РОЗДІЛ 3
ІНФОРМАЦІЙНІ СИСТЕМИ ТА ТЕХНОЛОГІЇ

УДК 004.047 : 371.38

ОПТИМІЗАЦІЯ ТА ВЗАЄМОВИКОРИСТАННЯ АЛГОРИТМІВ ВИЗНАЧЕННЯ НАКЛАДАННЯ ГРАФІЧНИХ ПРИМІТИВІВ

Панчук Олександр
здобувач третього (освітньо-наукового) рівня вищої освіти
другого року навчання
Приватного вищого навчального закладу
«Міжнародний економіко-гуманітарний університет
імені академіка Степана Дем'янчука» (м. Рівне, Україна)

Ковалевич Артем,
здобувач першого (бакалаврського) рівня вищої освіти
другого року навчання Приватного вищого навчального закладу
«Міжнародний економіко-гуманітарний університет
імені академіка Степана Дем'янчука» (м. Рівне, Україна)

Науковий керівник: **Шпортько О. В.**
кандидат технічних наук, доцент,
доцент кафедри інформаційних систем та обчислювальних методів
Приватного вищого навчального закладу
«Міжнародний економіко-гуманітарний університет
імені академіка Степана Дем'янчука» (м. Рівне, Україна)

Анотація. *У статті обгрунтована доцільність розробки та вдосконалення алгоритмів для визначення накладань графічних примітивів. Наведено декілька таких алгоритмів для визначення накладань відрізків на прямій та на площині з визначенням їх асимптотичної обчислювальної складності. Для зменшення обчислювальної складності визначення накладань графічних примітивів в окремих задачах запропоновано виконувати їх попереднє сортування та визначити взаємозв'язок показників попереднього і наступного примітиву. Алгоритми цього напрямку для меншої кількості вимірів запропоновано програмувати в окремих методах так, щоб мати можливість їх використати у вироджених випадках аналогічних алгоритмів для більшої кількості вимірів. Реалізації розроблених алгоритмів наведено мовою програмування C#. Правильність теоретичних викладок та коректність наведених фрагментів програм підтверджена експериментально за допомогою*

віддаленого незалежного обчислювального середовища на сайті https://basecamp.eolymp.com.

Ключові слова: *накладання графічних примітивів, асимптотична обчислювальна складність, оптимізація алгоритмів.*

Abstract. *The article substantiates the feasibility of developing and improving algorithms for determining overlaps of graphic primitives. Such several algorithms are presented for determining overlaps of segments on a straight line and on a plane with the determination of their asymptotic computational complexity. It is proposed to perform their preliminary sorting and determine the connection between the indicators of the previous and next primitive in order to reduce the computational complexity of determining overlaps of graphic primitives in individual problems,. It is proposed to program algorithms of this direction for a smaller number of dimensions in separate methods to be able to use them in degenerate cases of similar algorithms for a larger number of dimensions. Implementations of the developed algorithms are given in the C# programming language. The correctness of the theoretical calculations and the correctness of the given program fragments are confirmed experimentally using a remote independent computing environment on the website https://basecamp.eolymp.com.*

Keywords: *overlay of graphic primitives, asymptotic computational complexity, optimization of algorithms.*

Алгоритми для визначення накладання графічних примітивів [5] (відрізків, кіл, прямокутників, полігонів та ін.), з яких складаються складніші об'єкти, широко використовуються у векторній комп'ютерній графіці, зокрема в процесі програмування комп'ютерних ігор з ефектами анімації. Зрозуміло, що ці алгоритми мають працювати швидко, безпомилково та використовувати якомога менше апаратних ресурсів. Тому розробка нових і вдосконалення існуючих алгоритмів для визначення накладання графічних примітивів є актуальним науковим завданням на сьогодні і залишатиметься таким в найближчому майбутньому.

Покажемо, як можна використати алгоритми визначення накладання графічних примітивів одновимірного простору (на числовій прямій) в аналогічних алгоритмах для двовимірного простору на прикладі відрізків.

Для використання алгоритму визначення накладання відрізків на числовій прямій розв'яжемо задачу № 5259 «Відрізки» з сайту https://basecamp.eolymp.com: «На координатній прямій задано n відрізків $[a_i, b_i]$. Визначити кількість пар (i, j) таких, що $i < j$ і відрізки $[a_i, b_i]$ та $[a_j, b_j]$ мають хоча б одну спільну точку.

Як відомо, два відрізки на координатній прямій перетинаються, якщо початок першого з цих відрізків не перевищує кінця другого відрізка і початок другого відрізка не перевищує кінця першого відрізка (рис. 1).

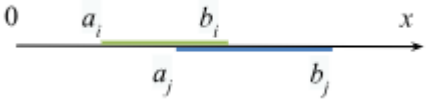

Рис. 1. Два відрізки на координатній прямій, що перетинаються

Тобто для перетину відрізків $[a_i, b_i]$ та $[a_j, b_j]$ має виконуватися умова:

$$a_i \le b_j \wedge a_j \le b_i, \quad (1)$$

а спільний відрізок між цими відрізками визначимо як

$$[\max(a_i; a_j); \min(b_i; b_j)]. \quad (2)$$

Цей спільний відрізок може вироджуватися в точку, якщо його ліва і права межі співпадають.

Отже, алгоритм очевидного розв'язку поставленої задачі полягає послідовному переборі для кожного чергового відрізка *i* всіх наступних відрізків *j* та збільшенні на одиницю кількості пар відрізків, що перетинаються, якщо виконується умова (1). Код відповідного і наступних фрагментів програм наведемо мовою програмування C#, оскільки на сьогодні вона є однією з основних мов програмування прикладних додатків:

```
//структура відрізка для збереження його початку і кінця
struct Line {public int a, b; }
…
Line[] lines = new Line[n]; //оголошення масиву відрізків
…
long count = 0;
for (i = 0; i < n - 1; i++)
 for (j = i + 1; j < n; j++)
  if (lines[i].a <= lines[j].b && lines[j].a <= lines[i].b)
   count++;
Console.WriteLine(count.ToString());
```

Результати тестування цієї (рис. 2) та наступних програм подамо з незалежної автоматичної системи тестування, доступної з сайту https://basecamp.eolymp.com. На цьому сайті до кожної задачі програміст може завантажити розв'язки у вигляді текстів різних програм, написаних навіть на різних мовах програмування. Після цього сервер сайту компілює отримані програми і подає їм на вхід виконання різні тести, невідомі

програмістам, та звіряє отримані результати на виході з очікуваними. Задача вважається розв'язаною лише тоді, коли відповідна програма пройшла всі тести за відведені проміжки часу та не перевищила допустимі обсяги використання оперативної пам'яті. Переваги від використання такої віддаленої автоматичної системи тестування очевидні:

– програміст може писати тексти програм для розв'язування однієї задачі на різних мовах програмування і порівнювати їх ефективність;

– розробник може розв'язувати задачу різними методами та порівнювати їх ефективність;

– програмісти можуть змагатися між собою, намагаючись написати найефективнішу програму як очно, так і дистанційно;

– розробники не можуть адаптувати програми під конкретні вхідні дані, оскільки тести їм невідомі. Вони починають усвідомлювати значення ефективності програмного коду та критеріїв його оцінювання.

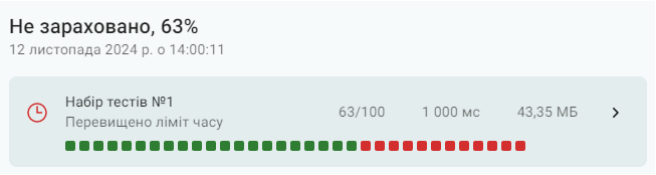

Рис. 2. Результати тестування першого варіанту програми до задачі № 5259

У другому стовпці результатів тестування розв'язку задачі на сайті https://basecamp.eolymp.com після номера набору тестів та основного результату (див. рис. 2) наводиться кількість зарахованих тестів та їх загальна кількість, у третьому – максимальний час виконання, у четвертому – максимальний обсяг використаної пам'яті.

Бачимо, що описаний вище алгоритм дав змогу вчасно виконати 63 % тестів. Решта тестів були не пройдені через перевищення ліміту часу. Для прискорення розв'язування задачі модифікуємо наведений алгоритм, впорядкувавши вхідні відрізки за зростанням їх початку. Відрізки з однаковими початками впорядкуємо між собою за зростанням їх кінців, хоча це не обов'язково. Тоді для кожного чергового відрізка *i* доцільно перебирати наступні відрізки *j* лише до тих пір, поки початок *j*-го відрізка не перевищує кінця *i*-го та так само збільшувати на одиницю кількості пар відрізків, що перетинаються, якщо виконується умова (1):

```
for (i = 0; i < n - 1; i++)
 for (j = i + 1; j < n; j++)
  {if (lines[i].a<=lines[j].b && lines[i].b>=lines[j].a)
```

```
    count++;
   if (lines[j].a > lines[i].b) break; }
```

Таке вдосконалення алгоритму дало змогу збільшити частку пройдених тестів до 69 % (рис. 3), але не ліквідувало перевищення ліміту часу для решти тестів набору.

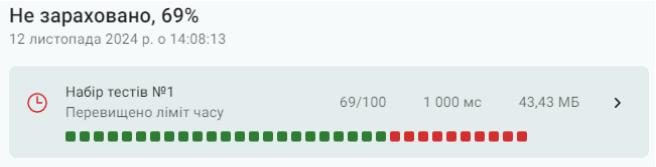

Рис. 3. Результати тестування другого варіанту програми до задачі № 5259

Чому ж реалізації наших алгоритмів перевищують ліміт часу? Справа в тому, що наведені алгоритми реалізуються вкладеними циклами і тому мають асимптотичну обчислювальну складність [1; 3; 4; 6] порядку $O(n^2)$, а для кардинального прискорення розв'язування задачі потрібно розробляти альтернативні алгоритми з нижчою обчислювальною складністю.

Для зменшення обчислювальної складності алгоритму розв'язування поставленої задачі визначимо кількості пар відрізків, що перетинаються, за кількістю перетинів *початків* кожного відрізка з відрізками, які розпочалися раніше і ще не закінчилися (назвемо такі відрізки *наявними*). Для цього впорядкуємо початки відрізків a_i разом з кінцями b_i за зростанням, а серед однакових координат розмістимо спочатку початки, а потім – кінці відрізків. Таке впорядкування під час послідовного перебору дає змогу правильно врахувати кількості перетинів відрізків, що мають одну спільну точку (відрізків, які в черговій точці розпочинаються, з іншими відрізками, які в цій же точці закінчуються). Початки кожного відрізка збільшують, а кінці – зменшують кількість наявних відрізків на одиницю, тому під час перебору країв відрізків кількість наявних відрізків доцільно модифікувати, а не підраховувати щоразу спочатку. Фрагмент програми для підрахунку кількості перетинів відрізків з використання поточної кількості наявних відрізків може виглядати так:

```
struct Point // структура для початків та кінців відрізків
 {public int x;
  public int type; } // 0-початок, 1-кінець відрізка

static int ComparePoint(Point A, Point B)
```

```
 {if (A.x < B.x || (A.x == B.x && A.type < B.type))
  return -1;
 if (A.x == B.x && A.type == B.type)
  return 0;
 return 1; }
…
int n = int.Parse(Console.ReadLine()); // кількість точок
Point[] points = new Point[2 * n];
for (i = 0, j = 0; i < n; i++)
 {// зберігаємо початки і кінці відрізків
  string[] input = Console.ReadLine().Split(' ');
  points[j].x = int.Parse(input[0]);
  points[j++].type = 0;
  points[j].x = int.Parse(input[1]);
  points[j++].type = 1; }
Array.Sort(points, ComparePoint);
long count = 0; // кількість перетинів відрізків
long availableLine = 0; // к-ть наявних ліній в черговій точці
for (i = 0; i < 2 * n; i++)
 if (points[i].type == 0) // зустріли початок лінії
  {count += availableLine;
   availableLine++; } // врахували лінії в наявних
 else availableLine--; // врахували кінець лінії
Console.WriteLine(count.ToString());
```

Наведена реалізація алгоритму дала змогу пройти всі тести і тому повністю розв'язати задачу (рис. 4) за рахунок зменшення часу виконання в середньому у понад 6 разів. Прискорення розв'язування задачі відбулося через зменшення асимптотичної обчислювальну складності підрахунку кількості перетинів відрізків з $O(n^2)$ до $O(n)$. Хоча в останньому алгоритмів додатковим найтривалішим процесом є сортування масиву початків і кінців відрізків, асимптотична обчислювальна складність якого становить з $O(n \log_2 n)$.

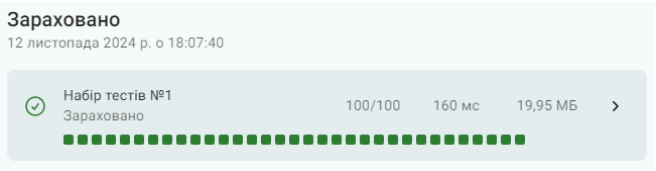

Рис. 4. Результати тестування третього варіанту програми до задачі № 5259

Використаємо тепер підходи визначення накладань відрізків на координатній прямій для часткових випадків визначення накладань відрізків на площині. Для цього розв'яжемо, наприклад, задачу № 1615 «Перетин відрізків» з цього ж сайту https://basecamp.eolymp.com: «Задано два відрізки: *AB* та *CD*. Визначте, яка множина точок є перетином цих відрізків. Якщо вказані відрізки не перетинаються, то виведіть рядок «Empty». Якщо відрізки перетинаються у одній точці, то виведіть два числа – координати точки перетину. Якщо перетином є відрізок, то виведіть чотири числа – координати двох кінців відрізка у лексикографічному порядку … ».

Бачимо, що розв'язком цієї задачі може бути точка, відрізок чи порожня множина. Для розв'язування цієї задачі спочатку впорядкуємо в лексикографічному порядку координати кінців відрізків *AB* та *CD* та визначимо коефіцієнти прямих, на яких лежать ці відрізки. Для цього скористаємося загальновідомим рівнянням прямої, що проходить через дві точки (x_1, y_1) та (x_2, y_2) [2]:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y(x) - y_1}{y_2 - y_1}, \quad (3)$$

звідки слідує, що

$$y(x) = \frac{y_2 - y_1}{x_2 - x_1} x - \frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1. \quad (4)$$

Рівняння прямої (4), що проходить через точки *A* та *B* запишемо скорочено з кутовим коефіцієнтом у вигляді $y_1(x)=k_1x+b_1$, а прямої, яка проходить через точки *C* та *D* – у вигляді $y_2(x)=k_2x+b_2$ (рис. 5), де

$$k_1 = \frac{y_b - y_a}{x_b - x_a}, \; b_1 = y_a - k_1 x_a, \quad k_2 = \frac{y_d - y_c}{x_d - x_c}, \; b_2 = y_c - k_2 x_c. \quad (5)$$

Тут і надалі індекс 1 вказує на відношення коефіцієнта чи функції до відрізка *AB*, а індекс 2 – до відрізка *CD*, а загальне рівняння прямої з кутовим коефіцієнтом запишеться так:

$$y_i(x) = k_i x + b_i, \; i=1, 2. \quad (6)$$

Якщо довільна точка $K(x_k, y_k)$ лежать на прямій (4), то $y_k = y(x_k)$, тобто $y_k - y(x_k) = 0$. Коли ця точка лежить вище прямої, то $y_k - y(x_k) > 0$. Якщо ж ця точка лежить нижче прямої (4), то $y_k - y(x_k) < 0$. У випадку,

коли два відрізки лежать на різних прямих і не перетинаються, краї хоча б одного з цих відрізків обов'язково лежать з одного боку від прямої іншого відрізка, тому добуток різниць ординат країв одного відрізка зі значеннями функції прямої іншого відрізка від абсцис цих країв більший від нуля (це, зокрема, стосується і випадку, коли відрізки лежать на паралельних прямих ($k_1 = k_2 \wedge b_1 \neq b_2$)). Наприклад, на рис. 5 відрізок *AB* лежить з одного боку від прямої відрізка *CD* і тому $(y_A - y_2(x_A)) \times (y_B - y_2(x_B)) > 0$. Якщо відрізки лежать на одній прямій або перетинаються між собою, то такий добуток не більший нуля.

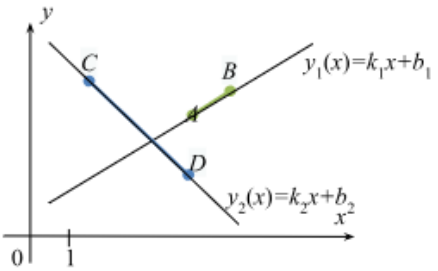

Рис. 5. Відрізки, що не перетинаються

Тому достатньою умовою того, що відрізки, які лежать на різних прямих, не перетинаються, є додатність добутку різниць ординат країв одного відрізка зі значеннями функцій прямих другого відрізка від абсцис цих країв. В програмній реалізації алгоритму розв'язку цієї задачі для уникнення переповнень ми обчислювали навіть не різниці, а знаки цих різниць відносно обраної прямої

$$\text{sign}_i(x_k; y_k) = \begin{cases} 1, & \text{якщо } y_k - y_i(x_k) > 0; \\ 0, & \text{якщо } y_k - y_i(x_k) = 0; \\ -1, & \text{якщо } y_k - y_i(x_k) < 0, \end{cases} \quad (7)$$

якщо функція $y_i(x)$ визначена. Використовуючи цю функцію, фрагмент програми для встановлення відсутності перетину відрізків, що лежать на різних прямих, може виглядати так:

```
if (sign1(xc, yc) * sign1(xd, yd) > 0 ||
    sign2(xa, ya) * sign2(xb, yb) > 0)
 {Console.WriteLine("Empty");
  return; }
```

Інакше, якщо добуток знаків цих різниць недодатний і $k_1 \neq k_2$, то координати точки $S(x_s; y_s)$ перетину відрізків *AB* і *CD* визначимо з умови перетину прямих, на яких лежать ці відрізки:

$$y_1(x_s) = y_2(x_s) \Rightarrow k_1 x_s + b_1 = k_2 x_s + b_2 \Rightarrow x_s = \frac{b_2 - b_1}{k_1 - k_2}; y_s = y_1(x_s)$$

.

Коли ж $k_1 = k_2 \wedge b_1 = b_2$, то відрізки *AB* і *CD* лежать на одній прямій і їх перетином може бути не лише точка чи порожня множина, а й відрізок. Саме в цьому випадку використаємо раніше описаний алгоритм визначення накладань відрізків на координатній прямій для абсцис відрізків *AB* і *CD*, який описується виразом (2):

```
xs = Math.Max(xa, xc); // початок спільної частини відрізків
xf = Math.Min(xb, xd); // кінець спільної частини відрізків
if (xf < xs)
 {Console.WriteLine("Empty"); return; }
ys = y1(xs); yf = y1(xf);
Console.WriteLine(formatWrite(xs)+" "+ formatWrite(ys));
if (!equal(xs, xf) || !equal(ys, yf)) // виводимо кінець,
 // якщо початок спільної частини не співпадає з кінцем
 Console.WriteLine(formatWrite(xf) + " " + formatWrite(yf));
```

Нажаль, реалізація наведеного алгоритму дає змогу правильно виконати лише 62 % тестів (рис. 6). Для решти тестів цей варіант програми видає неправильні відповіді.

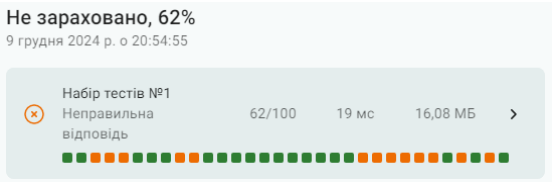

Рис. 6. Результати тестування першого варіанту програми до задачі № 1615

Справа тут в тому, що записуючи рівняння прямих з кутовими коефіцієнтами, які проходять через краї відрізків *AB* та *CD* у вигляді (4), ми не врахували області визначення коефіцієнтів k_1, b_1, k_2 та b_2 (5). А вони визначені, лише коли абсциси точок, через які проходять ці прямі, різні ($x_a \neq x_b$ та $x_c \neq x_d$). Рівняння ж прямих, що проходять через дві точки у вигляді (3), не можна використовувати не лише при однакових абсцисах, країв, а й при однакових їх ординатах. Тому у випадку, коли абсциси країв відрізка однакові, запишемо рівняння прямої у вигляді

$$x_i = constx_i,\ i=1,\ 2. \qquad (8)$$

де $constx_i$ – ця спільна абсциса. Для ідентифікації такого випадку, коли функція $y_i(x)$ не визначена, у відповідному кутовому коефіцієнті k_i збережемо максимально допустиме додатне значення його типу даних. Фрагмент коду для визначення коефіцієнтів, наприклад, першої прямої подамо так:

```
if (xa == xb) // випадок (8)
 {k1 = double.MaxValue;
  b1 = 0; // для однозначності порівнянь
  constx1 = xa; }
else // випадок (6)
 {k1 = (double)(yb - ya) / (xb - xa); // обчислюємо за (5)
  b1 = ya - xa * k1;
  constx1 = double.MaxValue; } // для однозначності порівнянь
```

У випадку рівності абсцис двох точок, через які проходить пряма, знак довільної точки відносно цієї прямої вигляду (8) обчислимо так:

$$\mathrm{sign}_i(x_k; y_k) = \begin{cases} 1, & \text{якщо } x_k - constx_i > 0; \\ 0, & \text{якщо } x_k - constx_i = 0; \\ -1, & \text{якщо } x_k - constx_i < 0. \end{cases} \qquad (9)$$

Тому, наприклад, функція для обчислення знаку точки відносно першої прямої з врахуванням (7) та (9) може бути реалізована так:

```
static double sign1(double xk, double yk)
 {if (k1 != double.MaxValue) // рівняння вигляду (6)
   {if (y > k1 * xk + b1) return 1; // обчислюємо за (7)
    else if (y < k1 * xk + b1) return -1;
        else return 0; }
  else // рівняння вигляду (8)
   {if (xk > constx1) return 1; // обчислюємо за (9)
    else if (xk < constx1) return -1;
        else return 0; }}
```

Використання двох варіантів запису рівняння прямої ((6) або (8)) для кожного з двох відрізків збільшує до двох кількість варіантів визначення перетинів, якщо ці прямі співпадають. При цьому порівняння коефіцієнтів двох прямих слід здійснювати з врахуванням можливої похибки ділень, яку ми поклали рівною 10^{-12}:

```
static bool equal(double u, double v)
 {if (Math.Abs(u - v) < 0.000000000001) return true;
  else return false; } …
// після перевірки перетинів по знаках країв кожного відрізка
```

```
// щодо прямої іншого відрізка аналізуємо накладання прямих
if (equal(k1, k2) && equal(b1, b2) && equal(constx1,
   constx2)) // всі чотири точки лежать на одній прямій
  {if (k1==double.MaxValue) // накладання прямих вигляду (8)
    {// накладання для вертикальних відрізків визначаємо за (2)
     ys = Math.Max(ya, yc); // початок спільної частини по у
     yf = Math.Min(yb, yd); // кінець спільної частини по у
     if (yf < ys)
      {Console.WriteLine("Empty");
       return; }
     xs = xf = constx1; }
   else … // накладання прямих вигляду (6), як описано вище
```

Крім цього, застосування двох варіантів запису рівняння прямої для кожного з двох відрізків збільшує до трьох кількість варіантів визначення перетинів, якщо ці прямі не співпадають (четвертий варіант паралельних вертикальних прямих обробляється раніше перевіркою перетину по знаках країв кожного відрізка відносно прямої іншого відрізка):

```
if (k1 != double.MaxValue && k2 != double.MaxValue)
 {// перетин двох невертикальних прямих за (5)
  xs = (b2 - b1) / (k1 - k2);
  ys = y1(xs); }
else
 if (k1 != double.MaxValue && k2 == double.MaxValue)
  {// перетин невертикальної з вертикальною прямою
   xs = constx2; ys = y1(xs); }
 else
  if (k1 == double.MaxValue && k2 != double.MaxValue)
   {// перетин вертикальної з невертикальною прямою
    xs = constx1; ys = y2(xs); }
Console.WriteLine(formatWrite(xs) + " " + formatWrite(ys));
```

Реалізація всіх зазначених вище варіантів перетинів відрізків з визначенням їх прямих в одному з двох варіантів ((6) або (8)) дала змогу пройти всі тести задачі (рис. 7).

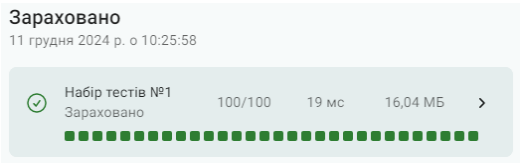

Рис. 7. Результати тестування другого варіанту програми до задачі № 1615

З наведених результатів дослідження приходимо до таких *висновків*:

1. Кардинальне зменшення обчислювальної складності алгоритму можливо саме за рахунок зменшення асимптотичної обчислювальної складності, тому під час розробки алгоритмів слід визначати і по можливості зменшувати саме асимптотичну обчислювальну складність.

2. Поняття асимптотичної обчислювальної складності в контексті часу виконання [1] варто вивчати ще з середньої школи під час опановування вкладених циклів. Це формуватиме навики написання не лише правильних, а й ефективних (з погляду часу виконання та використання обчислювальних ресурсів) програм.

3. Для зменшення обчислювальної складності визначення накладань графічних примітивів доцільно спочатку виконати їх сортування, яке має асимптотичну обчислювальну складність $O(n \log_2 n)$ та визначити взаємозв'язок показників попереднього і наступного примітиву.

4. Алгоритми визначення накладань графічних примітивів для меншої кількості вимірів доцільно програмувати в окремих методах так, щоб мати можливість їх використати у вироджених випадках алгоритмів накладань цих примітивів для більшої кількості вимірів.

5. Для кожної математичної функції, що використовується в програмі, слід враховувати її область визначення і для значень, які не входять в її область визначення, аналізувати доцільність альтернативного подання цієї функції.

ЛІТЕРАТУРА

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Introduction to Algorithms, Third Edition. Cambridge: MIT Press, 2009. 1320 p. **2.** Клепко В. Ю., Голець В. Л. Вища математика в прикладах і задачах: Навчальний посібник. Київ: Центр учбової літератури, 2009. С. 89. **3.** Обчислювальна складність. URL: https://znaimo.com.ua/Обчислювальна_складність#link0 (дата звернення: 28.11.2024). **4.** Оцінка складності алгоритмів, або Що таке O(log n). URL: https://echo.lviv.ua/dev/53 (дата звернення: 28.11.2024). **5.** Романюк О. Н. Комп'ютерна графіка. Навчальний посібник. Вінниця: УНІВЕСУМ-Вінниця. 2001. 129 с. URL: https: //web.posibnyky.vntu.edu.ua/fitki/8romanyuk_komp_grafika (дата звернення: 28.11.2024). **6.** Шинкаренко В. І. Особливості практичного застосування показників обчислювальної складності алгоритмів. *Проблеми програмування*. 2008. № 2-3. С. 57-63.