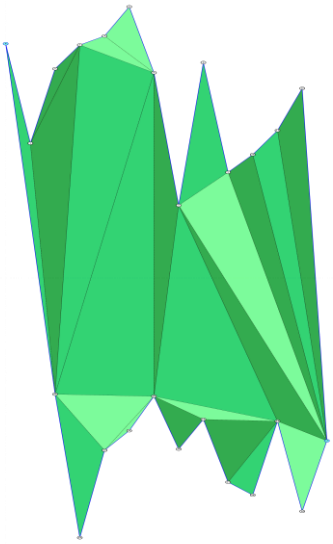
1. Pořadí ve kterém jsou souřadnice vrcholů zadány musí tvořit validní polygon.
2. Pořadí musí být zadáno po směru hodinových ručiček.

Vstupní data

Projekt obsahuje třídu DataHelper která disponuje dvěma funkcemi create_random_polygon() a create_complex_polygon()které generují vector<pair<double, double>> představující seznam souřadnic bodů polygonu.

COMPLEX **RANDOM**





COMPLEX

```
create_complex_polygon(int n, double r)
```

Complexní vytvoří pravidelný n-úhelník o daném poloměru a počtu vrcholů.

RANDOM

```
create_random_polygon(int n, int top, int threshold)
```

Random vygeneruje polygon o n vrcholech. Prvních n/2 vrcholů je Y souřadnice kladná v rozsahu threshold až top, tato hodnota je generována funkcí

```
get_random_double(int bottom, int top).
```

Pro zbývající polovinu platí to samé, pouze se zápornými hodnotami. X souřadnice je jednoduše hodnota iterace for loopu, který běží v intervalu (0; n/2 nebo (*n+1/2)*).

```
if (n % 2 == 0) { break_point = n / 2; } else { break_point = (n + 1) / 2; }

polygon.reserve(break_point);
for (int i = 0; i < break_point; ++i) {
    polygon.emplace_back(i, get_random_double(threshold, top));
}
for (int i = 0; i < n - break_point; ++i) {
    polygon.emplace_back(break_point - i, -get_random_double(threshold, top));
}
```

Algoritmus Triangulace

Podmínka velikosti:

- Funkce začíná kontrolou, zda graf obsahuje alespoň tři vrcholy. Pokud ne, vrací se z funkce, protože triangulace by neměla proběhnout na grafu s menším počtem vrcholů.

Kopie vrcholů:

- Vytvoří se kopie vektoru vertices do vecCopy, aby neovlivnila původní data.

Triangulace:

- Následně probíhá cyklus, dokud má vecCopy více než dva vrcholy.
- V rámci cyklu prochází všechny vrcholy v vecCopy.

Výběr vrcholů:

- Pro každý vrchol cur vybírá předchozí vrchol prev a následující vrchol next.

Podmínky konvexnosti a obsahu:

- Zkontroluje, zda jsou vrcholy prev, cur, a next konvexní a zda uvnitř této trojice nejsou žádné další vrcholy (kontrola pomocí funkcí is_convex a dont_contains).

Přidání trojúhelníku:

- Pokud jsou splněny podmínky, přidá trojúhelník do výsledné triangulace pomocí funkce add_triangle a odstraní vrchol cur z vecCopy.

Opakování:

- Cyklus se opakuje, dokud není vecCopy redukována na dva vrcholy.

```
void Graph::triangulate() {
  if (size()<3){ return;}

  vector<vertex> vecCopy = vertices;
  vertex cur, next, prev;
  while (vecCopy.size()>2) {
    for (int i = 0; i < vecCopy.size(); ++i) {
      cur = vecCopy[i];
      next = get_next(i, vecCopy);
      prev = get_prev(i, vecCopy);

 if (is_convex(prev, cur, next) && dont_contains(prev, cur, next,vecCopy))     {
          add_triangle(&prev, &cur, &next);
          vecCopy = removeElementAtIndex(vecCopy, i);
        }}}}
```

Proces výpočtu obsahu Polygonu

Ve třídách se nachází funkce Process(), která vykoná veškeré nezbytné úkony ve správném pořadí a zároveň provede měření.

```
double Graph::Process(const vector<pair<double, double>> &vxs) {
    add_vertex_list(vxs);
    separate_high_low();
    exclude_vertex();
    triangulate();
    calculate_area();
}
```

Přidání vrcholů:

- Na začátku se přidají vrcholy do grafu pomocí funkce add_vertex_list(vxs). Měří se doba trvání této operace.

Rozdělení na horní a dolní část:

- Následně se provede rozdělení na horní a dolní část pomocí funkce separate_high_low().

Kde se nalznou pivotní body Graph

- **Ve vícevláknové verzi doje dle těchto pivotních bodů k rozdělení na lower a higher chain**

A jejich alternaci.

Vyloučení vrcholů na přímce:

- V grafu se vyloučí vrcholy, které leží na přímce, pomocí funkce exclude_vertex().

Triangulace:

- Probíhá triangulace vrcholů v grafu pomocí funkce triangulate().
- **Ve vícevláknové verzi je tento úkon proveden asynchroně pro oba chain naráz.**

Výpočet plochy:

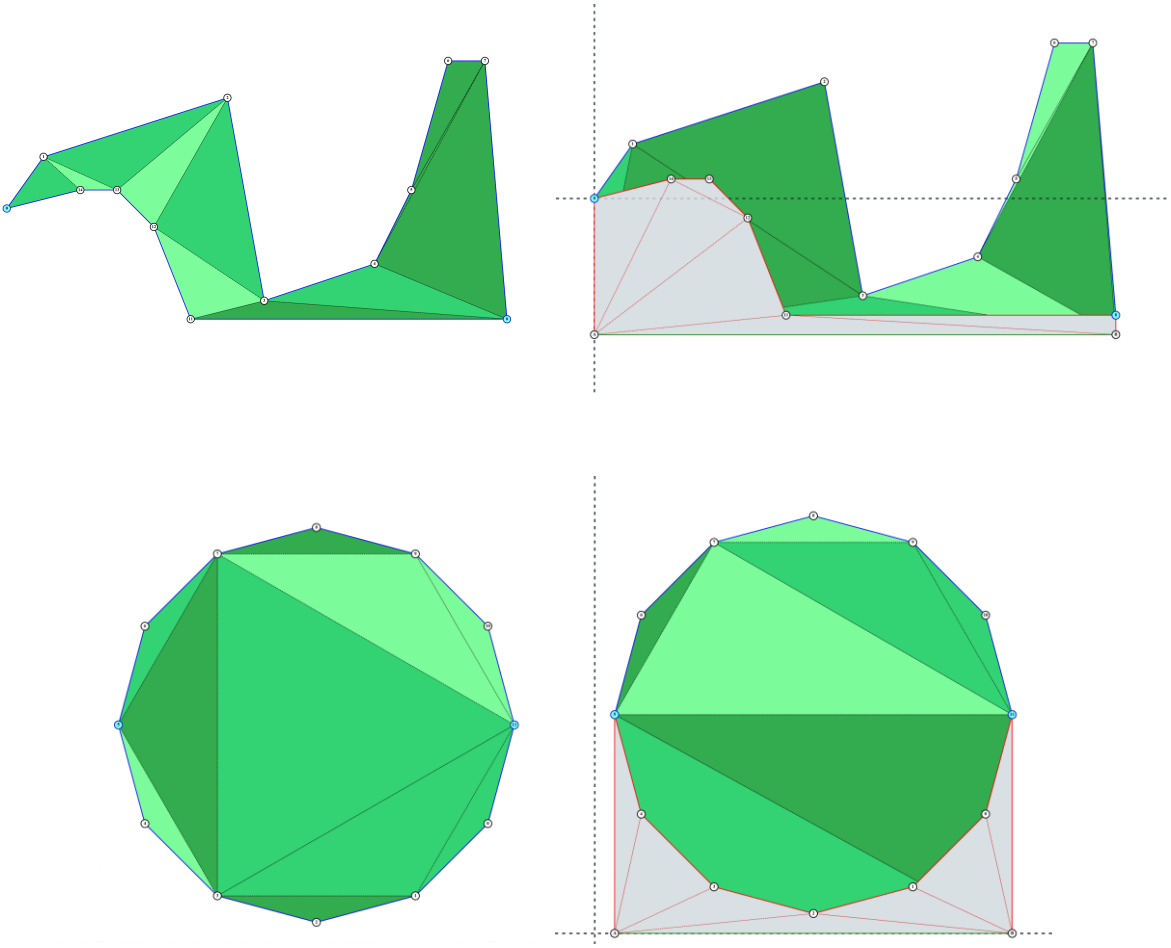
- Následně se vypočítá celková plocha triangulovaného polygonu pomocí funkce calculate_area(). Měří se doba trvání této operace.
- **Ve vícevláknové verzi je tento úkon proveden asynchroně pro oba chain naráz.**

Výstup a měření celkového času:

- Nakonec jsou vypsány informace o výsledku (plocha, počet trojúhelníků, počet vrcholů) a celkový čas zpracování všech operací včetně měření pomocí std::chrono.

Více vláknová implementace

Primárním rozdílem implementace mezi jedno a více vláknovou verzí je že ThreadGraph využívá pro triangulaci dvě vlákna pro každý chain jedeno. Tento způsob jsem zvolil proto že rozdělení polygonu na dva řetězce, konkrétně horní a dolní je nejspolehlivější způsob, jak lze tento proces rozdělit na vícero na sobě nezávislých úkonů.



V levé časti můžeme vidět vizualizaci výsledné triangulace polygonu jedno vláknovou implementací třídy Graph. Vidíme, že došlo k rozdělení polygonu na trojúhelníky uvnitř polygonu. Zatím co v případě ThreadGraph došlo k přesahu. To proto že po rozdělení polygonu na chainy a přidání dvou bodů funkcí `void alternate_chains();` která do higher a lower chain přidá takové dva vrcholy, které jsou X souřadnicí shodné s pivotními body a jejich Y složka je nižší, než nejnižšího vrcholu polygonu čímž vytvoří dva samostatné polygony jejichž absolutní hodnota rozdílů obsahu je rovno obsahu polygonu původního.

Jelikož se jedná o dva nezávislé polygony je možno proces jejich triangulace paralelizovat stejně jako jejich výpočet. Zdlouhavé popisování codu asi není nijak přínosné. Ve zkratce jsou vytvořeny dvě funkce pro triangulaci a výpočet obou chainů zvlášť, které jsou pak paralelně spuštěny ve funkci triangulate().

```
double ThreadGraph::triangulate_higher_chain_async() {
    triangulate_higher_chain();
    return calculate_area(higher_triangles);
}
```

Odhad náročnosti

Pro zhodnocení složitosti funkce Proces představující celkový průběh v třídě ThreadGraph, musíme vzít v úvahu jednotlivé kroky, které tato funkce provádí. Při analýze složitosti zohledníme několik klíčových částí kódu, které jsou v této funkci obsaženy:

add_vertex_list: Vložení vrcholů z poskytnutého seznamu. Složitost této operace bude O(n), kde n je počet vrcholů v seznamu.

exclude_vertex: Vyloučení některých vrcholů, což vyžaduje procházení všech vrcholů a mazání těch, které jsou na přímce mezi sousedními body. V nejhorším případě bude mazání O(n), kde n je počet vrcholů.

separate_high_low: Rozdělení vrcholů na horní a dolní řetězce. Tato operace také vyžaduje procházení všech vrcholů a několik dalších operací na vytvoření horního a dolního řetězce. Složitost bude O(n).

triangulate: Triangulace oblastí vytvořených horním a dolním řetězcem. Složitost triangulace při použití algoritmus jako "Ear Clipping" je kvadratická, složitost bude nejhůře O(n^2). Podmínková funkce dont_contains() je závislá na délce vektoru zbývajících vrcholů nicméně její složitost je O(n^2). Celkový odhad je tedy cca O(n^3)

calculate_area: Výpočet plochy pomocí součtu ploch jednotlivých trojúhelníků. Složitost této operace bude O(m), kde m je počet trojúhelníků.

Ve více vláknové verzi jsou poslední dvě funkce asynchroní tudíž předpokládáme poloviční náročnost **O(n^3 / 2)**, a však muže se stát že dojde k velkému nepoměru počtu vrcholů v chain a výsledná složitost bude **O((n+2)^3)**. Tato verze je velice nepravděpodobná, a muselo by se jednat o velice specifický polynom jehož pivoti budou hned za sebou.

Celková složitost funkce Proces bude součtem složitostí jednotlivých kroků. V nejhorším případě bude **O(n^3 + c),** kde n je počet vrcholů a c je zbytková zanedbatelná složitost cca 4n. Je však třeba zdůraznit, že konkrétní složitost úzce souvisí s podobou vstupních dat.

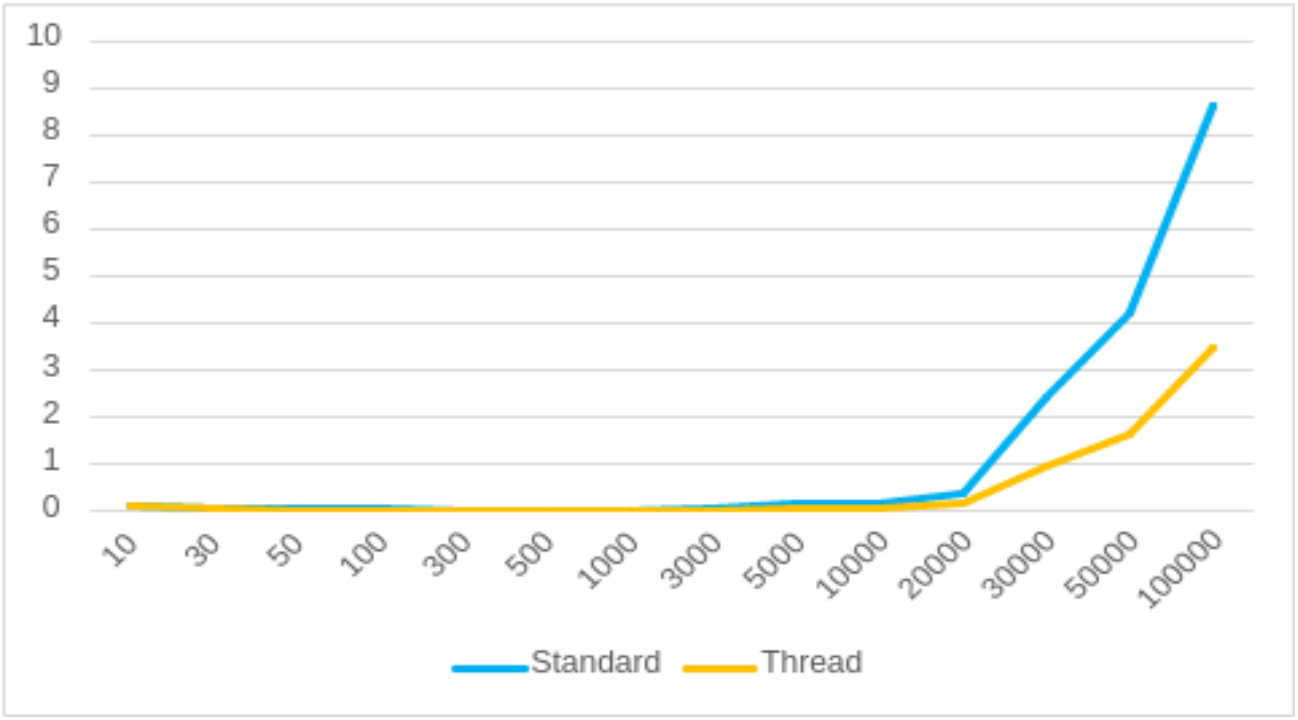
Výsledky měření

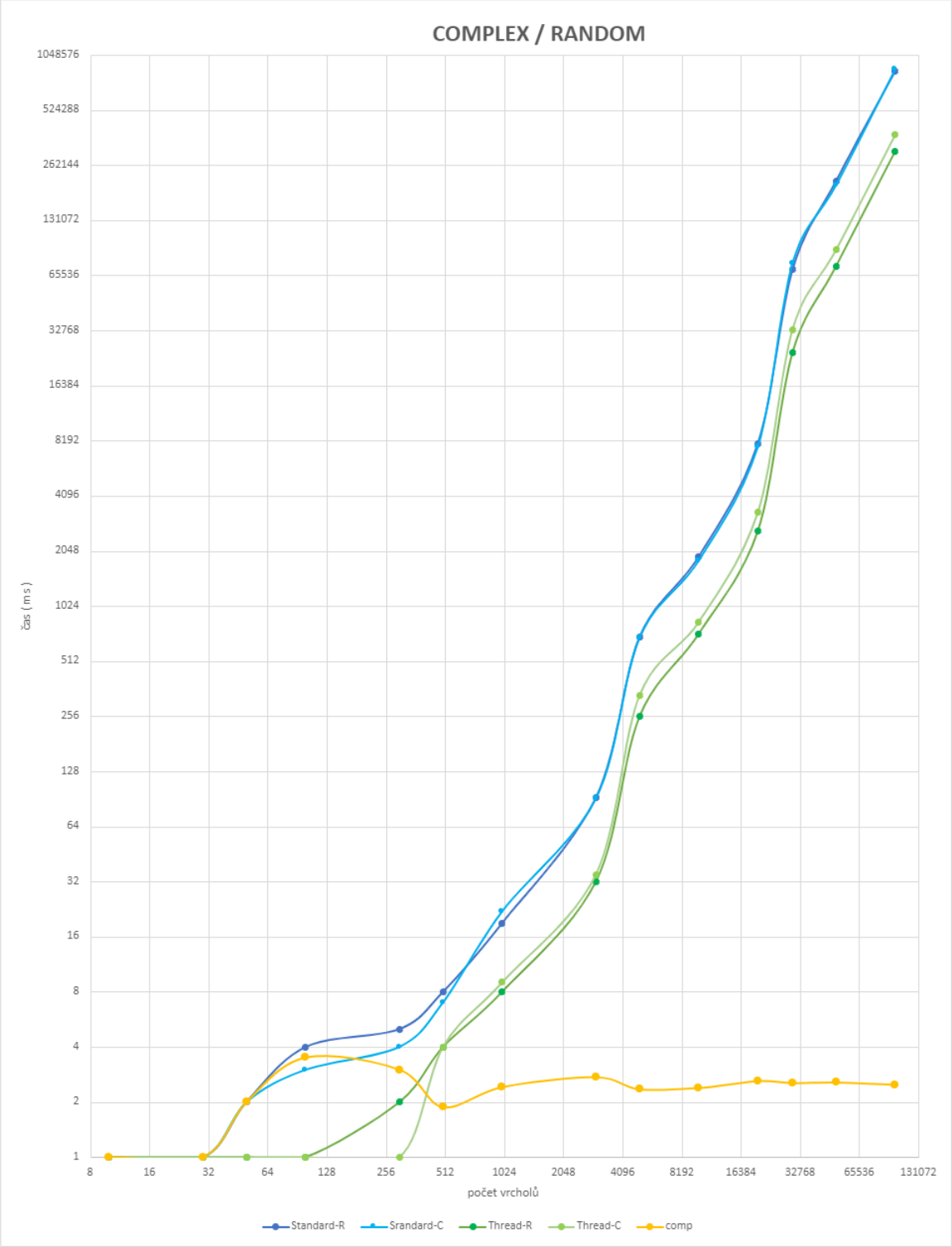
Měření proběhlo na 6 jádrovém procesoru i7-8850H CPU @ 2.60GHz na vygenerovaných polygonech v rozsahu od 3 do1 000 000 vrcholů. Pro každý počet vrcholů byl vytvořen jeden COMPLEX a jeden DANDOM polygon a následně triangularizován oběma implementacemi Graph a ThreadGraph. Celkový test v tomto rozsahu je velice časově náročný proto při spuštění programu proběhne verze končící 100 000 vrcholy. Rozsah je možný upravit v DataHelper.cpp proměnná `const vector<vector<int>> DataHelper::Data;`

Průběh je doprovázen intuitivní výpisem. Na konci je vidět celkové srovnání.

Celkový test jsem provedl 10x a výsledky zprůměroval

Počet vrcholů	Graph	ThreadGraph
10	1 ms	1 ms
30	1 ms	1 ms
50	2 ms	1 ms
100	3 ms	1 ms
300	4 ms	2 ms
500	7 ms	4 ms
1000	22 ms	8 ms
3000	91 ms	32 ms
5000	687 ms	255 ms
10000	1,810 s	721 ms
20000	7,612 s	2,638 s
30000	1 m 16,206 s	24,782 s
50000	3 m 23,435 s	1 m 13,346 s
100000	14 m 41,041 s	5 m 12,561 s


10
9
8
7
6
5
4
3
2
1
0
10
30
50
100
300
500
1000
3000
5000
10000
20000
30000
50000
100000
Standard
Thread



Závěr

Z grafů je viditelné že náročnost obou implementací má takřka kubickou složitost O(N^3), dle predikce je složitost rozdílných implementací v důsledku použití vícero vláken konstantní. Více vláknová verze **ThreadGraph** je v průměru (**comp**) 2.16x rychlejší než standardní jedno vláknová verze **Graph**.