ლევან შოშიაშვილი

მოდელირება და ვიზუალიზაცია

თბილისი
2016,2025

სარჩევი

თავი 1

C++ ელემენტები

1.1 მონაცემების გადაცემა მისამართით

გავიხსენოთ Cკოდი სადაც ხდებოდა მონაცემების გადაცემა ფუნქციაში მიმთითებლის საშუალებით(იხ. არქივი *sqequation*)

კოდი 1.1. ფუნქციის დეკლარირება

```
#ifndef FUNCTIONS___H
#define FUNCTIONS___H
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
    #ifdef __cplusplus
    extern "C" {
    #endif
    int solve_sqeq(const double* _a, const double* _b,
    const double* _c, double* _x1, double* _x2);
    #ifdef __cplusplus
    };
    #endif
#endif // MYHEADER___H
```

კოდი 1.2. ფუნქციის იმპლემენტაცია

```
#include <stdio.h>
#include <math.h>
#include "functions.h"
#define EPS 0.000001
int solve_sqeq(const double*_a, const double* _b,
const double* _c, double* _x1, double* _x2)
{
  double d=(*_b)*(*_b)-4*(*_a)*(*_c);
```

```
9
10  if(d<0){
11  #ifdef _DEBUG
12     printf("D<0 araa amoxsna\n");
13  #endif
14    return 0;
15  }
16  else if(d<EPS)
17  {
18    #ifdef _DEBUG
19    printf("D=0 aqvs 1 amonaxseni\n");
20    #endif
21    (*_x1)=-(*_b)/(2*(*_a));
22   _x2=_x1;
23   #ifdef _DEBUG
24   printf("x1=x2=%g\n",(*_x1));
25   #endif // _DEBUG
26   return 1;
27  }
28  #ifdef _DEBUG
29  printf("aris 2 fesvi:\n");
30  #endif // _DEBUG
31  (*_x1)=(-(*_b)-sqrt(d))/(2*(*_a));
32  (*_x2)=(-(*_b)+sqrt(d))/(2*(*_a));
33  #ifdef _DEBUG
34  printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
35  #endif
36  return 2;
37 }
```

კოდი 1.3. quadratic.c პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას

```
#include "functions.h"
#define EPS 0.0000001

int main(int argc, char* argv[])
{
   double a=1;
 double b=2;
 double c=4;
 double x1=0;
 double x2=0;
 int result=-1;
 puts("shemoitanet kv. gant. koeficientebi: a, b, c");
 scanf("%lf,%lf,%lf",&a,&b,&c);
 result=solve_sqeq(&a,&b,&c,&x1,&x2);
 switch(result)
 {
  case 0:
  printf("D<0 ar aqvs amoxsna\n");
  break;
  case 1:
  printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
  printf("x1=%g:\n x2=%g\n", x1, x2);
  break;
  case 2:
  printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
  printf("x1=%g:\n x2=%g\n", x1, x2);
  break;
  default:
```

```
29   break;
30  }
31  return 0;
32 }
```

კოდში 1.2 ჩანს როგორ უნდა ვიმუშაოთ მიმთითებელთან. ფუნქციის გამოძახებისას კი ფუნქციას უნდა გადავცეთ ცვლადის მისამართი(მიმთითებლის მნიშვნელობა არის მისამართი.). C-ში მონაცემების გადაცემა ხდება მნიშვნელობით. ეს შეიძლება იყოს რიცხვი –ცვლადის მნიშვნელობა ან მისამართი ასევე ცვლადის(მისამართის შემთხვევაში მიმთითებლის მნიშვნელობა.

C++-ში ცხადია გვაქვს, როგორც მონაცემების გადაცემა მნიშვნელობით, ასევე მიმთითებლით. . ამას გარდა C++-ში გვაქვს მონაცემების გადაცემა მისამართით(Reference). ამისათვის გამოიყენება ნიშანი ამპერსანდი „&“.

განვიხილოთ კოდები სადაც გამოყენებულია მისამართით გადაცემა (კოდების არქივი *sqequationcpp*)

კოდი 1.4. ფუნქციის დეკლარირება *functionscpp.h*

```
1 #ifndef FUNCTIONSCPP___H
2 #define FUNCTIONSCPP___H
3 #include <math.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int solveSQeq(const double& _a, const double& _b,
8    const double& _c, double& _x1, double& _x2);
9
10 #endif // MYHEADER___H
```

კოდი 1.5. ფუნქციის იმპლემენტაცია *functionscpp.cpp*

```
1 #include <stdio.h>
2 #include <math.h>
3 #include "functionscpp.h"
```

```
4  #define EPS 0.000001
5  int solveSQeq(const double& _a, const double& _b,
6  const double& _c, double& _x1, double& _x2)
7  {
8    double d=_b*_b-4*_a*_c;
9
10   if(d<0){
11   #ifdef _DEBUG
12      printf("D<0 araa amoxsna\n");
13   #endif
14     return 0;
15   }
16   else if(d<EPS)
17   {
18     #ifdef _DEBUG
19     printf("D=0 aqvs 1 amonaxseni\n");
20     #endif
21     _x1=-_b/(2*_a);
22    _x2=_x1;
23    #ifdef _DEBUG
24    printf("x1=x2=%g\n",(*_x1));
25    #endif // _DEBUG
26    return 1;
27   }
28   #ifdef _DEBUG
29   printf("aris 2 fesvi:\n");
30   #endif // _DEBUG
31   _x1=(-_b-sqrt(d))/(2*_a);
32   _x2=(-_b+sqrt(d))/(2*_a);
```

```
33  #ifdef _DEBUG
34  printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
35  #endif
36  return 2;
37 }
```

კოდი 1.6. პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას *quadraticcpp.cpp*

```
1  #include "functionscpp.h"
2  #define EPS 0.0000001
3  int main(int argc, char* argv[])
4  {
5   double a=1;
6   double b=2;
7   double c=4;
8   double x1=0;
9   double x2=0;
10  int result=-1;
11  puts("shemoitanet kv. gant. koeficientebi: a, b, c");
12  scanf("%lf,%lf,%lf",&a,&b,&c);
13  result=solveSQeq(a,b,c,x1,x2);
14  switch(result)
15  {
16   case 0:
17   printf("D<0 ar aqvs amoxsna\n");
18   break;
19   case 1:
20   printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
21   printf("x1=%g:\n x2=%g\n", x1, x2);
22   break;
```

```
23   case 2:
24   printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
25   printf("x1=%g:\n x2=%g\n", x1, x2);
26   break;
27   default:
28   break;
29  }
30  return 0;
31 }
```

თუ შევადარებთ კოდებს 1.2 და 1.5 , ასევე 1.3 და 1.6 ვნახავთ რომ C++ ვარიანტში(უფრო სწორად მონაცემების მისამართით გადაცემისას) აღარაა საჭირო ფრჩხილები და „*“ გამოყენება. ასევე ფუნქციის გამოძახება (ხაზი 13 კოდში 1.6) არაფრით განსხვავდება ფუნქციის გამოძახებისგან, მისამართების ნაცვლად ცვლადის მნიშვნელობით გადაცემა რომ გამოგვეყენებინა(მაგრამ ამ შემთხვევაში როგორც ვნახეთ C-ს კურსში ცვლადის მნიშვნელობას „ვერ გამოვიტანდით“ ფუნქციის გარეთ).

როგორც ვხედავთ, მისამართის გადაცემის შემთხვევაშიც შენარჩუნებული გვაქვს `const` სპეციფიკატორი (ხაზი 5 კოდში 1.5), რაც ნიშნავს რომ აკრძალულია ამ ცვლადების მნიშვნელობების შეცვლა ფუნქციის შიგნით.

ხშირად `const` პარამეტრებს ფუნქციაში გადასცემენ მისამართის საშუალებით როგორც ზემოთ მაგალითში გვაქვს, და არა `const` ცვლადებს მიმთითებლის საშუალებით(როგორც ეს გვაქვს Cვარიანტში).

C++-ში არის შემთხვევები როცა ფუნქციაში ცვლადის გადაცემა მიმთითებლით აუცილებელია. მაგალითად როცა არ ვიცით რა ტიპის ცვლადია და საჭიროა ცვლადის დაყვანა(კასტირება, რეინტერპრეტაცია) ფუნქციის შიგნით.

სადაც შესაძლებელია, ჩვენ გამოვიყენებთ ცვლადის გადაცემას მისამართით, რაც კოდის ადვილად წაკითხვას და შეცდომებისგან ცოტა მეტ დაზღვევას განაპირობებს.

1.2 კლასები

C++ ენას თავიდან ერქვა „Cკლასებით“, რაც აღნიშნავდა რომ C++ წარმოადგენდა Cენის შემდგომ გაფართოებას. კლასის იდეა C++ ენაში შევიდა იმ დროს არსებული სხვა ობიექტზე ორიენტირებული ენებიდან *SmallTalk* და *Simula*.

კლასი `class` მსგავსია `struct`, მაგრამ თუ სტრუქტურაში ყველა წევრი ხელმისაწვდომია, კლასში შეიძლება კლასის წევრ ობიექტებზე პირდაპირი წვდომა აკრძალული იყოს. ასევე კლასში შეიძლება გვქონდეს ფუნქციები. მათ შორის სპეციალური ფუნქციები,

რომლებიც უზრუნველყოფენ კლასის ობიექტის შექმნას და წაშლას მეხსიერებაში. ამ ფუნქციების ეწოდებათ კლასის კონსტრუქტორი და დესტრუქტორი,

კლასში ღია წვდომის წევრები აღწერილი უნდა იყოს, როგორც `public`. წვდომა შეზღუდული წევრებისთვის გამოიყენება სიტყვა `private`.

C++ საშუალებას იძლევა წევრები და ფუნქციები გვქონდეს სტრუქტურაშიც. განსხვავება კლასისაგან ისაა, რომ თუ სპეციალურად აღნიშნული არაა სტრუქტურისთვის ყველა წევრი იგულისხმება თავისუფალი წვდომის წევრად (`public`), ხოლო კლასისათვის აკრძალული წვდომის წევრად (`private`).

ქვემოთ მოყვანილი კოდი კომპილირდება და მუშაობს როგორც C++, მაგრამ არა როგორც C.

```
1  typedef struct type_A
2  {
3   private:
4   double z;
5   public:
6   double x;
7   double y;
8   double getX()
9   { return x; }
10  void setX(double x_)
11  { x = x_; }
12  setZ(double _z)
13  { z=_z;}
14
15 } A;
16 int main(int argc, char* argv[])
17 {
18  A x;
19  x.x=1;
20  x.y=3;
```

```
21   x.z=1;// shecdomaa. z-ze pirdapiri cvdoma akrdzalulis
22   double t=x.getX();
23   x.setX(6);
24   x.setZ(6);//amis ufleba gvaqvs
25   return 0;
26  }
```

სწორი პრაქტიკაა შემდეგი: თუ ობიექტის წევრები არის თავისუფალი წვდომის და ობიექტისთვის რამდენიმე ფუნქცია გჭირდებათ აღწერეთ ეს ობიექტი როგორც სტრუქტურა. ცალკე აღწერეთ ამ ობიექტზე სამუშაო ფუნქციები(ანუ მოექეცით ობიექტს როგორც C ენის სტრუქტურას). წინააღმდეგ შემთხვევაში ობიექტისათვის შექმენით კლასი.

კლასის შექმნა და კლასთან მუშაობა განვიხილოთ `Point2D` კლასის მაგალითზე.

1.3 კლასი 2D წერტილი — `Point2D`

როცა რაღაცას განვიხილავთ როგორც ობიექტს და გვსურს შევქმნათ კლასი უნდა გავიაზროთ შემდეგი:

- ☑ რა თვისებები/წევრები აქვს ობიექტს
- ☑ რა თვისებები/წევრები გვსურს იყოს თავისუფლად ხელმისაწვდომი და რა შეზღუდული.
  ხშირად ყველა წევრს აკეთებენ `private` *და ამატებენ ფუნქციებს(ფუნქციებს უწოდებენ „მეთოდებს“) წევრებთან სამუშაოდ.*
- ☑ რა მეთოდები(ფუნქციები) გვჭირდება ობიექტთან სამუშაოდ. რა მეთოდები უნდა იყოს თავისუფლად ხელმისაწვდომი და რა შეზღუდული.
  არის შემთხვევები როცა გვჭირდება ფუნქცია კლასის შეიგნით სამუშაოდ, მაგრამ არაა საჭირო, რომ ეს ფუნქცია ხელმისაწვდომი იყოს კლასის გარეთ.
- ☑ რა ოპერაციები შეიძლება გაკეთდეს ობიექტებზე (რიცხვზე გამრავლება გაყოფა. ორი ობიექტის შეკრება გამოკლება.)

ორ განზომილებაში წერტილის კლასის (იხ. კოდების არქივი `Point2D`) (ჯერჯერობით განსხვავებას არ ვაკეთებთ წერტილს და ვექტორს შორის) მაგალითზე განვიხილავთ C++ კლასის ძირითად კონსტრუქციებს.

კლასის კონსტრუქცია არის შემდეგი: როგორც წესი გვაქვს ორი ფაილი `'კლასის სახელი'.h` და `'კლასის სახელი'.cpp`

```
1  //ფაილი 'კლასის სახელი'.h
2  class ' კლასის სახელი'{
3  public:
4  'კლასის კონსტრუქტორი1 'კლასის სახელი'(){}'
5  'კლასის კონსტრუქტორი2 'კლასის სახელი'(parametri)'
6  'კლასის კონსტრუქტორი3 'კლასის სახელი'(parametrebi)'
7  'კლასის წევრები გარედან თავისუფალიწვდომით'
8  'ფუნქციების დეკლარაცია'
9   'კლასის დესტრუქტორი '~ კლასის სახელი(){}'
10   private:
11  ' კლასის წევრები გარედან წვდომის შეზღუდვით'
12  ' კლასის ფუნქციების დეკლარაცია გარედან წვდომის შეზღუდვით'
13    public:
14  ' რაღაც წევრები ფუნქციები'
15   private:
16  ' რაღაც წევრები ფუნქციები'
17 };// „;‘‘- წერტილმძიმე აუცილებელია
```

განვიხილოთ `Point2df.h`. სწორი პრაქტიკაა, კლასის სახელი იყოს მარტივი და ასახავდეს კლასის არსს. ხშირად კლასის სახელს იწყებენ 'C' ასო ნიშნით, თუმცა ეს აუცილებელი მოთხოვნა არაა. როცა ბევრი კლასები გვაქვს ამას არავითარი უპირატესობა არ აქვს.

ჩვენს შემთხვევაში `Point` აღნიშნავს, რომ გვაქვს წერტილის კლასი; `2d`— გვაქვს 2 განზომილება და `f`— `float` ტიპი.

1.4 კონსტრუქტორი და დესტრუქტორი

კოდში 1.7 გამოცხადებული გვაქვს კლასი `Point2df`, რომელიც შედგება ორი `float` ტიპის ცვლადი `x` და `y` `public` წვდომის (ხაზი 5-8)..

ხაზზე 10-22 გამოცხადებული გვაქვს კლასის სამი კონსტრუქტორი. ანუ როგორ შეგვიძლია შევქმნათ კლასის ობიექტი. როგორც ვხედავთ ოთხი სხვადასხვა ვარიანტია:

1. ხაზი 13. ესაა ე.წ. ნაგულისხმევი (default) კონსტრუქტორი. იქმნება ობიექტი და მისი წევრები ინიციალიზდება ნულით

2. ხაზი 16. ესაა ე.წ. კოპირების კონსტრუქტორი. ობიექტი იქმნება სხვა ობიექტის კოპირების გზით. იქმნება ახალი ობიექტი და `v` ობიექტის `x` და `y` მნიშვნელობები ენიჭება ახალი ობიექტის `x` და `y` მნიშვნელობებს.
3. ხაზი 19. ობიექტი იქმნება `float` ტიპის მასივის საშუალებით.
4. ხაზი 22. ობიექტი იქმნება ორი `float` ტიპის ცვლადის საშუალებით.
5. ხაზი 65. ესაა დესტრუქტორი. როგორც ვხედავთ დესტრუქტორი ცარიელია, რაც ნიშნავს, რომ ობიექტი წაიშლება მეხსიერებიდან ისე როოგრც ეს განმარტებულია ოპერატიულ სისტემაში (ანუ უბრალოდ წაიშლება `x` და `y` ცვლადები ოპ. სისტემის მიერ). უფრო რთული კლასის შემთხვევაში, როცა მაგალითად დინამიურად ხდება ცვლადების ალოკაცია კონსტრუქტორში, დესტრუქტორში შეგვიძლია დავამატოთ დინამიურად ალოკირებული ცვლადების წაშლა.

კოდი 1.7. **კლასის დეკლარირება** `Point2df.h`

```
1  #ifndef POINT2DF_H
2  #define POINT2DF_H
3  #include <math.h>
4
5  class Point2df {
6  public:
7   float x;
8   float y;
9
10 public:
11
12  /// Default constructor; value is not initialized
13  Point2df() {x=0;y=0;}
14
15  /// Initialize from another vector
16  Point2df(const Point2df& v) :x(v.x), y(v.y) {}
17
18  /// Initialize from array of floats
```

```
19 Point2df(const float v[]) :x(v[0]), y(v[1]) {}
20
21 /// Initialize from components
22 Point2df(float xx, float yy) :x(xx), y(yy) {}
23
24 /// Return a non-const reference to the ith element
25 float& operator[](int i) { return (&x)[i]; }
26
27 /// Return a const reference to the ith element
28 const float& operator[](int i) const { return (&x)[i]; }
29
30 /// Assignment
31 Point2df& operator=(const Point2df& v) { x = v.x; y = v.y; return *this; }
32
33 /// Assignment from array of floats
34 Point2df& operator=(const float v[]) { x = v[0]; y = v[1]; return *this; }
35
36 /// Set value from another vector
37 Point2df& set(const Point2df& v) { x = v.x; y = v.y; return *this; }
38
39 /// Set value from array of floats
40 Point2df& set(const float v[]) { x = v[0]; y = v[1]; return *this; }
41
42 /// Set value from components
43 Point2df& set(float xx, float yy) { x = xx; y = yy; return *this; }
44
45 /// Assigning operators
46 Point2df& operator*=(float n) { return set(x*n, y*n); }
47 Point2df& operator/=(float n) { return set(x / n, y / n); }
```

```
48 Point2df& operator+=(const Point2df& v) { return set(x + v.x, y + v.y); }
49 Point2df& operator-=(const Point2df& v) { return set(x - v.x, y - v.y); }
50
51 /// Test if zero
52 bool operator!() const { return x == 0.0f && y == 0.0f; }
53
54 /// Unary
55 Point2df operator+() const { return *this; }
56 Point2df operator-() const { return Point2df(-x, -y); }
57
58 /// Length and square of length
59 float length2() const { return x*x + y*y; }
60 float length() const { return sqrt(length2()); }
61 float getX() const;
62 float getY() const;
63
64 /// Destructor
65 ~Point2df() {}
66 };
67
68
69 /// Dot product
70 inline float operator*(const Point2df& a, const Point2df& b)
71 { return a.x*b.x + a.y*b.y; }
72
73 /// Scaling
74 inline Point2df operator*(const Point2df& a, float n)
75 { return Point2df(a.x*n, a.y*n); }
76 inline Point2df operator*(float n, const Point2df& a)
```

```
77 { return Point2df(n*a.x, n*a.y); }
78 inline Point2df operator/(const Point2df& a, float n)
79 { return Point2df(a.x / n, a.y / n); }
80
81
82 /// Vector and vector addition
83 inline Point2df operator+(const Point2df& a, const Point2df& b)
84 { return Point2df(a.x + b.x, a.y + b.y); }
85 inline Point2df operator-(const Point2df& a, const Point2df& b)
86 { return Point2df(a.x - b.x, a.y - b.y); }
87
88
89
90 /// Equality tests
91 inline bool operator==(const Point2df& a, const Point2df& b)
92 { return a.x == b.x && a.y == b.y; }
93 inline bool operator!=(const Point2df& a, const Point2df& b)
94 { return a.x != b.x || a.y != b.y; }
95
96
97 /// Lowest or highest components
98 inline Point2df lo(const Point2df& a, const Point2df& b)
99 { return Point2df(fmin(a.x, b.x), fmin(a.y, b.y)); }
100 inline Point2df hi(const Point2df& a, const Point2df& b)
101 { return Point2df(fmax(a.x, b.x), fmax(a.y, b.y)); }
102
103 /// Normalize vector
104 extern Point2df normalize(const Point2df& v);
105
```

```
106
107 /// Linearly interpolate
108 extern Point2df lerp(const Point2df& u, const Point2df& v, float f);
109 extern Point2df perp(const Point2df &v);
110
111 #endif
```

1.5 ფუნქციის გადატვირთვა

C-ში აკრძალულია ორი ფუნქცია ერთი და იგივე სახელით. C++ შესაძლებელია რამდენიმე ფუნქცია ერთი და იგივე სახელით და ერთი და იგივე დაბრუნების ტიპით(ეს აუცილებელია), მაგრამ სხვადასხვა არგუმენტებით. ამას ეწოდება ფუნქციის გადატვირთვა.

ამის მაგალითია კოდი ხაზზე 37, 40 და 43. როგორც სხვადასხვა კონსტრუქტორის შემთხვევაში შეგვეძლო ობიექტის შექმნა სხვადასხვა გზით, ასევე შეგვიძლია სხვადასხვა გზით შევცვალოთ კლასის წევრები. ამას აკეთებს ფუნქცია `set`. კლასის წევრების მნიშვნელობების შეცვლა შეგვიძლია სხვა ობიექტის, რიცხვები მასივის ან ორი რიცხვის საშუალებით.

1.6 მიმთითებელი `this`

`this` არის მიმთითებელი რომელიც უთითებს კლასის ობიექტზე რომელშიც ვმუშაობთ, მაგალითად კლასის ფუნქციაში. ამის მაგალითია ზემოთ `set`ფუნქციის მაგალითი — შეიცვალა რა კლასის წევრების მნიშვნელობა `set` ფუნქცია აბრუნებს კლასის ობიექტის მისამართს, რომელიც შეიცვალა(რომელსაც ეკუთვნის ფუნქცია `set`).

1.7 ოპერატორები

C++ შესაძლებელია ოპერატორების განმარტება მომხმარებლის მიერ. ამას ეწოდება ოპერატორების გადატვირთვა. კოდის ხაზი [24-56] ნაწილში განმარტებულია ოპერატორები კლასის წევრებთან სამუშაოდ.

25 და 28 კვადრატული ფრჩხილები. ინდექსის(0 ან 1) მითითებით აბრუნებს x, y მნიშვნელობებს[1]. ყურადღება მიაქციეთ, რომ ინ-

[1] ყურადღება მიაქციეთ რომ კლასი იწყება წევრებით x და y. ეს ნიშნავს, რომ კლასის მისამართი იგივეა და x-ის მისამართი(არ გვაქვს "ღრეჩო"კლასის ობიექტს და კლასის პირველ წევრს შორის(მსგავსი მდგომარეობა გვქონდა მასივის შემთხვევაშიც)). ეს გვაძლევს საშუალებას გვქონდეს კვადრატული ფრჩხილების ოპერატორი(ხაზი 25 და 28). კლასის წევრებზე სწრაფად მიმართვისათვის უმჯობესია კლასის წევრები განვმარტოთ კლასის დასაწყისში. რა თანმიმდევრობით გვაქვს შემოტანილი კლასის წევრები იგივე თანმიმდევრობით იქმნებიან ისინი კლასის ობიექტში მეხსიერებაში ერთმანეთის გვერდით. სადაც მთავრდება ერთი იწყება მეორე(მასივის მსგავსად).

დექსის მნიშვნელობა არ მოწმდება. ანუ პროგრამისტი ვალდებულია სწორად მიუთითოს მნიშვნელობები. პირველი განმარტება საჭიროა არა მუდმივ ობიექტზე სამუშაოდ მეორე კი `const` ტიპის ობიექტზე სამუშაოდ.

31 და 34 ხაზებზე განმარტებულია მინიჭების ოპერატორი. პირველ შემთხვევაში ობიექტს ენიჭება მეორე ობიექტი. მეორე შემთხვევაში კი მასივი.

37,40,43 ხაზებზე განმარტებულია `set` რის შესახებაც ზემოთ ვისაუბრეთ

46-49 ხაზებზე განმარტებულია ობიექტზე „ოპერაცია=“ ოპერატორები ზემოთ აღწერილი `set` ფუნქციის გამოყენებით.

52 ხაზი ნულზე ტოლობის შემოწმების ოპერატორი (!)

55 და 56 ხაზი + და - ოპერატორები

59-62 განმარტებულია ფუნქციები: სიგრძის კვადრატი, სიგრძე და x და y მნიშვნელობების დაბრუნების ფუნქციები. როგორც ვხედავთ ეს ორი ფუნქცია მხოლოდ დეკლარირებულია. ცხადი სახე, ანუ იმპლემენტაცია მოცემულია კლასის შესაბამის `cpp` ფაილში.

შეგახსენებთ რომ `const` ფუნქციის სახელის შემდეგ აღნიშნავს, რომ ეს ფუნქცია არ ცვლის კლასის წევრების მნიშვნელობებს. `const` საშუალებას აძლევს კომპილატორს ოპტიმიზაცია გაუკეთოს კოდს თუ ეს შესაძლებელია.

ჩვენ ასევე შეგვეძლო ზემოთ მოყვანილი ფუნქციებისთვის და ოპერატორებისთვის კლასის `.h` ფაილში გვქონოდა მხოლოდ დეკლარაციები და იმპლემენტაციები გვქონოდა შესაბამის `cpp` ფაილში. ორივე მიდგომა თანაბარუფლებიანია და მუშაობს. კლასის შესაბამისი `cpp` ფაილის სახეს და სტრუქტურას მოგვიანებით განვიხილავთ.

65 ხაზზე გვაქვს დესტრუქტორი, რის შესახებაც ზემოთ გვქონდა საუბარი.

66 ხაზი ამთავრებს კლასის დეკლარაციას. წერტილმძიმე აუცილებელია

1.8 კლასის ობიექტებზე სამუშაო ფუნქციები

70-94 აღწერილი ფუნქციები არაა კლასის წევრები. ისინი უზრუნველყოფენ კლასის ობიექტებზე მუშაობას.

ხაზი 70: გამრავლების ოპერატორი განმარტავს ორი ვექტორის სკალარულ ნამრავლს.

74 და 76 ხაზზე აღწერილია წერტილის მარცხნიდან და მარჯვნიდან რიცხვზე გამრავლების ოპერატორი `*`.

78-ე ხაზზე წერტილის რიცხვზე გაყოფის ოპერატორი `/`

83 და 85 ხაზზე აღწერილია ორი ვექტორის შეკრება და გამოკლების ოპერატორები `+` და `-`

91 და 93 ხაზზე გვაქვს ოპერატორი `==`, რომელიც ამოწმებს ორი ვექტორის ტოლობას და ოპერატორი `!=`, რომელიც ამოწმებს ორი ვექტორის არა ტოლობას.

98 და 100 ხაზზე განმარტებულია ორი ფუნქცია, რომელიც იღებს არგუმენტებად ორ ვექტორს და ამ ორი ვექტორიდან პირველ

შემთხვევაში აგებს ვექტორის მინიმალური x y კოორდინატებით. მეორე შემთხვევაში კი მინიმალური x y კოორდინატებით. ეს ფუნქციები დაგვჭირდება ასეთ ამოცანაში: ვთქვათ მოცემულია რაღაც წერტილები. ვიპოვნოთ მინიმალური მართკუთხედი რომელიც მოიცავს ყველა განსახილველ წერტილს.

104 ხაზზე დეკლარირებულია ნორმალიზაციის ფუნქცია— იღებს ვექტორის მისამართს და აბრუნებს იგივე მიმართულების ერთეულოვან ვექტორს. ამ ფუნქციის იმპლემენტაცია გვაქვს მინიმალური cpp ფაილში და ქვემოთ განვიხილავთ. extern შეგვეძლო არც დაგვეწერა ვინაიდან ამ ფუნქციას ამ ფაილში არ ვიყენებთ, მაგრამ თავიდან რომ ავიცილოთ კომპილაციის თანმიმდევრობაზე და ერთი ფაილის მეორეში ჩართვაზე დამოკიდებულება გვაქვს extern, რაც კომპილატორს ეუბნება:„არ იდარდო ფუნქციის განმარტებაზე სადღაც გვაქვს განმარტებული“. თუ ფუნქცია განმარტებული არ იქნება შეცდომა გვექნება მიკავშირების (ლინკირების) ეტაპზე.

108 ხაზზე დეკლარირებულია წრფივი ინტერპოლაციის ფუნქცია.

109 ხაზზე დეკლარირებულია ფუნქცია რომელიც ითვლის და აბრუნებს მოცემული ვექტორის მართობულ ვექტორს.

extern იგივე ფუნქციას ასრულებს რასაც ზემოთ განხილულ ფუნქციებისთვის.

1.9 inline

inline ორი დატვირთვა აქვს. inline დეკლარირებული ფუნქცია ეუბნება კომპილატორს ფუნქციის ტანი ჩაეშენოს ფუნქციის კოდში გამოძახების ადგილას, როგორც ეს გვქონდა #define საშუალებით განმარტებული ფუნქციისთვის, მაგრამ #define-ისაგან განსხვავებით inline შემთხვევაში ეს კომპილატორისთვის არის რეკომენდაცია და კომპილატორი თავად წყვეტს ჩააშენოს ფუნქცია ორობით კოდში თუ არა. არის კიდევ ერთი სიტყვა __forceinline(მიკროსოფტ კომპილატორისთვის), რაც უკვე რჩევა კი არა ბრძანებაა კომპილატორისათვის ჩააშენოს ფუნქცია კოდში. შეძლებს თუ არა ამას კომპილატორი სხვა საკითხია.

```
1  #include <iostream>
2  #define mult1(a, b) a * b
3  #define mult2(a, b) (a) * (b)
4  #define mult3(a, b) ((a) * (b))
5
6  inline int multiply(int a, int b)
7  {
8   return a * b;
```

```
9  }
10
11 int main()
12 {
13  std::cout << (48 / mult1(2 + 2, 3 + 3)) << std::endl; // outputs 33
14  std::cout << (48 / mult2(2 + 2, 3 + 3)) << std::endl; // outputs 72
15  std::cout << (48 / mult3(2 + 2, 3 + 3)) << std::endl; // outputs 2
16  std::cout << (48 / multiply(2 + 2, 3 + 3)) << std::endl; // outputs 2
17
18  std::cout << mult3(2, 2.2) << std::endl; // no warning
19  std::cout << multiply(2, 2.2);
20 // Warning C4244 'argument': conversion from 'double' to 'int', possible loss of data
21 }
```

თითქოს 2,3 და 4 ხაზზე განმარტებული მაკროსები ერთი და იგივეა, მაგრამ სწორი და უსაფრთხო არის მე-4 ხაზზე მოცემული განმარტება. ამიტომ უნდა ვერიდოთ `#define` და გამოვიყენოთ `inline` ან `__forceinline`.

! gcc კომპილატორისთვის `__forceinline` ეკვივალენტი არის `#define __forceinline __attribute__((always_inline))`

`inline` მეორე დანიშნულება: ზემოთ მოყვანილ კლასის ფაილში `inline` საჭიროა იმიტომ, რომ ფუნქციების(რომლებიც არ არიან კლასის წევრები) დეკლარაცია და იმპლემენტაცია არის ერთ `.h` ფაილში. როცა ამ ფაილის რამდენჯერმე ჩართვა მოხდება პროექტის სხვადასხვა ფაილებში, ბინარული პროგრამული კოდის მიკავშირების(ლინკირების) ეტაპზე გვექნება შეცდომა ერთი და იგივე სიმბოლოების(ჩვენს შემთხვევაში ფუნქციების) მრავალჯერადად განმარტების შესახებ(მსგავსი პრობლემა გვქონდა როცა არ გვქონდა `#define`-ით განმარტებული მაკროსები .h ფაილში). `inline` უზრუნველყოფს, რომ ფუნქცია საბოლოო კოდში ჩართული იქნება მხოლოდ ერთხელ.

`inline` არ იქნებოდა საჭირო `.h` ფაილში რომ გვქონდეს მხოლოდ დეკლარაციები და იმპლემენტაციები გვქონდეს `.cpp` ფაილში. უფრო მეტიც კლასის ეს დამხმარე ფუნქციები დეკლარაციით და იმპლემენტაციით შეგვეძლო გაგვეკეთებინა არა კლასის ფაილებში არამედ სხვა `.h` და `.cpp` ფაილებში. ამ შემთხვევაში კლასის ფაილის ჩართვასთან ერთად ჩართვა უნდა გავუკეთოთ ამ დამხმარე

ფუნქციების `.h` ფაილსაც.

1.10 კლასის იმპლემენტაცია `Point2df.cpp`

კლასის იმპლემენტაციის ფაილს აქვს ასეთი სახე

```
1   //ფაილი 'კლასის სახელი'.cpp
2  #include "კლასისსახელი.h"
3  'ტიპი' ' კლასის სახელი'::  ფუნქცია1(არგუმენტები){
4  ფუნქციის აღწერა
5  return 'ტიპი';
6  }
7  'ტიპი' ' კლასის სახელი'::  ფუნქცია2(არგუმენტები){
8  ფუნქციის აღწერა
9  return 'ტიპი'ს ცვლადი;
10 }
11 ......
```

ცხადია ფუნქცია შეიძლება არაფერს არ აბრუნებდეს. ფუნქციის ქვეშ შეგვიძლია ვიგულისხმოთ კლასის კონსტრუქტორი და დესტრუქტორიც. `Point2df` კლასის შემთხვევაში იმპლემენტაციის ფაილი მოყვანილია ქვემოთ **??**.

როგორც ვხედავთ აქ განმარტებულია როგორც კლასის შიგნით დეკლარირებული ფუნქციები ასევე დამხმარე ფუნქციები რომლებიც დეკლარირებული იყო კლასის ფაილში.

1.11 `Point2df` კლასის გამოყენების მაგალითი

კლასის გამოყენების მაგალითი იხილეთ *Point2df* არქივში
არქივში ასევე არის სამკუთხედის `Triangle` კლასი.

დავალება 1.1: სამკუთხედის კლასი

სამუთხედის კლასისთვის დაამატეთ:

1. მინიჭების ოპერატორი
2. ტოლობაზე შედარების ოპერატორი
3. არტოლობაზე შედარების ოპერატორი
4. კლასში დაამატეთ `private` წევრები პერიმეტრი და ფართობი.
5. პერიმეტრის გამოთვლის ფუნქცია.
6. ფართობის გამოთვლის ფუნქცია.
7. getPerimeter და getArea ფუნქციები რომელიც დააბრუნებს პერიმეტრს და ფართობს თუ ისინი გამოთვლილია და თუ გამოთვლილი არაა გამოთვლის და შემდეგ დააბრუნებს მიღებულ მნიშვნელობას.
8. set და get ფუნქციები(set ღებულობს სამ წერტილს და ანიჭებს სამკუთხედის წევრ წერტილებს. get მიეწოდება სამი წერტილის მიმთითებლები და ამ წერტილებში კოპირდება სამკუთხედის შემადგენელი წერტილების მონაცემები)
9. ფუნქცია, რომელიც ეკრანზე ბეჭდავს სამკუთხედის წერტილების კოორდინატებს.
10. ფუნქცია, რომელიც ეკრანზე ბეჭდავს სამკუთხედის ფართობს და პერიმეტრს.

1.12 `friend` ფუნქციები

განხილულ კოდში `x` `y` ცვლადები დეკლარირებულია როგორც `public`. შეცვალეთ კოდში დეკლარაცია `private`-ით და ნახავთ რომ კოდი კომპილაციისას მოგვცემს შეცდომას. ეს იმიტომ, რომ ობიექტებთან სამუშაო ფუნქციებს, რომლებიც არ არიან კლასის წევრები, აკრძალული აქვთ `x` და `y` წევრებზე წვდომა. იმისათვის, რომ ასეთ შემთხვევებში კოდმა იმუშაოს საჭიროა ეს ოპერატორები და ფუნქციები გამოვაცხადოთ კლასის „მეგობრებად“. იხილეთ არქივი `Point2D_private`. ამ არქივიდან კოდის ნაწილი `Point2D.h` ფაილიდან მოყვანილია ქვემოთ პროგრამის კოდში **??**.

კლასის ობიექტებთან სამუშაო ოპერატორები და ფუნქციები გადატანილია კლასის შიგნით და დეკლარაციაში დამატებულია სიტყვა `friend`. არ არის აუცილებელი ოპერატორის/ფუნქციის იმპლემენტაცია იყოს კლასის შიგნით. მხოლოდ დეკლარაცია საკმარისია, ხოლო იმპლემენტაცია შეიძლება იყოს სხვა ფაილში, მაგალითად კლასის შესაბამის `.cpp` ფაილში, მაგრამ ცხადია ეს ოპერატორი ფუნქციები არ არიან კლასის წევრები.

კოდი 1.8 კლასი Point2D.h private წევრებით

```
65 /// Dot product
66 inline friend float operator*(const Point2df& a, const Point2df& b) {
67  return a.x*b.x + a.y*b.y; }
68
69 /// Scaling
70 inline friend Point2df operator*(const Point2df& a, float n) {
71  return Point2df(a.x*n, a.y*n); }
72 inline friend Point2df operator*(float n, const Point2df& a) {
73  return Point2df(n*a.x, n*a.y); }
74 inline friend Point2df operator/(const Point2df& a, float n) {
75  return Point2df(a.x / n, a.y / n); }
76
77 /// Vector and vector addition
78 inline friend Point2df operator+(const Point2df& a, const Point2df& b) {
79  return Point2df(a.x + b.x, a.y + b.y); }
80 inline friend Point2df operator-(const Point2df& a, const Point2df& b) {
81  return Point2df(a.x - b.x, a.y - b.y); }
82
83 // Equality tests
84 inline friend bool operator==(const Point2df& a, const Point2df& b) {
85  return a.x == b.x && a.y == b.y; }
86 inline friend bool operator!=(const Point2df& a, const Point2df& b) {
87  return a.x != b.x || a.y != b.y; }
88
89 // Lowest or highest components
90 inline friend Point2df lo(const Point2df& a, const Point2df& b) {
91  return Point2df(fmin(a.x, b.x), fmin(a.y, b.y)); }
92 inline friend Point2df hi(const Point2df& a, const Point2df& b) {
```

```
93  return Point2df(fmax(a.x, b.x), fmax(a.y, b.y)); }
94
95 /// Normalize vector
96 friend  Point2df normalize(const Point2df& v);
97
98 /// Linearly interpolate
99 friend  Point2df lerp(const Point2df& u, const Point2df& v, float f);
100 friend Point2df perp(const Point2df &v);
```

1.13 წინასწარი დეკლარირება და ორი კლასის მეგობარი ფუნქცია

შესაძლებელია რომ ერთ კლასს სამუშაოდ სჭირდებოდეს მეორე კლასი ან პირიქით მეორეს პირველი. ამ შემთხვევაში თუ საჭირო კლასის ობიექტი უკვე დაკომპილირებული არ იქნა მივიღებთ შეცდომას. წინასწარი დეკლარირება მსგავსია `extern` გამოცხადების, როცა კომპილატორს ვეუბნებით რომ არ იფიქროს ფუნქციის იმპლემენტაციაზე. განვიხილოთ მაგალითი

```
1  #include <iostream>
2  using namespace std;
3
4  // Add members of two different classes using friend functions
5  // forward declaration
6  class ClassB;
7  class ClassA {
8   public:
9   // constructor to initialize numA to 12
10  ClassA() : numA(12) {}
11
12  private:
13  int numA;
14
```

```
15  // friend function declaration
16  friend int add(ClassA, ClassB);
17 };
18
19 class ClassB {
20
21  public:
22  // constructor to initialize numB to 1
23  ClassB() : numB(1) {}
24
25  private:
26  int numB;
27
28  // friend function declaration
29  friend int add(ClassA, ClassB);
30 };
31 // access members of both classes
32 int add(ClassA objectA, ClassB objectB) {
33  return (objectA.numA + objectB.numB);
34 }
35 int main() {
36  ClassA objectA;
37  ClassB objectB;
38  "Sum: " << add(objectA, objectB);
39  return 0;
40 }
```

1.14 `friend` კლასი

კლასის „მეგობარი“ შეიძლება იყოს არა მხოლოდ გლობალური ფუნქცია არამედ სხვა კლასიც

```cpp
#include <iostream>
using namespace std;
class A {
 private:
 int private_variable;
 public:
 A()
 {
  private_variable = 10;
 }
 // friend class declaration
 friend class C;
};
// Here, class C is declared as a friend inside class A. Therefore,
// C is a friend of class A. Class C can access the private members of class A.
class C {
 public:
 void display(A& objectA)
 {
  cout << "The value of Private Variable = "
  << objectA.private_variable << endl;
 }
};
// Driver code
int main()
{
 A g;
 C fri;
 fri.display(g);
```

```
30  return 0;
31 }
```

1.15 friend კლასის ფუნქცია

მეგობრად ასევე შეიძლება გამოცხადდეს არა კლასი არამედ კლასის ფუნქციაც

```
1  #include <iostream>
2  using namespace std;
3  class A; // forward definition needed
4  // another class in which function is declared
5  class anotherClass {
6   public:
7   void memberFunction(A& obj);
8  };
9
10 // base class for which friend is declared
11 class A {
12  private:
13  int private_variable;
14
15  public:
16  A()
17  {
18   private_variable = 10;
19  }
20
21  // friend function declaration
22  friend void anotherClass::memberFunction(A&);
23 };
```

```
24
25 // friend function definition
26 void anotherClass::memberFunction(A& obj)
27 {
28  cout << "Private Variable: " << obj.private_variable
29  << endl;
30 }
31
32 // driver code
33 int main()
34 {
35  A object1;
36  anotherClass object2;
37  object2.memberFunction(object1);
38  return 0;
39 }
```

1.16 namespace

ზემოთ კოდებში რამდენჯერმე შეგვხვდა `using namespace std;` C++ შემოტანილია ე.წ. `namespace` ცნება. ესაა არე რომლის შიგნითაც შეიძლება განმარტებულ იქნას კლასები, ცვლადები, ფუნქციები.

`namespace` უზრუნველყოფს რომ როცა ვიყენებთ რამდენიმე ბიბლიოთეკას გამოვიყენოთ მოცემული ფუნქცია/კლასი მოცემული ბიბლიოთეკიდან. ანუ შეიძლება გვქონდეს სხვადასხვა კლასები ფუნქციები ერთი და იგივე სახელით, მაგრამ თუ ისინი განმარტებულია სხვადასხვა `namespace`-ში გამოყენება არ იქნება პრობლემა. განვიხილოთ მაგალითი:

```
1 #include<stdio.h>
2 namespace NameSpaceA
3 {
4  class A
```

```
 {
  public:
  void DoSomething() {}
 };
 void Func(A &objectA) {}
}
```

განმარტებული გვაქვს `namespace NameSpaceA` და მის შიგნით რაღაც კლასი და ფუნქცია. ამ კლასზე და ფუნქციაზე მუშაობა შეიძლება რამდენიმე გზით, მაგალითად ვუთითებთ `namespace` და კლასს/ფუნქციას

```
NameSpaceA::A aklasisobieqti;
aklasisobieqti.DoSomething();
NameSpaceA::Func(aklasisobieqti);
```

ან სიტყვა `using` გამოყენებით

```
using NameSpaceA::A;
A mgr;
mgr.DoSomething();
```

ან ვაცხადებთ `namespace NameSpaceA` გლობალურად

```
using namespace NameSpaceA;
A mgr;
mgr.DoSomething();
Func(mgr);
```

ეს არაა სასურველი როცა რამდენიმე `namespace` გვაქვს. ერთი შეიძლება გამოვაცხადოთ გლობალურად და სხვა `namespace`-ების ელემენტებს, ფუნქციებს კლასებს მივმართოთ „`::`“ საშუალებით.

1.17 std::vector

C++-ში გვაქვს ე.წ. სტანდარტული ბიბლიოთეკა. namespace std სწორედ სტანდარტული ბიბლიოთეკის „ნეიმსპეისია“. განვიხილოთ სტანდარტული ბიბლიოთეკის (***Standard Template Library (STL)***) კონტეინერ კლასი ვექტორი (vector.)

vector არის დინამიური მასივი რომელშიც შეიძლება გვქონდეს ერთი ტიპის ელემენტები. მას შეუძლია ავტომატურად შეიცვალოს ზომა როცა ხდება ელემენტის დამატება ან ელემენტის წაშლა.

vector არის შაბლონური კლასი, რაც ნიშნავს რომ ტიპი არის პარამეტრი და რა ტიპზეა საუბარი ეს მომხმარებელმა უნდა დააკონკრეტოს. vector<T> v; T არის პარამეტრი. შეიძლება იყოს ნებისმიერი ტიპი C++ ენაში არსებული ტიპი(მათ შორის მიმთითებელიც) ან მომხმარებლის მიერ განმარტებული კლასი(მიმთითებელი კლასზე).

განვიხილოთ მაგალითი:

კოდი 1.9 vector ობიექტი

```
#include <vector>
#include <iostream>

int main() {
 //ინიციალიზაციის მაგალითები

 std::vector<int> numbers = {1, 2, 3, 4, 5};
 //ცხადი სახით ინიციალიზაცია

 int firstNumber = numbers[0]; // firstNumber will be 1
 std::cout << "The first number is: " << firstNumber << std::endl;

  int arr[] = {11, 23, 45, 89};
 int n = sizeof(arr) / sizeof(arr[0]);

 // Initialize the std::vector v by arr
 vector<int> v = {arr, arr + n};
```

```
18  //მასივით ინიციალიზაცია
19
20
21   vector<int> v1 = {11, 23, 45, 89};
22
23  // Initialize the vector v2 from vector v1
24  // სხვა ვექტორით ინიციალიზაცია
25
26  vector<int> v2(v1.begin(), v1.end());
27
28   // Initializing all the elements of a
29  // vector using a single value
30
31 //ინიციალიზაცია კონსტანტით.
32   // იქმნება 5 ელემენტიანი მასივი
33   // ყველა ელემენტი არის 11
34  vector<int> v3(5, 11);
35  //იქმნება მასივი ზომით 0
36   vector<int> v4;
37  return 0;
38 }
```

ვექტორზე მუშაობა:

კოდი 1.10 vector გამოყენება

```
1  #include <vector>
2  #include <iostream>
3
4  int main() {
```

```
5   std::vector<int> numbers = {1, 2, 3, 4, 5};
6   numbers[1] = 20; // The second element of the vector will now be 20
7   std::cout << "The modified second number is: " << numbers[1] << std::endl;
8   numbers.push_back(4); // Adds the number 4 to the end of the vector
9   numbers.pop_back(); // Removes the last element (4)
10  return 0;
11 }
```

1.18 იტერატორი

იტერატორი C++-ში არის მიმთითებლის მსგავსი ობიექტი, რომელიც მიუთითებს STL კონტეინერის ელემენტზე. ისინი ძირითადად გამოიყენება C++-ში STL კონტეინერის ელემენტებთან სამუშაოდ. STL იტერატორების მთავარი უპირატესობა ის არის, იტერატორების გამოყენებით STL ალგორითმები დამოუკიდებლი ხდება გამოყენებული კონტეინერის ტიპისგან. ჩვენ შეგვიძლია უბრალოდ მივაწოდოთ კონტეინერის იტერატორი ალგორითმს და არა თავად კონტეინერი ან მისი ელემენტი.

იტერატორის ფუმქცია	დაბრუნებული მნიშვნელობა
begin()	აბრუნებს იტერატორს რომელიც უთითებს კონტეინერის დასაწყისზე.
end()	აბრუნებს იტერატორს კონტეინერის ბოლო ელემენტის შემდგომ ელემენტზე. ესაა სპეც. ელემენტი კონტეინერებში რომელიც უთითებს კონტეინერის დასასრულს.
cbegin()	იგვეა რაც begin() ოღნოდ აბრუნებს მუდმივ იტერატორს(ანუ შეუძლებელია ელემენტის მნიშვნელობის შეცვლა, რომელზეც უთითებს იტერატორი).
cend()	იგივეა რაც end() ოღონდ აბრუნებს მუდმივ იტერატორს.
rbegin()	ესაა შებრუნებული იტერატორი რომელიც უთითებს დასაწყისს(ანუ ბოლოს).
rend()	ესაა შებრუნებული იტერატორი რომელიც უთითებს კონტეინერის დასასრულს.
crbegin()	ესაა შებრუნებული მუდმივი იტერატორი რომელიც უთითებს დასაწყისს.
crend()	ესაა შებრუნებული მუდმივი იტერატორი რომელიც უთითებს კონტეინერის დასასრულს.

იტერატორის მაგალითი:

```
1  // Create a vector called cars that will store strings
2  vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
3
4  // Create a vector iterator called it
5  vector<string>::iterator it;
6
7  // Loop through the vector with the iterator
8  for (it = cars.begin(); it != cars.end(); ++it) {
9   cout << *it << "\n";
10 }
```

1.19 std::vector და Point2df მაგალითი

იხ. არქივი *randomPoints*. utility.h ფაილში დეკლარირებულია int random_number(int low_num, int hi_num) ფუნქცია, რომელიც აგენერირებს შემთხვევით რიცხვს მოცემულ შუალედში. იმპლემენტაცია მოყვანილია ქვემოთ.

კოდი 1.11 random_number ფუნქციის იმპლემენტაცია ფაილი utility.cpp

```
1  #include "utility.h"
2  #include <stdlib.h>
3  #include<time.h>
4  #include<stdio.h>
5  int random_number(int low_num, int hi_num)
6  {
7   int result = 0;
8   result = (rand() % (hi_num+1 - low_num)) + low_num;
9   return result;
10 }
```

randomPoints.cpp კოდი მოყვანილია ქვემოთ. აქ დემონსტრირებულია წერილის კლასის ფუნქციები lo და hi.

14-21 ხაზზე გენერირდება შემთხვევითი წერტილები და ემატება m_points მასივს.

24-28 ხაზზე ციკლში გავდივართ მასივს და მასივის წერილებიდან ვაგებთ left(მარცხენა დაბალი წერტილი) და right(მარჯვენა

მაღალი წერტილი). ეს ორი წერილი გვაძლევს ისეთი მინიმალური მართკუთხედის ორ მოპირდაპირე წვეროს, რომელიც მოიცავს ყველა წერტილს.

კოდი 1.12. ფაილი randomPoints.cpp

```
#include "Point2df.h"
#include "utility.h"
#include <stdlib.h>
#include<time.h>
#include <stdio.h>
#include <vector>
using namespace std;
vector<Point2df> m_points;
int main()
{
 srand(time(NULL));
 int low = -50;
 int high = 50;
 for (int i=low;i<high;i++)
 {
  int ival_x = random_number(low, high);
  int ival_y = random_number(low, high);
  Point2df rpoint((float)ival_x, (float)ival_y);
  m_points.push_back(rpoint);
  printf("p.x=%f,p.y=%f\n", rpoint.x, rpoint.y);
 }
 Point2df left(m_points[0]);
 Point2df rigth(m_points[0]);
 for (size_t i = 1; i < m_points.size(); i++)
 {
```

```
26  left = lo(left, m_points[i]);
27  rigth = hi(rigth, m_points[i]);
28 }
29 printf("left.x=%f,left.y=%f\n", left.x, left.y);
30 printf("rigth.x=%f,rigth.y=%f\n", rigth.x, rigth.y);
31    return 0;
32 }
```

1.20 ნაწარმოები კლასი (შთამომავლები)

C++ შეგვიძლია შევქმნათ კლასი(ეს იქნება ნაწარმოები კლასი) სხვა კლასის(ეწოდება საბაზო,აბსტრაქტული კლასი, წინაპარი კლასი) ან კლასების საფუძველზე(ანუ ნაწარმოები იყოს ორი ან რამდენიმე კლასისაგან). რის შედეგადაც ნაწარმოებ კლასს ექნება ყველა ის თვისება რაც აქვს საბაზო კლასს(კლასებს) და აუცილებლობა აღარაა განმარტებულ იქნას ეს თვისებები ნაწარმოები კლასისთვის.

ხშირად განიხილავენ ასეთ მაგალითს. ვთქვათ შევქმენით კლასი „ცხოველი“. ამ კლასში გვექნება თვისებები რაც ახასიათებს ზოგადად „ცხოველს“. „ცხოველი“ კლასიდან შეგვიძლია ვაწარმოოთ სხვადასხვა კლასები. მაგალითად, კლასი „ძაღლი“, „კატა“, „მგელი“, „ცხვარი“. თითოეულს ექნება ყველა თვისება და ფუნქცია რაც აქვს კლასს „ცხოველს“. ამას გარდა თითოეულ კლასს შეგვიძლია დავუმატოთ ამ კლასის ცხოველის დამახასიათებელი თვისებები და ფუნქციები (მაგალითად, შეგვიძლია გვქონდეს ფუნქცია „ხმა“ საბაზისო კლასში, რომელიც სხვადასხვა ცხოველის შემთხვევაში გახსნის და გაუშვებს(დაუკრავს) ცხოველის შესაბამის ხმის ფაილს).

შეგვიძლია ავიღოთ უფრო ზოგადი სქემა, მაგალითად: „სულიერი არსება“, „ფრინველი“ და „ცხოველი“ შეგვიძლია ვაწარმოოთ ამ კლასიდან. შემდეგ „ფრინველი“ კლასიდან ვაწარმოოთ სხვადასხვა ფრინველის შესაბამისი კლასები და „ცხოველი“ კლასიდან ვაწარმოოთ ცხოველის შესაბამისად სხვადასხვა კლასები. ასე რომ კლასი „ძაღლი“ პირველ შემთხვევაში თუ იყო ნაწარმოები „ცხოველიდან“, მეორე შემთხვევაში „ძაღლი“ ნაწარმოები იქნება „ცხოველიდან“, რომელიც თავის მხრივ ნაწარმოებია „სულიერი არსებიდან“.

ამოცანიდან გამომდინარე შეიძლება გვქონდეს აბსტრაქციის სხვადასხვა დონე. სანამ კლასებს გამოვაცხადებთ უნდა გავიაზროთ ამოცანა ზოგადად. რა ობიექტები გვექნება. რა ურთიერთობაა ობიექტებს შორის. რა თვისებები შეიძლება განვაზოგადოთ და ა.შ. განზოგადების რაღაც დონე აუცილებელი იქნება მაგრამ ეს უნდა იყოს მინიმალური. როგორც წესი ერთი ან ორი თაობა (ერთი ან ორი „წინაპარი“) საკმარისია.

1.21 protected წევრები

განვიხილოთ პროექტი derivedClass. შემოტანილი გვაქვს შეკრული, მარტივი(თვით გადაკვეთის გარეშე) ობიექტის კლასი Solid2D. ეს აღწერს ობიექტს რომლისთვისაც განმარტებულია ფართობი. ანალოგი 3D შემთხვევაში იქნება ობიექტი რომლისთვისაც განმარტებულია მოცულობა. Solid2D კლასის განმარტება მოყვანილია ქვემოთ.

კოდი 1.13 კლასი Solid2D

```
#ifndef _SOLID2D_H
#define _SOLID2D_H
#include "Point2df.h"
#include <vector>
#include <string>
#define M_PI 3.14159265358979323846

class Solid2D
{
 protected:
 Point2df m_Position;
 std::vector<Point2df>m_points;
 std::string m_name;
 public:
 Solid2D():m_Position(0,0),m_name("Unnamed Solid") {};
 Solid2D(const Point2df& pos) { m_Position = pos; };
 Solid2D(const Solid2D& s)
 {
  m_Position = s.m_Position;
  m_points = s.m_points;
  m_name=s.m_name;
  m_name.append("_copy");
 };
```

```
24  Solid2D(float x, float y):m_name("Unnamed Solid") {
25   m_Position.x = x; m_Position.y = y;}
26  void setPosition(const Point2df& pos);
27  void setPosition(float x, float y) { m_Position.x = x; m_Position.y = y; }
28  const Point2df& getPosition() const;
29  std::string  getName()const{return m_name;}
30  void setName(std::string name){m_name=name;}
31  void setName(const char* name)
32  {
33   m_name.assign(name);
34  }
35  virtual float Perimeter() const;
36  virtual float Area() const = 0 ;
37
38  ~Solid2D() {};
39 };
40 #endif
```

ეს იქნება ზოგადი(ბაზური) კლასი ობიექტებისთვის რომლისთვისაც განმარტებული ფართობი.
Solid2D იმპლემენტაციის ფაილი:

კოდი 1.14. ფაილი Solid2D.cpp

```
1 #include "Solid2D.h"
2 void Solid2D::setPosition(const Point2df& pos)
3 {
4  m_Position = pos;
5 }
6 const Point2df& Solid2D::getPosition() const
7 {
```

```
8  return m_Position;
9 }
10
11 float Solid2D::Perimeter() const
12 {
13  const int n = m_points.size();
14  const Point2df diff = m_points[n - 1] - m_points[0];
15  float perimeter = diff.length();
16  for (int i=0;i<n-1;i++)
17  {
18   const float dist = (m_points[i] - m_points[i + 1]).length();
19   perimeter += dist;
20  }
21 return perimeter;
22 }
```

ბაზა კლასს აქვს თვისებები რაც საერთო შეიძლება იყოს 2D ფართობის მქონე ობიექტებისათვის:

- მდებარეობა
- სახელი
- წერტილების მასივი (ამას არ ვიყენებთ ზოგიერთი ობიექტისთვის, მაგრამ სასურველია რომ გვქონდეს. თუმცა აუცილებელია არაა. მომავალში ვნახავთ)

ასევე განმარტებული გვაქვს ფუნქციები:

- მდებარეობის მიღება(get) მიცემა(set).
- სახელის მიღება(get) მიცემა(set).
- პერიმეტრი გამოთვლა.
- ფართობის გამოთვლა.

კოდში არის რამდენიმე ახალი კონცეფცია, რის შესახებაც ადრე არ გვქონდა საუბარი. private ან public ნაცვლად ცვლადები დეკლარირებული გვაქვს როგორც protected. რაც ნიშნავს, რომ ნაწარმოებ კლასს აქვს უფლება შეცვალოს ეს წევრები. უცხო, გარე

კლასს ისევ ეკრძალება ამ წევრების შეცვლას(ანალოგიურად როგორც გვქონდა `private` შემთხვევაში)

1.22 `virtual` — ვირტუალური და წმინდა ვირტუალური ფუნქციები

კოდ 1.13-ში დეკლარირებული გვაქვს პერიმეტრის და ფართობის გამოსათვლელი ვირტუალური ფუნქციები (ხაზი 35, 36):

```
virtual float Perimeter() const;
virtual float Area() const = 0 ;
```

`Perimeter()` დეკლარირებული და განმარტებულია ბაზურ კლასში. ნაწარმოებ კლასში თუ განმარტებული იქნება `Perimeter()` გამოძახებული იქნება ნაწარმოები კლასის ფუნქცია. თუ არ იქნება განმარტებული გამოძახებული იქნება ბაზური კლასის ფუნქცია. `virtual float Area() const = 0 ;` ესაა წმინდა ვირტუალური ფუნქცია. რაც ნიშნავს რომ ვალდებული ვართ განვმარტოთ ეს ფუნქცია ყოველი ნაწარმოები კლასისათვის. თუ არა, კოდი არ დაკომპილირდება.

`Solid2D` კლასიდან ნაწარმოები გვაქვს ორი კლასი `Circle` წრეწირი და `Rect2df` მართკუთხედი. ამ კლასებთან მუშაობის მაგალითი მოყვანილია ფაილში `testclass.cpp`

კოდი 1.15. ფაილი `testclass.cpp`

```
1
2  #include "Circle.h"
3  #include "Rect2df.h"
4  #include<stdio.h>
5  #include<vector>
6  #include <iostream>
7  #include "utility.h"
8  int main()
9  {
10
11  Circle c1(0, 0, 1);
12  float per = c1.Perimeter();
13     c1.setName("Circle 1");
14  c1.setPosition(1,2);
```

```
15    printCircle(stdout,c1);
16  Circle c2(c1);
17
18  c2.setPosition(6,7);
19  per = c2.Perimeter();
20  printCircle(stdout,c2);
21  Point2df c1pos=c1.getPosition();
22  Rect2df rec(c1pos,2,4);
23  rec.setName("Rect 1");
24    float pp=rec.Solid2D::Perimeter();
25  c1 = c2;
26  puts("c1-s mienicha c2");
27  per= c1.Perimeter();
28
29  printCircle(stdout,c1);
30  std::vector<Solid2D*> solids;
31  solids.push_back(&c1);
32  solids.push_back(&c2);
33  solids.push_back(&rec);
34  for (size_t i=0;i<solids.size();i++)
35    {
36        const Solid2D *solid=solids[i];
37        const Point2df pos=solid->getPosition();
38        const std::string name=solid->getName();
39        const float per=solid->Perimeter();
40        const float area=solid->Area();
41        printf("name=%s, Position(x,y)=(%f,%f); Perimeter=%f, Area=%f\n",
42               name.c_str(),pos.x,pos.y, per, area);
43    }
```

```
44    return 0;
45 }
```

1.23 ტიპის გარდაქმნა(კასტირება) C++ში

ტიპის გარდაქმნა რაც გვაქვს C-ში ჩაშენებული ტიპებისთვის მუშაობს C++ში, მაგრამ C++ დამატებულია კიდევ ოთხი მექანიზმი:

1. `static_cast` გამოიყენება სტანდარტული ტიპების გარდაქმნისათვის

კოდი 1.16 static_cast

```
1  float myfloat = 3.14;
2  // convert float to int
3  int myint = static_cast<int>(myfloat);
```

2. `dynamic_cast` გამოიყენება პოლიმორფული ტიპის გარდაქმნებისათვის. კერძოდ, როცა საჭიროა ბაზური ტიპის მიმთითებლიდან აღვადგინოთ ნაწარმოები ტიპი. მაგალითად, ზემოთმოყვანილი `Solid2D` და `Circle` შემთხვევაში გვექნება:

კოდი 1.17 dynamic_cast

```
1  // create a Base class pointer
2  // pointing to a Derived object
3  Solid2D  *base_ptr = new Circle();
4  // use dynamic_cast to cast the
5  // Base pointer to a Derived pointer
6  Circle *derived_ptr = dynamic_cast<Circle*>(base_ptr);
7
8  // call the print function through the derived pointer
9  if (derived_ptr) {
10  derived_ptr->Area();
11 }
12 // delete the dynamically allocated object
```

```
13  delete base_ptr;
```

3. const_cast გამოიყენება const მოხსნისათვის

კოდი 1.18 const_cast

```
1  int x = 10;
2  // a const pointer for variable x
3  const int* ptr = &x;
4  // use const_cast to
5  // remove const qualifier and allow modification
6  int* mutableptr = const_cast<int*>(ptr);
7  // call the function
8  modify_data(mutableptr);
```

4. reinterpret_cast გამოიყენება ერთი ტიპის მისამართის ან მიმთითებლის მეორე ტიპის მიმთითებელში ან მისამართში გარდასაქმნელად, მაგრამ სტატიკური და დინამიური კასტრირებისგან განსხვავებით ეს ოპერაცია არ აკეთებს რეალურად ტიპის გარდაქმნას. ეს ოპერაცია ნიშნავს "მოექეცი როგორც"ან "შეხედე როგორც":

კოდი 1.19 reinterpret_cast

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4   // create an integer variable
5   int x = 67;
6
7   // pointer to an integer
8   int* ptr_to_int = &x;
9
10  // reinterpret the pointer to an integer
```

```
11  // as a pointer to char
12  char* ptr_to_char = reinterpret_cast<char*>(ptr_to_int);
13
14  // dereference the double pointer
15  // originally holding an integer as if it contains a double
16  cout << "Dereferencing ptr_to_char: " << *ptr_to_char << endl;
17
18  return 0;
19 }
```

ამიტომ ამ ოპერაციის გამოყენებისას ფრთხილად უნდა ვიყოთ. ეს ოპერაცია ძირითადად გამოიყენება `void *` ტიპის მიმთითებლებისთვის როცა ვიცით რა ტიპი დგას `void *` უკან.

1.24 დინამიური ელემენტები, `new` და `delete`

მასივები და ცვლადები სტეკში (ავტომატური ცვლადები) მუშაობს ზუსტად ისევე როგორც ეს გვქონდა C. რაც შეეხება დინამიურ ცვლადებს ე.წ. C სტილით ალოკაცია (`malloc`/ `calloc` `free`) დეალოკაცია იმუშავებს, მაგრამ *C++* გვაქვს თავის ფუნქციები `new` და `delete`, რომლებიც ცხადია მუშაობენ არა მხოლოდ ენაში „ჩაშენებული“ ცვლადებისათვის არემედ მომხმარებლის მიერ განმარტებული კლასებისათვისაც.

კოდი 1.20 `new` და `delete`

```
1  #include <iostream>
2  #include <memory>
3  using namespace std;
4
5  int main() {
6
7   // Declared a pointer to store
8   // the address of the allocated memory
```

```
9  int *nptr;
10
11 // Allocate and initialize memory
12 nptr = new int(6);
13
14 // Print the value
15 cout << *nptr << endl;
16
17 // Print the address of memory
18 // block
19 cout << nptr;
20 delete nptr;
21 nptr=NULL;
22 return 0;
23 }
```

მასივის ალოკაციის მაგალითი

კოდი 1.21 მასივის ალოკაცია

```
1 #include <iostream>
2 #include <memory>
3 using namespace std;
4
5 int main() {
6
7 // Declared a pointer to store
8 // the address of the allocated memory
9 int *nptr;
10
```

```
11  // Allocate and initialize array of
12  // integer with 5 elements
13  nptr = new int[5]{1, 2, 3, 4, 5};
14
15  // Print array
16  for (int i = 0; i < 5; i++)
17  cout << nptr[i] << " ";
18  delete[] nptr;
19  return 0;
20 }
```

ყურადღება მიაქციეთ მეხსიერების გასუფთავების ოპერატორის (`delete`) განსხვავებულ გამოყენებას მასივის და ერთი ცვლადის შემთხვევაში. ასევე რეკომენდირებულია წაშლის შემდეგ მიმთითებელს მივანიჭოთ `NULL`

ისევე როგორც C შემთხვევაში აქაც გვაქვს მექანიზმი შევამოწმოთ შესრულდა თუ არა ალოკაცია.

კოდი 1.22 მასივის ალოკაცია

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   int *ptr = NULL;
6
7   // Request memory for integer variable
8   // using new operator
9   ptr = new int(10);
10  if (!ptr) {
11   cout << "allocation of memory failed";
12   exit(0);
13  }
```

```
14
15  cout << "Value of *p: " << *ptr << endl;
16
17  // Free the value once it is used
18  delete ptr;
19  ptr=NULL;
20
21  // Allocate an array
22  ptr = new int[3];
23  ptr[2] = 11;
24  ptr[1] = 22;
25  ptr[0] = 33;
26  cout << "Array: ";
27  for (int i = 0; i < 3; i++)
28  cout << ptr[i] << " ";
29
30  // Deallocate when done
31  delete[] ptr;
32  ptr=NULL;
33  return 0;
34 }
```

კლასის ობიექტის შექმნა

კოდი 1.23 კლასის დინამიური ობიექტი

```
1 #include <iostream>
2
3 using namespace std;
4
```

```
5  class Person {
6
7  public:
8
9  string name;
10
11 int age;
12
13 // Default Constructor
14
15 Person() : name("Unknown"), age(0) {
16
17  cout << "Default constructor called" << endl;
18
19 }
20
21 // Parameterized Constructor
22
23 Person(string n, int a) : name(n), age(a) {
24
25  cout << "Parameterized constructor called for " << name << endl;
26
27 }
28
29 // Destructor
30
31 ~Person() {
32
33  cout << "Destructor called for " << name << endl;
```

```
34
35  }
36
37  void display() const {
38
39   cout << "Name: " << name << ", Age: " << age << endl;
40
41  }
42
43 };
44
45 int main() {
46
47  // Using new operator to create a single Person object
48
49  Person* singlePerson = new Person("Giorgi", 23);
50
51  singlePerson->display();
52
53  // Deallocating the single Person object
54
55  delete singlePerson;
56
57  // Using new operator to create an array of Person objects
58
59  int numPersons = 3;
60
61  Person* personArray = new Person[numPersons];
62
```

```
63  // Manually initializing each element in the array
64
65  personArray[0] = Person("tariel", 23);
66
67  personArray[1] = Person("fridon", 28);
68
69  personArray[2] = Person("avtandil", 22);
70
71  // Accessing the array of Person objects
72
73  for (int i = 0; i < numPersons; ++i) {
74
75   personArray[i].display();
76
77  }
78
79  // Deallocating the array of Person objects
80
81  delete[] personArray;
82  personArray=NULL;
83  return 0;
84
85 }
```

1.25 Timer კლასი

კოდის მუშაობის ეფექტურობის შესაფასებლად ერთერთი კრიტერიუმია თუ რა დრო დასჭირდა კოდს ამოცანის ამოსახსნელად. პროგრამირების სხვადასხვა ინტეგრირებულ გარემოებს აქვთ ამ დროის გაზომვის სხვადასხვა მექანიზმები, მაგრამ ამ მექანიზმების გარეშეც შეგვიძლია დავადგინოთ რა დრო გაატარა პროგრამამ ამა თუ იმ ოპერაციის(ან ოპერაციების/ფუნქციების) შესასრულებლად. ამისათვის გამოიძახება დროის ბიბლიოთეკიდან ფუნქცია ერთხელ ოპერაციის დაწყებისას, მეორედ დამთავრებისას. დროის ამ ორ მო-

მენტს შორის სხვაობა იქნება დრო რაც დასჭირდა კოდს შესასრულებლად. აქ არის ორი პრობლემა. პირველი, ცხადია ვგულისხმობთ რომ სხვა პროცესები არ აჩერებენ ჩვენი პროგრამის მუშაობას. ცხადია თუ პროგრამის მუშაობისას არის სხვა პროცესები რომელზეც გადაირთვება პროცესორი, მაშინ ზემოთმოყვანილი მეთოდით გამოთვლილი დრო არ იქნება ის დრო რაც გამოსაკვლევი ფუნქციის/კოდის შესრულებას დასჭირდა. მარტივი შემთხვევებისთვის ზემოთ მოყვანილი მეთოდი მშვენივრად მუშაობს. მეორე პრობლემა არის ის რომ სხვადასხვა ოპერატიულ სისტემაში დროის აღების ფუნქციები/ბიბლიოთეკა სხვადასხვაა.

ქვემოთ მოყვანილია კროსპლატფორმული თაიმერის კლასი(არქივი codes/Timer.zip).

კოდი 1.24. Timer.h დეკლარირება

```
1  ///////////////////////////////////////////////////////////////////////////
2  // Timer.h
3  // =======
4  // High Resolution Timer.
5  // This timer is able to measure the elapsed time with 1 micro-second accuracy
6  // using chrono STL (C++11) in both Windows, Linux and Unix system
7  //
8  // AUTHOR: Song Ho Ahn (song.ahn@gmail.com)
9  // CREATED: 2003-01-13
10 // UPDATED: 2024-04-17
11 //
12 // Copyright (c) 2003 Song Ho Ahn
13 ///////////////////////////////////////////////////////////////////////////
14
15 #ifndef TIMER_H_DEF
16 #define TIMER_H_DEF
17
18 #include <chrono>
19
20
21 class Timer
```

```
22 {
23 public:
24     Timer(); // default constructor
25     ~Timer(); // default destructor
26 
27     void start(); // start timer
28     void stop(); // stop the timer
29     double getElapsedTime(); // get elapsed time in second
30     double getElapsedTimeInSec(); // get elapsed time in second (same as getElapsedTime)
31     double getElapsedTimeInMilliSec(); // get elapsed time in milli-second (10^-3)
32     double getElapsedTimeInMicroSec(); // get elapsed time in micro-second (10^-6)
33     double getElapsedTimeInNanoSec(); // get elapsed time in nano-second (10^-9)
34 
35 
36 protected:
37 
38 
39 private:
40     std::chrono::high_resolution_clock::time_point startPoint;
41     std::chrono::high_resolution_clock::time_point endPoint;
42     int stopped; // stop flag
43 };
44 
45 #endif // TIMER_H_DEF
```

იმპლემენტაციის ფაილი

კოდი 1.25. ფაილი Timer.cpp

```cpp
///////////////////////////////////////////////////////////////////////////////
// Timer.cpp
// =========
// High Resolution Timer.
// This timer is able to measure the elapsed time with 1 micro-second accuracy
// using chrono STL (C++11) in both Windows, Linux and Unix system
//
// AUTHOR: Song Ho Ahn (song.ahn@gmail.com)
// CREATED: 2003-01-13
// UPDATED: 2024-04-17
//
// Copyright (c) 2003 Song Ho Ahn
///////////////////////////////////////////////////////////////////////////////

#include "Timer.h"

////////////////////////////////////////////////////////////////////////////////
// constructor
////////////////////////////////////////////////////////////////////////////////
Timer::Timer()
{
    startPoint = std::chrono::high_resolution_clock::now();
    endPoint = startPoint;
    stopped = 0;
}


```

```
29 ////////////////////////////////////////////////////////////////////////////
30 // distructor
31 ////////////////////////////////////////////////////////////////////////////
32 Timer::~Timer()
33 {
34 }
35
36
37
38 ////////////////////////////////////////////////////////////////////////////
39 // start timer.
40 // startPoint will be set at this point.
41 ////////////////////////////////////////////////////////////////////////////
42 void Timer::start()
43 {
44     stopped = 0; // reset stop flag
45     startPoint = std::chrono::high_resolution_clock::now();
46 }
47
48
49
50 ////////////////////////////////////////////////////////////////////////////
51 // stop the timer.
52 // endPoint will be set at this point.
53 ////////////////////////////////////////////////////////////////////////////
54 void Timer::stop()
55 {
56     stopped = 1; // set timer stopped flag
57     endPoint = std::chrono::high_resolution_clock::now();
```

```
58 }
59
60
61
62 ///////////////////////////////////////////////////////////////////////////
63 // compute elapsed time in nano-second resolution.
64 // other getElapsedTime will call this first, then convert to correspond resolution
65 ///////////////////////////////////////////////////////////////////////////
66 double Timer::getElapsedTimeInNanoSec()
67 {
68     if(!stopped)
69         endPoint = std::chrono::high_resolution_clock::now();
70
71     // time interval (nanosec) as double
72     return std::chrono::duration<double, std::nano>{endPoint - startPoint}.count();
73 }
74
75
76
77 ///////////////////////////////////////////////////////////////////////////
78 // divide elapsedTimeInNanoSec by 1000
79 ///////////////////////////////////////////////////////////////////////////
80 double Timer::getElapsedTimeInMicroSec()
81 {
82     return this->getElapsedTimeInNanoSec() * 0.001;
83 }
84
85
86
```

```
87 ////////////////////////////////////////////////////////////////////////////
88 // divide elapsedTimeInNanoSec by 1000000
89 ////////////////////////////////////////////////////////////////////////////
90 double Timer::getElapsedTimeInMilliSec()
91 {
92     return this->getElapsedTimeInNanoSec() * 0.000001;
93 }
94
95
96
97 ////////////////////////////////////////////////////////////////////////////
98 // divide elapsedTimeInNanoSec by 1000000000
99 ////////////////////////////////////////////////////////////////////////////
100 double Timer::getElapsedTimeInSec()
101 {
102     return this->getElapsedTimeInNanoSec() * 0.000000001;
103 }
104
105
106
107 ////////////////////////////////////////////////////////////////////////////
108 // same as getElapsedTimeInSec()
109 ////////////////////////////////////////////////////////////////////////////
110 double Timer::getElapsedTime()
111 {
112     return this->getElapsedTimeInSec();
113 }
```

გამოყენების მაგალითს ვნახავთ შემდეგ ამოცანაში.

1.26 3D მასივი

გავარჩიოთ 3D მასივის რამდენიმე ვარიანტი (იხ. არქივი codes/Array3D.zip).) გვაქვს სამი ფუნქცია

1. array3Dtest C სტილის დინამიური მასივი.
2. array3DLineartest std::vector ერთ განზომილებიანი მასივი სადაც ჩალაგებულია array3Dtest ფუნქციაში გამოყენებული 3D მასივი. მასივის 3D მასივის (i,j,k) ელემენტზე მიმართვა ხდება ერთ განზომილებიანი მასივის შესაბამისი ინდექსის გამოთვლით.
3. array3DLinearOneIndextest იგივე std::vector მასივი სადაც ჩალაგებულია 3D მასივი. მასივის ელემენტზე მიმართვა ხდება ერთი ინდექსით და ამ ინდექსის საშუალებით ხდება 3D მასივის (i,j,k) ინდექსების გამოთვლა.

კოდში გამოყენებულია Timer კლასი თითოეული ფუნქციის მუშაობის დროის შესაფასებლად. array3DLineartest ფუნქციაში გამოყენებულია OpenMP პარალელიზაცია.

კოდი 1.26. პროექტი Array3D main.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <omp.h>
4  #include "Timer.h"
5  using namespace std;
6  void array3Dtest(int nx, int ny, int nz);
7  void array3DLineartest(int nx, int ny, int nz);
8  void array3DLinearOneIndextest(int nx, int ny, int nz);
9
10 int main()
11 {
12 int numthtreads=0;
13    #pragma omp parallel
14    numthtreads=omp_get_num_threads();
15
16    printf("\n Number of threads=%d \n",numthtreads);
17    omp_set_num_threads(numthtreads-1);
```

```
18
19 int nx = 200, ny =100, nz = 2100;
20 Timer tm;
21 tm.start();
22 array3Dtest(nx,ny,nz);
23 tm.stop();
24 cout<<tm.getElapsedTimeInMilliSec();
25 cout<<endl;
26 tm.start();
27 array3DLineartest(nx,ny,nz);
28 tm.stop();
29 cout<<tm.getElapsedTimeInMilliSec();
30 cout<<endl;
31 tm.start();
32 array3DLinearOneIndextest(nx,ny,nz);
33 tm.stop();
34 cout<<tm.getElapsedTimeInMilliSec();
35 cout<<endl;
36
37 return 0;
38
39
40 }
41 void array3Dtest(int nx, int ny, int nz)
42 {
43 // Array Iterators
44 int i, j, k;
45 // Allocate 3D Array
46 int ***array3D = new int**[nx];
```

```
47
48 for(i = 0; i < nx; i++)
49 {
50  array3D[i] = new int*[ny];
51
52  for(j = 0; j < ny; j++)
53  {
54   array3D[i][j] = new int[nz];
55  }
56 }
57
58 // Access array elements
59 for(i = 0; i < nx; i++)
60 {
61  for(j = 0; j < ny; j++)
62  {
63   for(k = 0; k < nz; k++)
64   {
65    array3D[i][j][k] = (i * ny * nz) + (j * nz) + k;
66
67   }
68  }
69
70 }
71
72 // Deallocate 3D array
73 for(i = 0; i < nx; i++)
74 {
75  for(j = 0; j < ny; j++)
```

```
76   {
77    delete[] array3D[i][j];
78   }
79
80   delete[] array3D[i];
81  }
82  delete[] array3D;
83 }
84 void array3DLineartest(int nx, int ny, int nz)
85 {
86  // Array Iterators
87  int i, j, k;
88  vector<int>array1D;
89  array1D.resize(nx*ny*nz,0);
90  const int nyz0=ny * nz;
91  bool notprinted=true;
92  // Access array elements
93
94 #pragma omp parallel for num_threads(4) private(i,j,k) shared(array1D,nx,ny,nz,nyz0,notprinted)
95   for(i = 0; i < nx; i++)
96   {
97
98   const int iSlice=(i * nyz0);
99
100  for(j = 0; j < ny; j++)
101  {
102
103   //static int jCol=(j * nz);
104   static int rPtr=iSlice+j * nz;//jCol;
```

```
105   for(k = 0; k < nz; k++)
106   {
107    //const int index=iSlice + jCol + k;
108    //const int ind2=(i * ny * nz) + (j * nz) + k;
109    const int ind3=rPtr+k;
110
111    array1D[ind3]=(i * nyz0) + (j * nz) + k;
112    //array1D[rPtr+k]=(i * nyz) + (j * nz) + k;
113    //array1D[(i * nyz) + (j * nz) + k]=array1D[index];
114    //array1D[ind3]=(i * nyz) + (j * nz) + k;
115    int tid = omp_get_thread_num();
116     /* Only master thread does this */
117    if(tid == 0 && notprinted)
118    {
119     int nthreads = omp_get_num_threads();
120     printf("Number of threads = %d\n", nthreads);
121     notprinted=false;
122    }
123
124   }
125  }
126 }
127
128 }
129 void array3DLinearOneIndextest(int nx, int ny, int nz)
130 {
131  // Array Iterators
132  int index;
133  const int size=nx*ny*nz;
```

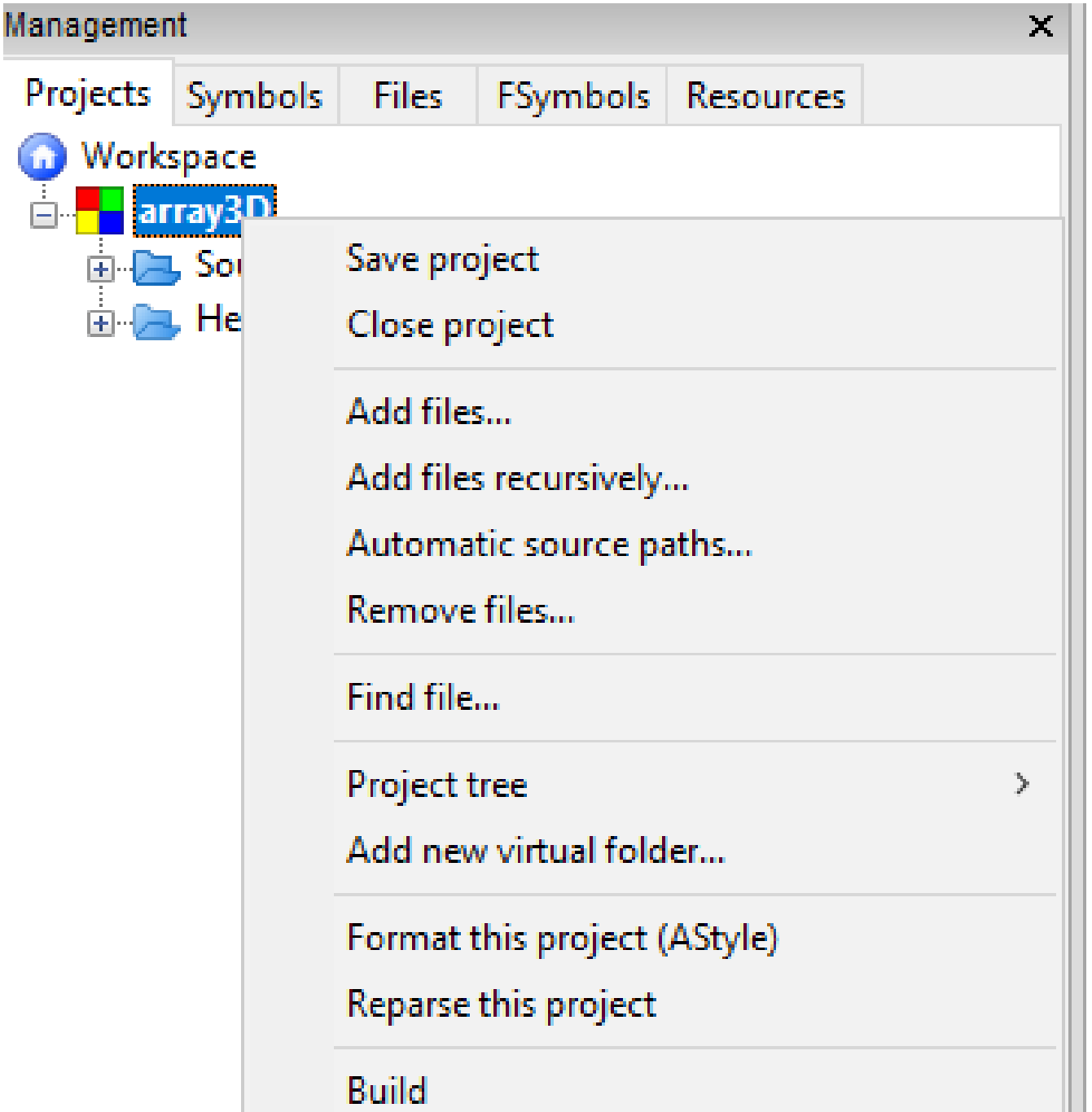
```
134 vector<int>array1D;
135 const int dept=ny * nz;
136 array1D.resize(nx*ny*nz,0);
137 // Access array elements
138 #pragma omp parallel for num_threads(4) private(index)
139  for(index = 0; index < size; index++)
140 {
141  const int i=index/dept;
142  const int _ny=ny;
143  const int _nz=nz;
144  const int iSlice=i*dept;
145  const int jCol=index%(dept);
146  const int j=jCol/_nz;
147  const int rPtr=iSlice+j*_nz;
148  const int k=jCol-j*_nz;
149  const int tmpind=rPtr+k;
150 
151  array1D[tmpind]=(i * _ny * _nz) + (j * _nz) + k;
152  }
153 }
```

თუ (i,j,k) ინდექსების გამოთვლა საჭირო არაა (მაგ. ერთ მასივს ვაკოპირებთ მეორე მასივში, ან თითოეულ ელემენტზე ვატარებთ რაღაც ოპერაციას, რაც ინდექსებზე არაა დამოკიდებული) მაშინ მესამე ფუნქციაში თუ ამოვიღებთ ინდექსების გამოთვლას ეს ფუნქცია ყველაზე სწრაფად იმუშავებს.

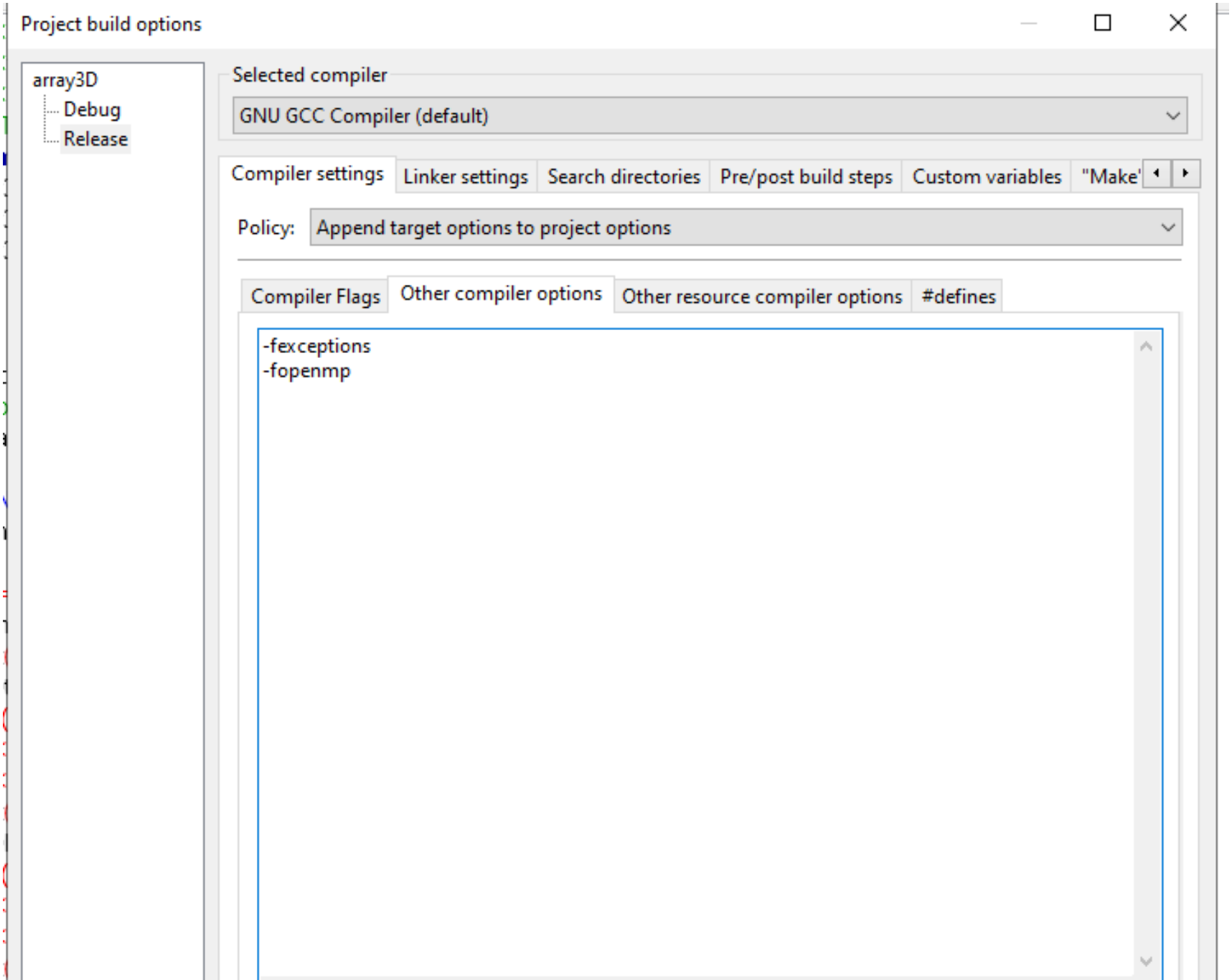
მოცემულ მაგალითში მეორე ფუნქცია OpenMP ინსტრუქციების და ფუნქციების გარეშეც კი უფრო სწრაფად იმუშავებს ვიდრე პირველი. OpenMP დაპარალელება საგრძნობია დიდი მასივების შემთხვევაში.

1.27 CodeBlocks გარემოში OpenMP ჩართვა და პროგრამის ოპტიმიზაცია

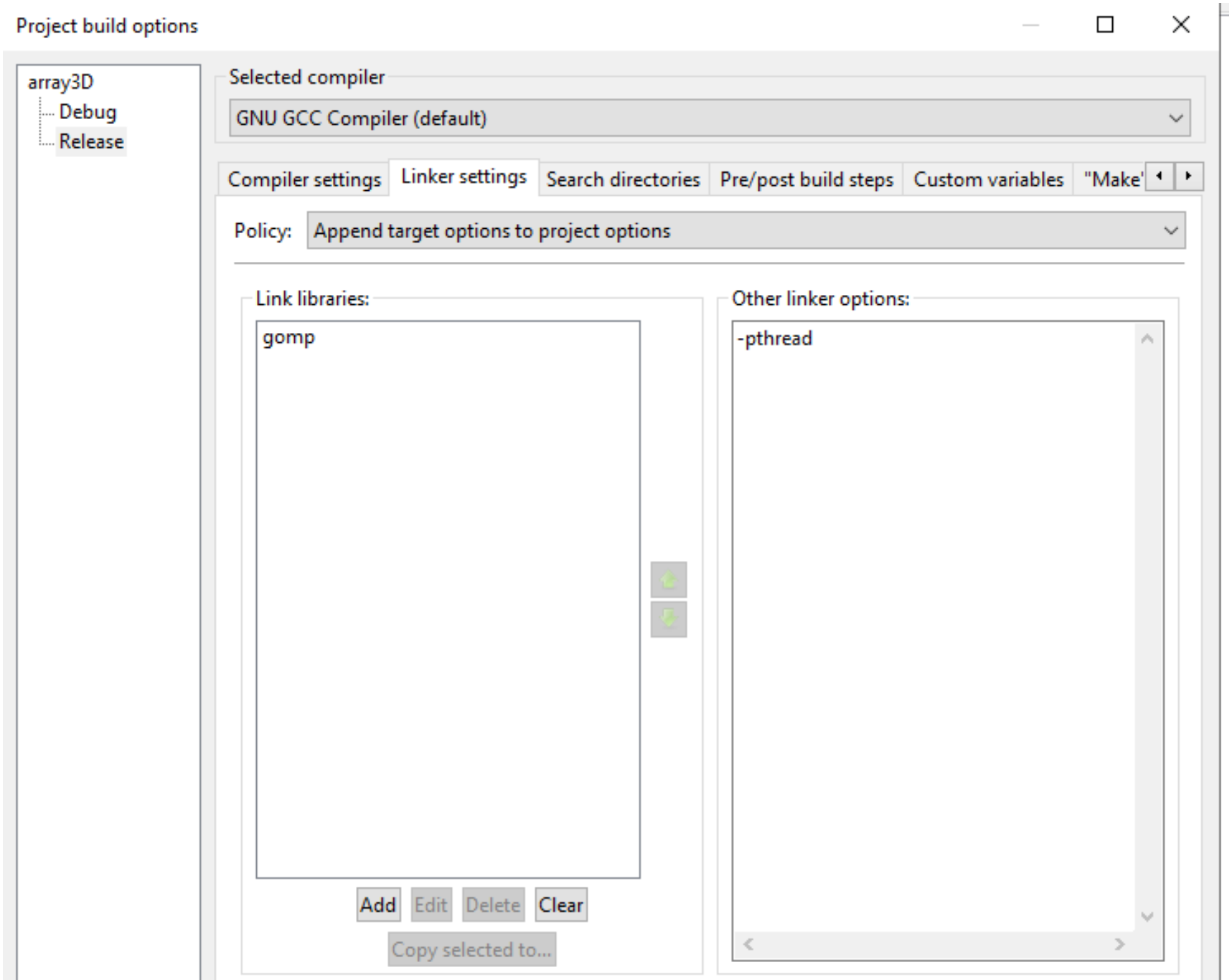
დააწაკუნეთ მაუსით პროექტზე შემდეგ დააჭირეთ მაუსის მარჯვენა ღილაკს. გამოჩნდება ქვემოთ მოყვანილი სურ.1.1. გახსენით კომპილაციის პარამეტრები(„Build options . . . “)

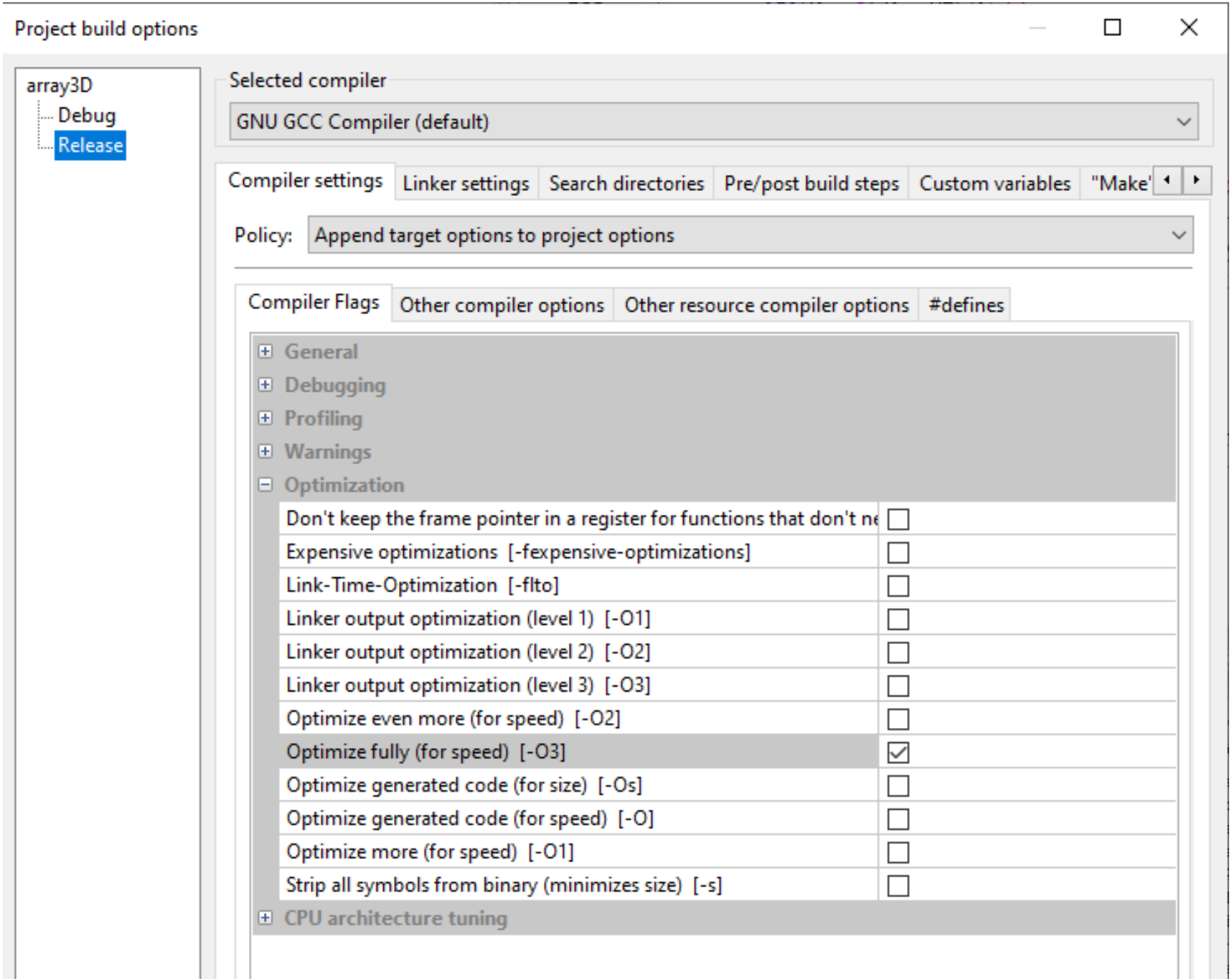

Management
Projects
Symbols
Files
FSymbols
Resources
Workspace
array3D
Save project
Close project
Add files...
Add files recursively...
Automatic source paths...
Remove files...
Find file...
Project tree
Add new virtual folder...
Format this project (AStyle)
Reparse this project
Build

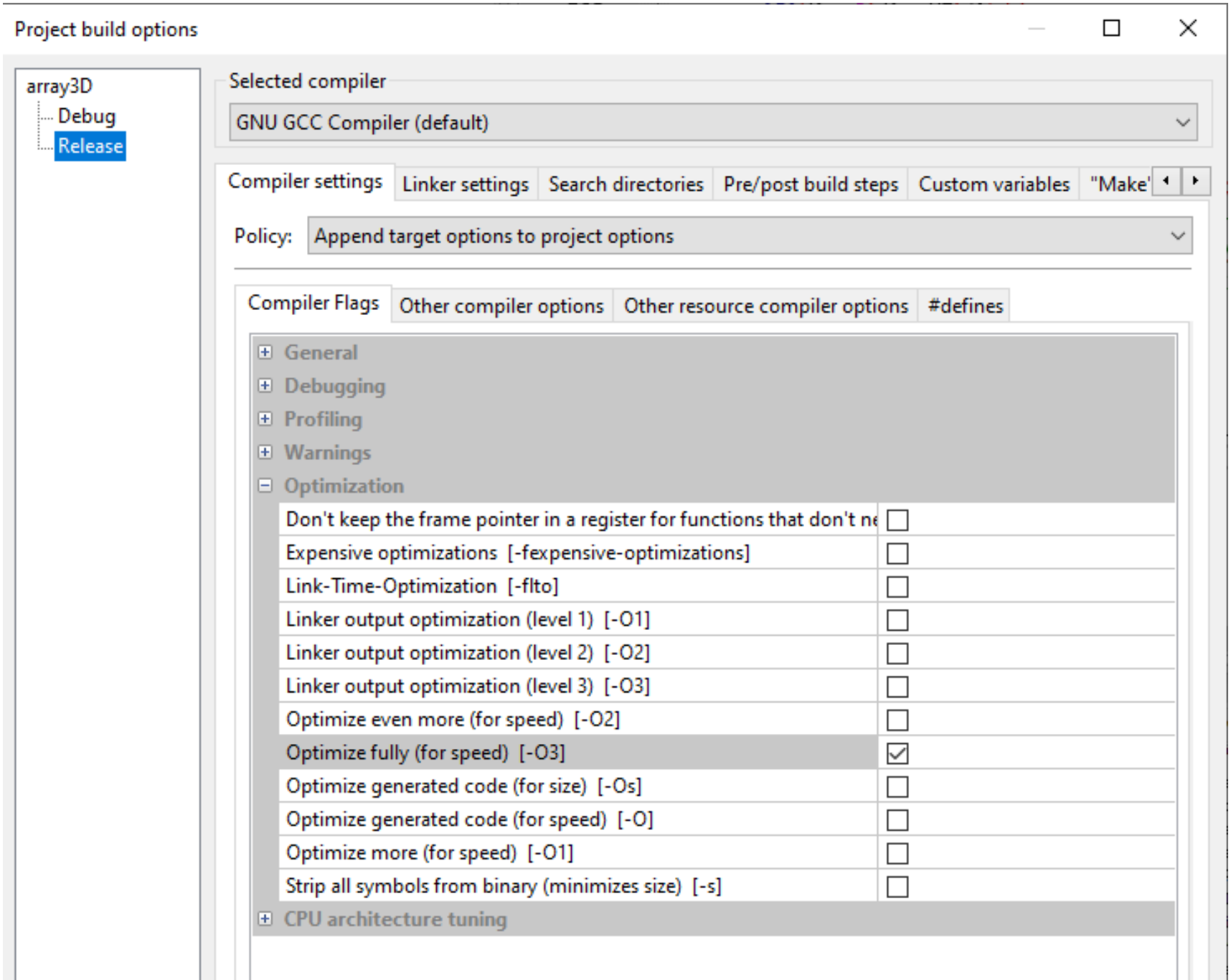
შემდეგ კომპილაციის სხვა პარამეტრების(„Other compiler options “) დაფაზე მიუთითეთ როგორც ქვემოთ სურ.1.2


Project build options
array3D
Debug
Release
Selected compiler
GNU GCC Compiler (default)
Compiler settings
Linker settings
Search directories
Pre/post build steps
Custom variables
"Make'
Policy:
Append target options to project options
Compiler Flags
Other compiler options
Other resource compiler options
#defines
-fexceptions
-fopenmp

შემდეგ კი „Linker Settings“ დაფაზე ვუთითებთ gomp ბიბლიოთეკებში და ასევე -pthread მარჯვენა ნაწილში ნაკადების ბიბლიოთეკის მისაერთებლად, როგორც ეს ნაჩვენებია ქვემოთ სურ.1.3. კომპილატორის დაფაზე ვურთავთ „-O3“ ოპტიმიზაციას. სურ.1.4 პროცესორის მოდელიდან გამომდინარე შეგვიძლია ჩავურთოთ შესაბამისი ოპტიმიზაციაც. სურ.1.5


Project build options
array3D
Debug
Release
Selected compiler
GNU GCC Compiler (default)
Compiler settings
Linker settings
Search directories
Pre/post build steps
Custom variables
"Make"
Policy:
Append target options to project options
Link libraries:
gomp
Add
Edit
Delete
Clear
Copy selected to...
Other linker options:
-pthread


Project build options
array3D
Debug
Release
Selected compiler
GNU GCC Compiler (default)
Compiler settings
Linker settings
Search directories
Pre/post build steps
Custom variables
"Make'
Policy:
Append target options to project options
Compiler Flags
Other compiler options
Other resource compiler options
#defines
General
Debugging
Profiling
Warnings
Optimization
Don't keep the frame pointer in a register for functions that don't ne
Expensive optimizations [-fexpensive-optimizations]
Link-Time-Optimization [-flto]
Linker output optimization (level 1) [-O1]
Linker output optimization (level 2) [-O2]
Linker output optimization (level 3) [-O3]
Optimize even more (for speed) [-O2]
Optimize fully (for speed) [-O3]
Optimize generated code (for size) [-Os]
Optimize generated code (for speed) [-O]
Optimize more (for speed) [-O1]
Strip all symbols from binary (minimizes size) [-s]
CPU architecture tuning


Project build options
array3D
Debug
Release
Selected compiler
GNU GCC Compiler (default)
Compiler settings
Linker settings
Search directories
Pre/post build steps
Custom variables
"Make'
Policy:
Append target options to project options
Compiler Flags
Other compiler options
Other resource compiler options
#defines
General
Debugging
Profiling
Warnings
Optimization
Don't keep the frame pointer in a register for functions that don't ne
Expensive optimizations [-fexpensive-optimizations]
Link-Time-Optimization [-flto]
Linker output optimization (level 1) [-O1]
Linker output optimization (level 2) [-O2]
Linker output optimization (level 3) [-O3]
Optimize even more (for speed) [-O2]
Optimize fully (for speed) [-O3]
Optimize generated code (for size) [-Os]
Optimize generated code (for speed) [-O]
Optimize more (for speed) [-O1]
Strip all symbols from binary (minimizes size) [-s]
CPU architecture tuning

თავი 2

წრფივი ალგებრის ელემენტები

2.1 მატრიცა

რიცხვების ერთობლიობას $\mathcal{K}$ ველიდან (როგორც წესი ეს შეიძლება იყოს ნამდვილი ან კომპლექსური რიცხვების ველი) წარმოდგენილს მართკუთხა ცხრილის სახით m სტრიქონით და n სვეტით, ეწოდება $m \times n$ მატრიცა და აღვნიშნავთ როგორც

$$\mathbf{A}^{m\times n} = \begin{array}{c} \text{სვ.} \quad 1 \quad \cdots \quad k \quad \cdots \quad n \quad \text{სტრ.} \\ \begin{pmatrix} a^1{}_1 & \cdots & a^1{}_k & \cdots & a^1{}_n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a^i{}_1 & \cdots & a^i{}_1 & \cdots & a^i{}_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_n \end{pmatrix} \begin{array}{l} 1 \\ \cdot \\ i \\ \cdot \\ m \end{array} \end{array} \tag{2.1}$$

ანუ უფრო მარტივად, მატრიცას ავღნიშნავთ მუქი დიდი ასო ნიშნით($\mathbf{A}$), ხოლო მატრიცის ელემენტს — a^i_j სადაც $i = \overline{1,m}$,$j = \overline{1,n}$.

ხშირად ზედა ინდექსს წერენ ქვემოთ და გვაქვს a_{ij}. ამ შემთხვევაში პირველი ინდექსი ნომრავს სტრიქონს, ხოლო მეორე სვეტს. ტენზორულ აღრიცხვაში მრავალი ინდექსის შემთხვევაში თანმიმდევრობის სწორად აღსანიშნავად იყენებენ ცარიელ ადგილს $a^i_{\ j}$ ან

წერტილს $a^{i}_{\cdot j}$.

„$\mathbf{A}$ მატრიცის ელემენტი i,j“ ავღნიშნავთ როგორც $\{\mathbf{A}\}^i_j = a^i_j$.

„$\mathbf{A}$ მატრიცა ელემენტებით a^i_j “ ავღნიშნავთ როგორც $[a^i_j]$.

მატრიცას $\mathbf{B}^{m\times n} = [b^i_j]$ ეწოდება $\mathbf{A}^{n\times m} = [a^j_i]$ მატრიცის ტრანსპორირებული თუ $a^j_i = b^i_j$; $i = \overline{1,m}, j = \overline{1,n}$. ტრანსპორირებული მატრიცის აღსანიშნავად გამოვიყენებთ აღნიშვნას $\mathbf{A}^{\mathrm{T}}$,$\mathbf{A}^{\mathrm{t}}$.

ორი მატრიცისთვის გამრავლების ოპერაცია განმარტებულია, როცა პირველი მატრიცის სვეტების რაოდენობა მეორე მატრიცის სტრიქონების რაოდენობის ტოლია:

$$\mathbf{A}^{m\times p}\cdot\mathbf{B}^{p\times n} = \mathbf{C}^{m\times n}$$

და ნიშნავს შემდეგს:

$$\begin{pmatrix} a^1_1 & \cdots & a^1_2 & \cdots & a^1_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i_1 & \cdots & a^i_k & \cdots & a^i_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m_1 & \cdots & a^m_k & \cdots & a^m_p \end{pmatrix} \cdot \begin{pmatrix} b^1_1 & \cdots & b^1_j & \cdots & b^1_n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ b^k_1 & \cdots & b^k_j & \cdots & b^k_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b^p_1 & \cdots & b^p_j & \cdots & b^p_n \end{pmatrix} = \begin{pmatrix} c^1_1 & \cdots & c^1_j & \cdots & c^1_n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ c^i_1 & \cdots & c^i_j & \cdots & c^i_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c^m_1 & \cdots & c^m_j & \cdots & c^m_n \end{pmatrix} \tag{2.2}$$

სადაც მიღებული მატრიცის ელემენტები გამოითვლება როგორც

$$c^i_j = \sum_{k=1}^{p} a^i_k \cdot b^k_j \tag{2.3}$$

მატრიცული გამრავლების აღნიშვნაში „$\cdot$“ ნიშანს გამოვტოვებთ და ვიგულისხმებთ რომ ჩანაწერი $\mathbf{AB}$ აღნიშნავს $\mathbf{A}\cdot\mathbf{B}$— ზემოთ განმარტებულ მატრიცულ ნამრავლს. ზოგიერთ ლიტერატურაში მატრიცული ნამრავლისთვის გამოიყენებენ ,,$\times$“ ნიშანს.

$i = j$ პოზიციების სიმრავლეს მატრიცაში ეწოდება მთავარი დიაგონალი. კვადრატულ მატრიცას(სვეტების და სტრიქონების რაოდენობა ტოლია) მთავარ დიაგონალზე ერთიანებით და სხვა ელემენტებით ნული, ეწოდება ერთეულოვანი მატრიცა და აღინიშნება როგორც $\mathbf{E}$ ან $\mathbf{I}$

$$\mathbf{I} \equiv \mathbf{E} = \mathbf{E}_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} = [\delta^i_j]_{n\times n} = [\delta^i_j] \tag{2.4}$$

δ^i_j კრონეკერის სიმბოლო ეწოდება. სხვადასხვა შემთხვევებში ასევე შეგვიძლია შეგვხვდეს როგორც δ_{ij} ან δ^{ij}. δ^i_j გააჩნია ე.წ. ინდექსის ფილტრაციის თვისება $a_j = \sum_{i=1}^n \delta^i_j a_i$, $a^i = \sum_{j=1}^n \delta^{ij} a^j$ და $a_i = \sum_{j=1}^n \delta_{ij} a_j$ ანუ კრონეკერის სიმბოლოზე გამრავლება ერთი ინდექსის ავჯამვით ჯამის ინდექსს ცვლის კრონეკერის სიმბოლოს მეორე, თავისუფალი ინდექსით.

ინდექსს რის მიხედვითაც აიღება ჯამი მუნჯი ინდექსი ეწოდება. ხშირად ჯამის ნიშანს არ წერენ და განმეორებითი ინდექსის შემთხვევაში იგულისხმება ჯამი (აინშტაინის შეთანხმება). ანუ ზემოთ მოყვანილი ტოლობების შეგვიძლია ჩავწეროთ ჯამის ნიშნის გარეშე: $a_i = \delta_{ij} a_j$. განტოლება (2.3) შეგვიძლია დავწეროთ როგორც

$$c^i_j = a^i_k \cdot b^k_j, \quad k = \overline{1,p} \ ^1 \qquad . \tag{2.5}$$

თუ $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, $\mathbf{A}$-ს სიმეტრიული მატრიცა ეწოდება, ხოლო როცა $\mathbf{A}^{\mathrm{T}} = -\mathbf{A}$ — ანტისიმეტრიული.

მატრიცების გამრავლების ოპერაციის თვისებები:

1. $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$ ასოციატურობა
2. $(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC}+\mathbf{BC}$ და $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$
3. ნებისმიერი $x \in \mathcal{K}$ გვაქვს $x(\mathbf{AB}) = (x\mathbf{A})\mathbf{B} = \mathbf{A}(x\mathbf{B})$
4. $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$
5. $(\mathbf{A}^{\mathrm{T}})^{-1} = (\mathbf{A}^{-1})^{\mathrm{T}}$

2.2 ვექტორ სვეტი და ვექტორ სტრიქონი

ერთ სვეტიანი მატრიცის შემთხვევაში გამოვიყენებთ აღნიშვნას

$$\mathbf{A}^{m\times 1} \equiv \vec{\boldsymbol{a}} = \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} = [a^i]^m = [a^i] \tag{2.6}$$

რაც ნიშნავს, რომ ვექტორი მატრიცულ აღნიშვნებში წარმოადგენენ სვეტ მატრიცას. $\vec{\boldsymbol{a}}$ ვექტორში ვგულისხმობთ ფორმალურ ვექტორს — რაღაც ობიექტები რომლებიც შეგვიძლია ჩავწეროთ (გავაერთიანოთ) ერთ სვეტში. ეს შეიძლება იყოს რიცხვები, გეომეტრიული ვექტორის კომპონენტები, ფუნქციები, ოპერატორები (მაგ. სხვადასხვა რიგის წარმოებულები რაიმე ცვლადით)

[1] განმეორებული ინდექსით იგულისხმება ჯამი

ერთ სტრიქონიანი მატრიცის შემთხვევაში გამოვიყენებთ აღნიშვნას

$$\mathbf{A}^{1\times n} \equiv (\mathbf{A}^{n\times 1})^{\mathrm{T}} \equiv \vec{\boldsymbol{a}}^{\mathrm{T}} = (a_1 a_2 \cdots a_n) = [a_i^{\mathrm{t}}]_n = [a_i^{\mathrm{t}}] \tag{2.7}$$

ტრანსპორირების ოპერაციის გამოყენებით შეგვიძლია სტრიქონ მატრიცა წარმოვადგინოთ როგორც ტრანსპორირებული სვეტ მატრიცა და პირიქით.

$$[a_i]^{\mathrm{T}} = [a^i] \quad [a_i] = [a^i]^{\mathrm{T}} \tag{2.8}$$

ან ვექტორული სახით

$$\vec{\boldsymbol{a}} \equiv [a^i] \quad \vec{\boldsymbol{a}}^{\mathrm{T}} \equiv [a_i^{\mathrm{t}}] \equiv \vec{\boldsymbol{a}}^{\mathrm{t}} \tag{2.9}$$

ტრანსპორირებულ სვეტს(ანუ სტრიქონს) ფორმალურად ავღნიშნავთ როგორც $[a^i]^{\mathrm{T}} \equiv [a^{i\mathrm{t}}]$. t ზედა ინდექსით ავღნიშნავთ რომ ობიექტი არის სტრიქონ ჩანაწერი.

2.3 სკალარული და პირდაპირი ნამრავლი

დავუშვათ მოცემულია მატრიცა სვეტი $[a^i]$ და მატრიცა სტრიქონი $[b_i]$. მაშინ

$$[a^i][b_i] = \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} (b_1 b_2 \cdots b_p) = \begin{pmatrix} a^1 b_1 & a^1 b_2 & \cdots a^1 b_p \\ a^2 b_1 & a^2 b_2 & \cdots a^2 b_p \\ \vdots & \vdots & \vdots \\ a^m b_1 & a^m b_2 & \cdots a^m b_p \end{pmatrix} \tag{2.10}$$

და

$$[b_i][a^i] = (b_1 b_2 \cdots b_m) \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} = b_1 a^1 + b_2 a^2 + \cdots + b_m a^m = \sum_{j=1}^{m} b_j a^j \tag{2.11}$$

შემოტანილი აღნიშვნების თანახმად $[a^j]$ სვეტ მატრიცაა, ხოლო $[b_j]$ სტრიქონ მატრიცა. ამიტომ (2.10) და (2.11) ჩანაწერები არავითარ გაუგებრობას არ იწვევს. აღნიშვნები ნათლად აჩვენებს რომ $[a^j]$ და $[b_j]$ ურთიერთ ტრანსპორირებულია და ორივე შემთხვევაში ჩანაწერიდან ნათლად ჩანს რა იქნება შედეგი.

როგორც აღვნიშნეთ ზოგიერთ ლიტერატურაში გამოყენებულია მხოლოდ ქვედა ინდექსები(როგორც წესი ესაა წრფივი ალგებრის და ანალიზური გეომეტრიის, ვექტორული და ტენზორული აღრიცხვის სახელმძღვანელოების ის სექციები სადაც საუბარია ევკლიდის სივრცეზე და გამოყენებულია დეკარტის საკოორდინატო სისტემა).

თუ გვსურს ჩავწეროთ განტ. (2.11) მაშინ ვწერთ $[a^j]^{\mathrm{t}}[b^j]$. ანუ ვგულისხმობთ რომ გვაქვს სვეტ მატრიცები და (2.11) ჩანაწერი ნიშნავს, რომ ტრანსპორირებული სვეტ მატრიცა(რომელიც უკვე სტრიქონ მატრიცაა ტრანსპორირების შემდეგ) მრავლდება სვეტ მატრიცაზე და შედეგად გვაძლევს რიცხვს.

თუ გვსურს ჩავწეროთ განტ. (2.10) მაშინ ვწერთ $[a^j][b^j]^{\mathrm{t}}$. ანუ გვაქვს სვეტის[1] ნამრავლი სტრიქონზე და შედეგად მიიღება მატრიცა (2.10).

$\vec{\boldsymbol{a}} \otimes \vec{\boldsymbol{b}} \equiv [a^j][b^j]^{\mathrm{T}}$ აღნიშნავს ე.წ. "პირდაპირ ნამრავლს"(ტენზორული აღრიცხვის ტერმინია დიადური ნამრავლი, კრონეკერის ნამრავლი)[1].

თუ გვაქვს ორი სტრიქონ მატრიცა ამ შემთხვევაშიც (2.10) და (2.11) ოპერაციებისათვის საჭიროა ტრანსპორირება, რომ მატრიცული გამრავლების წესმა(სტრიქონი მრავლდება სვეტზე), რაც ყოველთვის დაცული უნდა იყოს, მოგვცეს ის შედეგები რაც გვაქვს (2.10) და (2.11) განტოლებებში.

განტ. (2.11) ბოლო ჯამში არ აქვს მნიშვნელობა b_j დავწერთ პირველს თუ a^j, ვინაიდან b_j და a^j რიცხვებია, მაგრამ მატრიცულ ნამრავლში მამრავლების თანმიმდევრობას აქვს მნიშვნელობა. როგორც განხილული მაგალითი აჩვენებს შედეგი შეიძლება იყოს რიცხვი ან მატრიცა. როცა მატრიცულ ნამრავლში მამრავლების გადანაცვლება შედეგს არ ცვლის ამბობენ, რომ მატრიცები კომუტირებენ.

ზემოთმოყვანილ სვეტ და სტრიქონ მატრიცის აღნიშვნებში და მატრიცების გამრავლების წესის გამოყენებით ორი ვექტორის სკალარული ნამრავლი მატრიცულ აღნიშვნებში ჩაიწერება როგორც[2]

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = \vec{\boldsymbol{a}}^{\mathrm{T}} \vec{\boldsymbol{b}} = [a_i][b^i] = a_i \cdot b^i \tag{2.12}$$

რომ არ გამოგვეყენებინა მატრიცული ჩაწერა, ან, თუ შევთანხმდებოდით, მატრიცაში ორივე ინდექსი ქვემოთაა ან ორივე ზემოთაა. შეგვეძლო დაგვეწერა

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = a^i \cdot b^i = a_i \cdot b_i \tag{2.13}$$

[1] ანუ ისევ ვუშვებთ რომ გვაქვს სვეტ მატრიცები და სვეტის ტრანსპორირება გვაძლევს სტრიქონს

[1] როგორც ვხედავთ სვეტ მატრიცის აღსანიშნავად შემოვიტანეთ დაბალი დახრილი მუქი ასო ნიშანი. ხშირად სვეტ მატრიცას ვექტორს უწოდებენ(რაც ზოგადად სწორი არაა)

[2] გამეორებული ინდექსით იგულისხმება ჯამი.

სვეტ და სტრიქონების ჩანაწერებისთვის ინდექსების განსხვავებულ პოზიციებზე (ზემოთ და ქვემოთ) წერა მოსახერხებელია და ბევრ შეცდომას გვარიდებს, ჩვენს შემთხვევაში (ვსაუბრობთ რიცხვებზე რომლებიც ჩაწერილია მატრიცაში და სხვა არაფერი) $a_i \cdot b^i$ ჯამში ინდექსები ზემოთ იქნება თუ ქვემოთ არ აქვს მნიშვნელობა, მაგრამ როცა შემოვიტანთ ბაზისის ცნებას და ვექტორის და ბაზისის გარდაქმნებს, ვნახავთ, რომ ზედა/ქვედა ინდექსები მოსახერხებელია. ამას გარდა მოსახერხებელია სვეტების და სტრიქონების, სვეტ მატრიცის და სტრიქონ მატრიცის კომპონენტების, სვეტ ვექტორის და სტრიქონ ვექტორს შორის განსხვავების აღსანიშნავად, რასაც ასევე ქვემოთ ვნახავთ. ახლა მხოლოდ ავღნიშნავთ რომ განტ. (2.12)-ში

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = \vec{\boldsymbol{a}}^{\mathrm{T}} \vec{\boldsymbol{b}} \tag{2.14}$$

ყოველთვის სამართლიანია და იძლევა სწორ პასუხს. განტოლება

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = [a_i][b^i] = a_i \cdot b^i$$

ასევე ყოველთვის სამართლიანია და იძლევა სწორ პასუხს. მაგრამ

$$[a_i] = [a^i]^{\mathrm{T}}$$

ვექტორის კომპონენტებისთვის სამართლიანია მხოლოდ დეკარტეს საკოორდინატო სისტემაში. ტრანსპორირება სვეტ მატრიცას აქცევს სტრიქონ მატრიცად, მაგრამ უნდა გვახსოვდეს, რომ ოპერაცია სახელად „ინდექსის ჩამოწევა“ არაა ტრანსპორირების შედეგი. ამიტომ ტრანსპორირებული სვეტის შემთხვევაში(ანუ მივიღეთ სტრიქონი) ავღნიშნავთ $[a^{i\mathrm{t}}]$.

ეს ნიშნავს შემდეგს:

- ტრანსპორირების ოპერაციას ავღნიშნავთ როგორც „T“
- სტრიქონ ჩანაწერს ავღნიშნავთ როგორც „t“

მატრიცის ერთ-ერთი საინტერესო მახასიათებელი არის ე.წ. "კვალი"(trace, spoor) $\mathrm{tr}(\mathbf{A}) = A^i_i$ — დიაგონალური ელემენტების ჯამი. გვაქვს სკალარული ნამრავლის განზოგადება მატრიცისთვის. მატრიცისთვის სკალარული ნამრავლი განმარტებულია როგორც

$$\mathrm{tr}(\mathbf{A}^{\mathrm{T}}\mathbf{B}) = \mathrm{tr}(\mathbf{C}) = c^i{}_i = a^i{}_j b^j{}_i. \tag{2.15}$$

სადაც $a^i{}_j$, $b^i{}_j$, $c^i{}_j$ აღნიშნავს $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ მატრიცების ელემენტებს, ხოლო განმეორებული ინდექსებით ვგულისხმობთ ჯამს.

ამ განმარტების თანახმად თუ შევადარებთ (2.10) და (2.11), ვნახავთ, რომ (2.10)-ს დიაგონალური ელემენტების ჯამი მართლაც არის (2.11).

2.4 მატრიცის გამრავლების სხვადასხვა წარმოდგენა

განვიხილოთ მატრიცების გამრავლების წესი

$$c^i_j = \sum_{k=1}^{p} a^i_k \cdot b^k_j = \vec{\boldsymbol{a}}^{i\mathrm{t}} \vec{\boldsymbol{b}}_j = \vec{\boldsymbol{a}}^i \cdot \vec{\boldsymbol{b}}_j\,, \quad i = \overline{1,m}; j = \overline{1,n} \tag{2.16}$$

ბოლო ჩანაწერი ტოლობაში ნიშნავს რომ „i“ სტრიქონზე ჩაწერილი ვექტორი (რომელიც სტრიქონია, რადგან ინდექსი ზემოთაა და ნომრავს სტრიქონს. სვეტების ინდექსია k, რომლიც „ჩამალულია“ ვექტორის აღნიშვნაში) სკალარულად (შიდა ნამრავლი/წერტილოვანი ნამრავლი) მრავლდება „j“ ნომერ სვეტ ვექტორზე(ინდექსი ქვემოთაა, რაც აღნიშნავს სვეტს მეორე მატრიცაში). ეს ინდექსები ნომრავენ არა ვექტორის კომპონენტებს არამედ თავად ვექტორებს. ვექტორის კომპონენტების ინდექსები არ ჩანს ვინაიდან ვექტორული ჩანაწერია. კომპონენტებს ნომრავს ჯამის ინდექსი „k“.

> **!** განტ. (2.16) ბოლო ორი ტოლობა არაა სტანდარტული აღნიშვნა. სახელმძღვანელოებში ნახავთ $\mathrm{row_i}(\mathbf{A}) \cdot \mathrm{col_j}(\mathbf{B})$

მატრიცული ნამრავლის სხვადასხვა წარმოდგენების განხილვამდე განვიხილოთ მატრიცების ნამრავლის ორი კერძო შემთხვევა:

1. მატრიცის გამრავლება სვეტზე.
2. სტრიქონის გამრავლება მატრიცაზე.

2.5 მატრიცის ნამრავლი სვეტზე

მატრიცის ნამრავლი სვეტზე არის სვეტი და გამოითვლება როგორც

$$\begin{pmatrix} c^1 \\ c^2 \\ \vdots \\ c^i \\ \vdots \\ c^m \end{pmatrix} = \begin{pmatrix} a^1{}_1 & \cdots & a^1{}_2 & \cdots & a^1{}_p \\ a^2{}_1 & \cdots & a^2{}_2 & \cdots & a^2{}_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i{}_1 & \cdots & a^i{}_k & \cdots & a^i{}_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_p \end{pmatrix} \cdot \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^k \\ \vdots \\ b^p \end{pmatrix} = \begin{pmatrix} a^1{}_1 \cdot b^1 + \cdots + a^1{}_k \cdot b^k + \cdots + a^1{}_p \cdot b^p \\ a^2{}_1 \cdot b^1 + \cdots + a^2{}_k \cdot b^k + \cdots + a^2{}_p \cdot b^p \\ \vdots \\ a^i{}_1 \cdot b^1 + \cdots + a^i{}_k \cdot b^k + \cdots + a^i{}_p \cdot b^p \\ \vdots \\ a^m{}_1 \cdot b^1 + \cdots + a^m{}_k \cdot b^k + \cdots + a^m{}_p \cdot b^p \end{pmatrix} \tag{2.17}$$

$$\begin{pmatrix} c^1 \\ c^2 \\ \vdots \\ c^i \\ \vdots \\ c^m \end{pmatrix} = \begin{pmatrix} a^1{}_1 \\ a^2{}_1 \\ \vdots \\ a^i{}_1 \\ \vdots \\ a^m{}_1 \end{pmatrix} \cdot b^1 + \cdots + \begin{pmatrix} a^1{}_k \\ a^2{}_k \\ \vdots \\ a^i{}_k \\ \vdots \\ a^m{}_k \end{pmatrix} \cdot b^k + \cdots + \begin{pmatrix} a^1{}_p \\ a^2{}_p \\ \vdots \\ a^i{}_p \\ \vdots \\ a^m{}_p \end{pmatrix} \cdot b^p \tag{2.18}$$

$$\boxed{\vec{\boldsymbol{c}} = \mathbf{A} \cdot \vec{\boldsymbol{b}} = \begin{pmatrix} | & | & | \\ \vec{\boldsymbol{a}}_1 \cdots & \vec{\boldsymbol{a}}_k & \cdots \vec{\boldsymbol{a}}_p \\ | & | & | \end{pmatrix} \cdot \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^k \\ \vdots \\ b^p \end{pmatrix} = \vec{\boldsymbol{a}}_1 \cdot b^1 + \cdots + \vec{\boldsymbol{a}}_k \cdot b^k + \cdots + \vec{\boldsymbol{a}}_p \cdot b^p = \vec{\boldsymbol{a}}_i b^i} \tag{2.19}$$

ცხადია თუ გამოვიყენებთ ზოგად წესს (სტრიქონი მრავლდება სვეტზე) მაშინ შედეგი ჩაიწერება როგორც(შეადარეთ გან. (2.12))

$$\boxed{\vec{\boldsymbol{c}} = \mathbf{A} \cdot \vec{\boldsymbol{b}} = \begin{pmatrix} -\vec{\boldsymbol{a}}^{1\mathrm{t}}- \\ -\vec{\boldsymbol{a}}^{2\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{m\mathrm{t}}- \end{pmatrix} \cdot \vec{\boldsymbol{b}} = \begin{pmatrix} \vec{\boldsymbol{a}}^{1\mathrm{t}} \cdot \vec{\boldsymbol{b}} \\ \vec{\boldsymbol{a}}^{2\mathrm{t}} \cdot \vec{\boldsymbol{b}} \\ \vdots \\ \vec{\boldsymbol{a}}^{m\mathrm{t}} \cdot \vec{\boldsymbol{b}} \end{pmatrix}} \tag{2.20}$$

ან უფრო კომპაქტურად

$$\boxed{c^i = \vec{\boldsymbol{a}}^{i\mathrm{t}} \cdot \vec{\boldsymbol{b}}} \quad \text{ცხადია იგივეა რაც 2.19} \tag{2.21}$$

2.6 სტრიქონის გამრავლება მატრიცაზე.

სტრიქონის ნამრავლი მატრიცაზე არის სტრიქონი და გამოითვლება როგორც

$$\left(\begin{array}{ccccccc} c_1 & c_2 & \cdots & c_k & \cdots & c_p \end{array}\right) = \left(\begin{array}{ccccccc} b_1 & b_2 & \cdots & b_i & \cdots & b_m \end{array}\right) \cdot \left(\begin{array}{ccccc} a^1{}_1 & \cdots & a^1{}_2 & \cdots & a^1{}_p \\ a^2{}_1 & \cdots & a^2{}_2 & \cdots & a^2{}_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i{}_1 & \cdots & a^i{}_k & \cdots & a^i{}_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_p \end{array}\right) \tag{2.22}$$

ადგილის დაზოგვის მიზნით ჩავწეროთ ტოლობა ყოველი c_i-სათვის

$$c_i = \sum_{k=1}^{m} b_k \cdot a_i^k$$

ეს ჯამი გვაქვს თითოეული c_i-სათვის. თუ გამოვიყენებთ მატრიცების შეკრების წესს. გავშლით მიღებულ ჯამს და დავაჯგუფებთ b_i წევრებს მივიღებთ

$$\left(\begin{array}{ccccccc} c_1 & \cdots & c_k & \cdots & c_p \end{array}\right) = b_1 \cdot \left(\begin{array}{ccccccc} a_1^1 & \cdots & a_k^1 & \cdots & a_p^1 \end{array}\right) + \cdots + b_m \cdot \left(\begin{array}{ccccccc} a_1^m & \cdots & a_k^m & \cdots & a_p^m \end{array}\right) \tag{2.23}$$

ან ზემოთ მოყვანილ განტოლებებს თუ ჩავწერთ სტრიქონ ვექტორების გამოყენებით

$$\vec{\boldsymbol{c}}^{\,\mathrm{t}} = \vec{\boldsymbol{b}}^{\,\mathrm{t}} \cdot \left(\begin{array}{c} -\vec{\boldsymbol{a}}^{1\mathrm{t}}- \\ -\vec{\boldsymbol{a}}^{2\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{i\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{m\mathrm{t}}- \end{array}\right) \tag{2.24}$$

და

$$\boxed{\vec{c}^{\,\mathrm{t}} = b_k \vec{a}^{\,k\mathrm{t}}} \tag{2.25}$$

შეგახსენებთ რომ t აღნიშნავს რომ გვაქვს სტრიქონ ვექტორები.

2.7 მატრიცული ნამრავლის სხვადასხვა წარმოდგენა

ზემოთ აღწერილი შემთხვევების განხილვის შემდეგ მატრიცული ნამრავლი (ე.წ. შიდა ნამრავლი) შეგვიძლია განვიხილოთ ხუთი სხვადასხვა გზით:

1. მატრიცული ნამრავლის განმარტების თანახმად ესაა სტრიქონ ვექტორების სკალარული ნამრავლი სვეტ ვექტორებზე(მატრიცების გამრავლების წესის თანახმად $\vec{a}$-ს ზომა ტოლი უნდა იყოს $\vec{b}$-ს ზომის.

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} -\vec{a}^{\,1\mathrm{t}}- \\ -\vec{a}^{\,2\mathrm{t}}- \\ \vdots \\ -\vec{a}^{\,m\mathrm{t}}- \end{pmatrix} \begin{pmatrix} | & | & & | \\ \vec{b}_1 & \vec{b}_2 & \dots & \vec{b}_n \\ | & | & & | \end{pmatrix} =$$

$$= \begin{pmatrix} \vec{a}^{\,1\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{\,1\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{\,1\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{\,1\mathrm{t}} \cdot \vec{b}_n \\ \vec{a}^{\,2\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{\,2\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{\,2\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{\,2\mathrm{t}} \cdot \vec{b}_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{a}^{\,m\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{\,m\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{\,m\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{\,m\mathrm{t}} \cdot \vec{b}_n \end{pmatrix} \tag{2.26}$$

2. მატრიცული ნამრავლი როგორც მატრიცის ნამრავლი სვეტზე

$$\mathbf{C} = \mathbf{AB} = \mathbf{A} \begin{pmatrix} | & & | \\ \vec{b}_1 & \dots & \vec{b}_n \\ | & & | \end{pmatrix} = \begin{pmatrix} | & & | \\ \mathbf{A}\vec{b}_1 & \dots & \mathbf{A}\vec{b}_n \\ | & & | \end{pmatrix} \tag{2.27}$$

აქედან გამომდინარეობს, რომ $\mathbf{C}$ მატრიცის i სვეტი წარმოადგენს $\mathbf{A}$ მატრიცის და $\mathbf{B}$-მატრიცის i სვეტის ნამრავლს.

$$\{\mathbf{C}\}_i \equiv \vec{\boldsymbol{c}}_i = \mathbf{A}\,\vec{\boldsymbol{b}}_i \tag{2.28}$$

3. მატრიცული ნამრავლი როგორც სტრიქონის ნამრავლი მატრიცაზე

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} -\vec{\boldsymbol{a}}^{1\mathrm{t}}- \\ -\vec{\boldsymbol{a}}^{2\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{m\mathrm{t}}- \end{pmatrix} \mathbf{B} = \begin{pmatrix} -\vec{\boldsymbol{a}}^{1\mathrm{t}}\mathbf{B}- \\ -\vec{\boldsymbol{a}}^{2\mathrm{t}}\mathbf{B}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{m\mathrm{t}}\mathbf{B}- \end{pmatrix} \tag{2.29}$$

$$\vec{\boldsymbol{c}}^{\,i\mathrm{t}} = \vec{\boldsymbol{a}}^{\,i\mathrm{t}}\mathbf{B} \tag{2.30}$$

4. მატრიცული ნამრავლი როგორც სვეტის და სტრიქონის პირდაპირი ნამრავლების ჯამი იხ. განტ (2.10)

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} | & & | \\ \vec{\boldsymbol{a}}_1 & \dots & \vec{\boldsymbol{a}}_p \\ | & & | \end{pmatrix} \begin{pmatrix} -\vec{\boldsymbol{b}}^{1\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{b}}^{p\mathrm{t}}- \end{pmatrix} = \left(\sum_i \left(\vec{\boldsymbol{a}}_i \otimes \vec{\boldsymbol{b}}^{\,i\mathrm{t}} \right) \right) \tag{2.31}$$

5. მატრიცული ნამრავლი, როგორც ბლოკებად დანაწევრებული მატრიცების ნამრავლი

$$\begin{aligned} \mathbf{C} = \mathbf{AB} &= \left(\begin{array}{c|c} \mathbf{A}_1 & \mathbf{A}_2 \\ \hline \mathbf{A}_3 & \mathbf{A}_4 \end{array} \right) \times \left(\begin{array}{c|c} \mathbf{B}_1 & \mathbf{B}_2 \\ \hline \mathbf{B}_3 & \mathbf{B}_4 \end{array} \right) \\ &= \left(\begin{array}{c|c} \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_3 & \mathbf{A}_1\mathbf{B}_2 + \mathbf{A}_2\mathbf{B}_4 \\ \hline \mathbf{A}_3\mathbf{B}_1 + \mathbf{A}_4\mathbf{B}_3 & \mathbf{A}_3\mathbf{B}_2 + \mathbf{A}_4\mathbf{B}_3 \end{array} \right) \end{aligned} \tag{2.32}$$

ხშირად გვჭირდება ასეთი მატრიცების ნამრავლი და შებრუნებული

$$\begin{aligned}\mathbf{C} = \mathbf{AB} &= \left(\begin{array}{c|c} \mathbf{C}_1 & \vec{\boldsymbol{d}}_1 \\ \hline 0 & 1 \end{array}\right) \times \left(\begin{array}{c|c} \mathbf{C}_2 & \vec{\boldsymbol{d}}_2 \\ \hline 0 & 1 \end{array}\right) \\ &= \left(\begin{array}{c|c} \mathbf{C}_1\mathbf{C}_2 & \mathbf{C}_1\vec{\boldsymbol{d}}_2 + \vec{\boldsymbol{d}}_1 \\ \hline 0 & 1 \end{array}\right)\end{aligned} \tag{2.33}$$

და

$$\mathbf{M} = \left(\begin{array}{c|c} \mathbf{C} & \vec{\boldsymbol{d}} \\ \hline 0 & 1 \end{array}\right)^{-1} = \left(\begin{array}{c|c} \mathbf{C}^{-1} & -\mathbf{C}^{-1}\vec{\boldsymbol{d}} \\ \hline 0 & 1 \end{array}\right) \tag{2.34}$$

თავი 3

წრფივი, ევკლიდური და აფინური სივრცეები

ყველა ადამიანი, თავის ბუნებით,
მიისწრაფვის ცოდნისკენ

არისტოტელე ,,მეტაფიზიკა“

ქვემოთ გავიხსენებთ წრფივი სივრცის თეორიის განმარტებებს და თეორემებს(დამტკიცების გარეშე), შემდეგ შემოვიტანთ ევკლიდური და აფინური სივრცის განმარტებას (ჰ. ვეილი) და ბოლოს განვიხილავთ როგორ არის აფინური სივრცის გარდაქმნები რეალიზებული **OpenGL** სისტემაში. ცხადია ეს არ იქნება წრფივი სივრცის და აფინური გეომეტრიის სრული მიმოხილვა. შევეხებით მხოლოდ იმ საკითხებს, რასაც უშუალო კავშირი აქვს ჩვენს კურსთან.

3.1 წრფივი სივრცე

წრფივი(ან *ვექტორული*)*სივრცე* ეწოდება $\mathcal{L}$ სიმრავლეს რომელზეც ამ სიმრავლის ორი ელემენტისათვის განმარტებული გვაქვს შეკრების ოპერაცია „+“ და ამ სიმრავლის ელემენტის გამრავლება რიცხვზე $\mathcal{K}$ ველიდან ($\mathcal{K}$ ველის ელემენტებს ეწოდებათ სკალარები. როგორც წესი არის ნამდვილი $\mathbb{R}$ ან კომპლექსური რიცხვების $\mathbb{C}$ ველი ანუ გვაქვს ნადვილი ან კომპლექსური რიცხვები(სკალარები)). და ეს ოპერაციები აკმაყოფილებენ პირობებს:

განმარტება 3.1: წრფივი სივრცე

1. $\vec{a} + \vec{b} = \vec{b} + \vec{a}$
2. $(\vec{a} + \vec{b}) + \vec{c} = \vec{a} + (\vec{b} + \vec{c})$
3. არსებობს ელემენტი $\mathbf{0} \in \mathcal{L}$ ისეთი, რომ $\vec{a} + \mathbf{0} = \vec{a}\ \forall\, \boldsymbol{a} \in \mathcal{L}$.
4. ნებისმიერი $\boldsymbol{a} \in \mathcal{L}$ არსებობს ისეთი ელემენტი $-\boldsymbol{a} \in \mathcal{L}$, რომ $\vec{a} + (-\vec{a}) = \mathbf{0}$
5. $\alpha \cdot (\vec{a} + \vec{b}) = \alpha \cdot \vec{a} + \alpha \cdot \vec{b}\ \forall\, \vec{a}$ და $\vec{b} \in \mathcal{L}$ და $\alpha \in \mathcal{K}$
6. $(\alpha + \beta) \cdot \vec{a} = \alpha \cdot \vec{a} + \beta \cdot \vec{a}\ \forall\, \vec{a} \in \mathcal{L}$ და $\alpha\ ,\ \beta \in \mathcal{K}$
7. $\alpha \cdot (\beta \cdot \vec{a}) = (\alpha\beta) \cdot \vec{a}\ \forall\, \vec{a} \in \mathcal{L}$ და $\alpha,\ \beta \in \mathcal{K}$
8. $1 \cdot \vec{a} = \vec{a}\ \forall\, \vec{a} \in \mathcal{L}$

წრფივი სივრცის ელემენტებს ეწოდებათ *ვექტორები*[1]. თვისებები 1)—4) ასევე ნიშნავს, რომ $\mathcal{K}$ წარმოადგენს აბელურ(კომუტატიურ) ჯგუფს შეკრების ოპერაციის მიმართ. $\mathbf{0}$ ელემენტს ეწოდება ნულ ვექტორი. მოცემულ წრფივ სივრცეში არსებობს მხოლოდ ერთი ნულ ვექტორი.

„$-\vec{a}$“ ეწოდება $\vec{a}$-ს შებრუნებული ელემენტი. ყოველ ელემნტისთვის არსებობს მხოლოდ ერთი შებრუნებული ელემენტი. თვისებები 5)—8) ნიშნავს, რომ ველი $\mathcal{K}$ წრფივად მოქმედებს $\mathcal{L}$-ზე. 1)—8) აქსიომების გამოყენებით შეიძლება დამტკიცდეს, რომ

1. $0\vec{a} = \alpha\mathbf{0} = \mathbf{0}$ ნებისმიერი $\vec{a} \in \mathcal{L}$, $\alpha \in \mathcal{K}$
2. $(-1)\vec{a} = -\vec{a}$ ნებისმიერი $\vec{a} \in \mathcal{L}$
3. თუ $\alpha\vec{a} = \mathbf{0}$, მაშინ $\alpha = 0$, ან $\vec{a} = \mathbf{0}$

მოცემული წრფივი სივრციდან შეგვიძლია ავიღოთ ვექტორები $\vec{v}_i \in \mathcal{L}$ და რიცხვები $\lambda_i \neq 0 \in \mathcal{K}$ და შევადგინოთ წრფივი კომ-

[1]ეს შეგვიძლია ჩავთვალოთ ვექტორის მეორე განმარტებად. პირველი განმარტება მოცემული იყო მატრიცების განხილვისას. ეს იყო ე.წ. "კოორდინატული ვექტორი"

ბინაცია $\sum_i \lambda_i \vec{\boldsymbol{v}}_i$. თუ $\sum_i \lambda_i \vec{\boldsymbol{v}}_i = \mathbf{0}$ მხოლოდ და მხოლოდ მაშინ როცა ყველა $\lambda_i = 0$, მაშინ $\vec{\boldsymbol{v}}_i$ ვექტორები ერთმანეთისგან წრფივად დამოუკიდებელია (ანუ რომელიმე მათგანი არ წარმოადგენს დანარჩენების წრფივ კომბინაციას)

წრფივად დამოუკიდებელი ვექტორების სისტემას $\vec{\boldsymbol{e}}_i$ ეწოდება $\mathcal{L}$ სივრცის ბაზისი, თუ ნებისმიერი $\vec{\boldsymbol{v}} \in \mathcal{L}$ შეიძლება წარმოდგენილ იქნას როგორც $\vec{\boldsymbol{e}}_i$ ვექტორების წრფივი კომბინაცია $\vec{\boldsymbol{v}} = \sum_i \lambda_i \vec{\boldsymbol{e}}_i$

სივრცეს ეწოდება სასრულ განზომილებიანი თუ საბაზისო ვექტორების რაოდენობა სასრულია. წინააღმდეგ შემთხვევაში სივრცის განზომილება უსასრულოა.

სასრულ განზომილებიან სივრცეში ნებისმიერი ბაზისის ელემენტების რაოდენობა ერთი და იგივეა და სივრცის განზომილების ტოლია.

თუ $\vec{\boldsymbol{e}}_i$ წარმოადგენს ბაზისს $\mathcal{L}$ სივრცეში მაშინ ნებისმიერი $\vec{\boldsymbol{v}} \in \mathcal{L}$-სათვის გაშლა

$$\vec{\boldsymbol{v}} = \sum_i \vec{\boldsymbol{e}}_i v_i \tag{3.1}$$

ერთადერთია. v_i ეწოდება $\vec{\boldsymbol{v}}$ ვექტორის კოორდინატები $\vec{\boldsymbol{e}}_i$ ბაზისში. ზემოთ მოყვანილი ტოლობა მატრიცული სახით შეგვიძლია ჩავწეროთ როგორც

$$\vec{\boldsymbol{v}} = \begin{pmatrix} | & & | & & | \\ \vec{\boldsymbol{e}}_1 & \cdots & \vec{\boldsymbol{e}}_k & \cdots & \vec{\boldsymbol{e}}_n \\ | & & | & & | \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ \cdot \\ v_k \\ \cdot \\ v_n \end{pmatrix} \tag{3.2}$$

ჩვენ საქმე გვექნება მხოლოდ ნამდვილ რიცხვთა ველზე სასრულ განზომილებიან $\mathbb{R}^2$,$\mathbb{R}^3$ სივრცეებთან[1].

$\vec{\boldsymbol{x}}$ ვექტორის კოორდინატებს ავღნიშნავთ x^i და გაშლაში $\vec{\boldsymbol{x}} = \sum_i \vec{\boldsymbol{e}}_i x^i$ ჯამის ნიშანს გამოვტოვებთ და განმეორებული ინდექსით ვიგულისხმებთ ჯამს, როგორც ეს გვქონდა მატრიცების განხილვისას.

მატრიცულ აღნიშვნებში ზემოთ მოყვანილი ფორმულა შეგვიძლია ჩავწეროთ როგორც

$$\vec{\boldsymbol{v}} = [\vec{\boldsymbol{e}}]^t [x] \tag{3.3}$$

$[\vec{\boldsymbol{e}}]^t$ აღნიშნავს რომ საბაზისო ვექტორებისთვის გვაქვს სტრიქონ ჩანაწერი, ხოლო $[x]$ ნიშნავს, რომ კოორდინატებისთვის გვაქვს სხვეტ ჩანაწერი. ანუ მაგ. სამი განზომილების შემთხვევაში $[x]$ აღნიშნავს კოორდინატების სვეტ მატრიცას $[x] \equiv \begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix}$

[1] უსასრულო განზომილებიანი სივრცეების მაგალითია მაგალითად სიგნალის გაშლა ფურიე მწკრივად ან ინტეგრალად სიგნალების თეორიაში ან ჰილბერტის სივრცე კვანტურ მექანიკაში და ველის თეორიაში.

თითოეული საბაზისო ვექტორი წარმოადგენს სვეტ ვექტორს.

$$\begin{pmatrix} | \\ \vec{\boldsymbol{e}}_i \\ | \end{pmatrix} = \begin{pmatrix} e_i^1 \\ e_i^2 \\ e_i^3 \end{pmatrix} \tag{3.4}$$

ბაზისითვის შემოვიტანოთ მოკლე აღნიშვნა $\mathcal{B} \equiv (\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \cdots, \vec{\boldsymbol{e}}_n) \equiv [\vec{\boldsymbol{e}}]^t$. დავუშვათ $\vec{\boldsymbol{x}}$ და $\vec{\boldsymbol{y}}$ წრფივი $\mathcal{L}$სივრცის ელემენტებია და $[\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \cdots, \vec{\boldsymbol{e}}_n]^t$ წარმოადგენს ამ სივრცის ბაზისს. მაშინ

$$\vec{\boldsymbol{x}} = \mathcal{B}[x];\ \vec{\boldsymbol{y}} = \mathcal{B}[y]; \tag{3.5}$$

$$\vec{\boldsymbol{x}} = \vec{\boldsymbol{e}}_i x^i;\ \vec{\boldsymbol{y}} = \vec{\boldsymbol{e}}_i y^i; \tag{3.6}$$

$$\vec{\boldsymbol{x}} \pm \vec{\boldsymbol{y}} = \vec{\boldsymbol{e}}_i (x^i \pm y^i) \tag{3.7}$$

$$a\vec{\boldsymbol{x}} = \vec{\boldsymbol{e}}_i (a x^i) \quad a \text{ რიცხვია} \tag{3.8}$$

3.2 ასახვები წრფივ სივრცეში

წრფივი ალგებრის კურსში მტკიცდება, რომ ვექტორის ნებისმიერი წრფივი ასახვა (ერთი წრფივი სივრციდან მეორეში ან იგივე სივრცეში) შეიძლება წარმოდგენილ იქნას მატრიცის სახით, როგორც ბაზისის ასახვა. ანუ $\phi(\vec{\boldsymbol{x}}) = \phi(\vec{\boldsymbol{e}}_i) x^i$ და რადგან ასახვა წრფივია $\phi(\vec{\boldsymbol{e}}_i) \equiv \vec{\boldsymbol{f}}_i = \{\mathbf{C}\}_i^j \vec{\boldsymbol{e}}_j$ (შეგახსენებთ, რომ $\{\mathbf{C}\}_i^j \equiv c_i^j$ აღნიშნავს $\mathbf{C}$ მატრიცის (j,i) ელემენტს.)

დაწვრილებით შევჩერდეთ გამოსახულებაზე:

$$\vec{\boldsymbol{f}}_i = [\mathbf{C}]_i^j \vec{\boldsymbol{e}}_j = c_i^j \vec{\boldsymbol{e}}_j \tag{3.9}$$

კერძოდ:

- ანასახ(გარდაქმნილ) ბაზისს ავღნიშნავთ როგორც $\vec{\boldsymbol{e}}_{i'}$. ინდექსი ქვემოთაა და „′“ ნიშანი აქვს ინდექსს და არა ვექტორს.
- ანასახ(გარდაქმნილ) ვექტორის კოორდინატს ავღნიშნავთ როგორც $x^{i'}$. ინდექსი ზემოთაა და „′“ ნიშანი აქვს ინდექსს და არა ვექტორის კოორდინატს.
- ანუ ზემოთ მოყვანილი ჩანაწერის ნაცვლად დავწერთ

$$\vec{\boldsymbol{e}}_{i'} = c_{i'}^j \vec{\boldsymbol{e}}_j \tag{3.10}$$

ან მატრიცული ფორმით:

$$\mathcal{B}' = \mathcal{B}\mathbf{C}$$
$$\begin{pmatrix} | & & | & & | \\ \vec{\boldsymbol{e}}_{1'} & \cdots & \vec{\boldsymbol{e}}_{k'} & \cdots & \vec{\boldsymbol{e}}_{n'} \\ | & & | & & | \end{pmatrix} = \begin{pmatrix} | & & | & & | \\ \vec{\boldsymbol{e}}_{1} & \cdots & \vec{\boldsymbol{e}}_{k} & \cdots & \vec{\boldsymbol{e}}_{n} \\ | & & | & & | \end{pmatrix} \mathbf{C} \tag{3.11}$$

ჩვენ საქმე გვექნება გეომეტრიასთან ორ და სამ განზომილებიან სივრცეში სადაც $\mathbf{C}$ არის *კვადრატული არა გადაგვარებული მატრიცა* *(*$\det|\mathbf{C}| \neq 0$*)* , რომელსაც ბაზისის გარდაქმნის მატრიცას უწოდებენ.

$$\begin{pmatrix} | & | & | \\ \vec{\boldsymbol{e}}_{1'} & \vec{\boldsymbol{e}}_{2'} & \vec{\boldsymbol{e}}_{3'} \\ | & | & | \end{pmatrix} = \begin{pmatrix} | & | & | \\ \vec{\boldsymbol{e}}_{1} & \vec{\boldsymbol{e}}_{2} & \vec{\boldsymbol{e}}_{3} \\ | & | & | \end{pmatrix} \mathbf{C} \tag{3.12}$$

მომავალში ვერტიკალურ ხაზებს გამოვტოვებთ და ვიგულისხმებთ, რომ $\vec{\boldsymbol{e}}_i$ ვექტორები წარმოადგენენ სვეტ ვექტორებს. ბაზისი $\mathcal{B}$ კი ჩაიწერება როგორც

$$\mathcal{B} \equiv \begin{pmatrix} | & | & | \\ \vec{\boldsymbol{e}}_{1} & \vec{\boldsymbol{e}}_{2} & \vec{\boldsymbol{e}}_{3} \\ | & | & | \end{pmatrix} \equiv [\vec{e}]^{\mathrm{t}} \tag{3.13}$$

ასევე ყურადღება მიაქციეთ, რომ საბაზისო ვექტორების მასივი(ბაზისი) ჩაწერილია როგორც საბაზისო ვექტორების სტრიქონი და მრავლდება მარჯვნიდან გადასვლის მატრიცაზე. თავად თითოეული საბაზისო ვექტორი წარმოადგენს სვეტ მატრიცას (განტ. 3.4)

3.3 წრფივი ასახვა

ვთქვათ გვაქვს ორი წრფივი სივრცე $\mathcal{V}$ და $\mathcal{W}$ რაღაც $\mathcal{K}$ ველზე. ასახვას $\mathcal{A}\colon \mathcal{V} \to \mathcal{W}$ ეწოდება წრფივი, თუ $\forall \vec{\boldsymbol{a}}, \vec{\boldsymbol{b}} \in \mathcal{V}$ და სკალარისთვის $\lambda \in \mathcal{K}$ ადგილი აქვს ტოლობას:

$$\mathcal{A}(\vec{\boldsymbol{a}} + \vec{\boldsymbol{b}}) = \mathcal{A}\vec{\boldsymbol{a}} + \mathcal{A}\vec{\boldsymbol{b}}, \quad \mathcal{A}(\lambda\vec{\boldsymbol{a}}) = \lambda\mathcal{A}\vec{\boldsymbol{a}} \tag{3.14}$$

ურთიერთცალსახა(ბიექცია) წრფივ ასახვას $\mathcal{A}\colon \mathcal{V} \to \mathcal{W}$ ეწოდება იზომორფიზმი. $\mathcal{V}$ და $\mathcal{W}$ ეწოდებათ იზომორფული თუ მათ შორის არსებობს იზომორფიზმი.

$\mathcal{V}$ და $\mathcal{W}$ წრფივი სივრცეები $\mathcal{K}$ ველზე(ველი ერთი და იგივე უნდა იყოს ორივე სივრცისთვის) იზომორფულია მხოლოდ და მხოლოდ მაშინ როცა ამ სივრცეების განზომილებები ტოლია.

3.3.1 წრფივი ასახვის მატრიცა დავუშვათ გვაქვს წრფივი ასახვა $\mathcal{A}\colon \mathcal{V} \to \mathcal{W}$. და $\mathcal{B}_\mathcal{V} = [\vec{\boldsymbol{v}}]^{\mathrm{t}} = (\vec{\boldsymbol{v}}_1, \cdots, \vec{\boldsymbol{v}}_n)$ არის $\mathcal{V}$ ბაზისი, ხოლო $\mathcal{B}_\mathcal{W} = [\vec{\boldsymbol{w}}]^{\mathrm{t}} = (\vec{\boldsymbol{w}}_1, \cdots, \vec{\boldsymbol{w}}_m)$ არის $\mathcal{W}$ ბაზისი.

განმარტება 3.2: წრფივი ასახვა

წრფივი ასახვის $\mathcal{A}\colon \mathcal{V} \to \mathcal{W}$ მატრიცა $\mathcal{B}_\mathcal{V}$ და $\mathcal{B}_\mathcal{W}$ ბაზისების მიმართ ეწოდება მატრიცას

$$\mathbf{A} = \begin{pmatrix} a_1^1 & \cdots & a_n^1 \\ \vdots & \ddots & \vdots \\ a_1^m & \cdots & a_n^m \end{pmatrix} \tag{3.15}$$

განზომილებით $m \times n$ სადაც i სვეტი შედგენილია $\mathcal{A}(\vec{\boldsymbol{v}}_i)$ კოორდინატებით $\mathcal{B}_\mathcal{W} = [\vec{\boldsymbol{w}}]^{\mathrm{t}}$ ბაზისის მიმართ:

$$\mathcal{A}\vec{\boldsymbol{v}}_i = a_i^j \vec{\boldsymbol{w}}_j = [\vec{\boldsymbol{w}}]^{\mathrm{t}} \mathbf{A} \tag{3.16}$$

ვიცით რა ასახვის მატრიცა $\mathcal{A}$, შეგვიძლია ვიპოვნოთ ყოველი $\vec{\boldsymbol{x}} \in \mathcal{V}$ ანასახი $\mathcal{A}$ წრფივი გარდაქმნისას როგორც

$$\begin{gathered} \vec{\boldsymbol{x}} = \vec{\boldsymbol{v}}_j x^j \quad \vec{\boldsymbol{x}} \in \mathcal{V} \\ \vec{\boldsymbol{y}} = \vec{\boldsymbol{w}}_i y^i \quad \vec{\boldsymbol{y}} \in \mathcal{W} \\ \vec{\boldsymbol{y}} = \mathcal{A}\vec{\boldsymbol{x}} \Rightarrow y^i = a_j^i x^j \\ \begin{pmatrix} y^1 \\ \vdots \\ y^m \end{pmatrix} = \begin{pmatrix} a_1^1 & \cdots & a_n^1 \\ \vdots & \ddots & \vdots \\ a_1^m & \cdots & a_n^m \end{pmatrix} \begin{pmatrix} x^1 \\ \vdots \\ x^n \end{pmatrix} \end{gathered} \tag{3.17}$$

3.4 წრფივი ასახვის მატრიცის გარდაქმნა ბაზისის გარდაქმნისას

თეორემა 3.1: წრფივი ასახვის მატრიცის გარდაქმნა

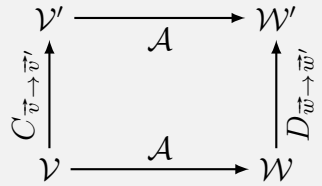
დავუშვათ გვაქვს წრფივი ასახვა $\mathcal{A}$: $\mathcal{V} \to \mathcal{W}$. $\mathcal{B}_\mathcal{V} = [\vec{v}]^{\mathrm{t}} = (\vec{v}_1, \cdots, \vec{v}_n)$ არის $\mathcal{V}$ ბაზისი, ხოლო $\mathcal{B}_\mathcal{W} = [\vec{w}]^{\mathrm{t}} = (\vec{w}_1, \cdots, \vec{w}_m)$ არის $\mathcal{W}$ ბაზისი.

და გვაქვს ბაზისის გარდაქმნები:

- $\mathcal{V} \to \mathcal{V}'$ გადასვლის მატრიცით C რაც ნიშნავს, $[\vec{v}'] = [\vec{v}]\mathbf{C}$.
- $\mathcal{W} \to \mathcal{W}'$ გადასვლის მატრიცით D რაც ნიშნავს, $[\vec{w}'] = [\vec{w}]\mathbf{D}$.

მაშინ ასახვა $\mathcal{A}$: $\mathcal{V}' \to \mathcal{W}'$ მოიცემა მატრიცით

$$\mathbf{A}' = \mathbf{D}^{-1}\mathbf{AC} \tag{3.18}$$



სურ 3.1: წრფივი გარდაქმნის მატრიცა და ბაზისის გარდაქმნა

დამტკიცება:

$$\underset{\text{განმარტების თანახმად ((3.16))}}{\mathcal{A}\vec{v}'} = \underset{\text{გადასვლის მატრიცა } \mathbf{D}_{\mathcal{W}\to\mathcal{W}'}}{[\vec{w}']^{\mathrm{t}}\mathbf{A}'} = [\vec{w}]^{\mathrm{t}}\mathbf{DA}' \tag{3.19}$$

მეორე მხრივ

$$\mathcal{A}\vec{v}' = \mathcal{A}([\vec{v}]^{\mathrm{t}}\mathbf{C}) = (\mathcal{A}[\vec{v}]^{\mathrm{t}})\mathbf{C} = [\vec{w}]^{\mathrm{t}}\mathbf{AC} \tag{3.20}$$

ბოლო ორი ტოლობის შედარება მოგვცემს

$$\mathbf{DA}' = \mathbf{AC} \tag{3.21}$$

საიდანაც მივიღებთ $\mathbf{A}' = \mathbf{D}^{-1}\mathbf{AC}$ სურ.3.1

3.5 წრფივი ოპერატორი

წრფივი ოპერატორი(გარდაქმნა) ეწოდება $\mathcal{L}$ წრფივი სივრცის წრფივ ასახვას თავის თავზე $\mathcal{A}$:$\mathcal{L} \to \mathcal{L}$. შემდგომში „ოპერატორში“ ვიგულისხმებთ წრფივ ოპერატორს.

წრფივი გარდაქმნის მატრიცის $\mathcal{A}$:$\mathcal{L} \to \mathcal{L}$ განმარტებიდან გამომდინარე, ბუნებრივია $\mathcal{L}$ სივრცის ორივე ეგზემპლარის შემთხვევაში ავიღოთ ერთი და იგივე ბაზისი $\mathcal{B}_\mathcal{L} = [\vec{e}]^{\mathrm{t}}$. მიღებულ კვადრატულ მატრიცას $\mathbf{A}$ ეწოდება $\mathcal{A}$:$\mathcal{L} \to \mathcal{L}$ წრფივი ოპერატორის მატრიცა

$\mathcal{B}_{\mathcal{L}} = [\vec{e}]^{\mathrm{t}}$ ბაზისში. რაც ნიშნავს, რომ $\mathbf{A}$ მატრიცის i სვეტი წარმოადგენს $\mathcal{A}\vec{e}_i$ ვექტორის კოორდინატებს $\mathcal{B}_{\mathcal{L}}$ ბაზისში:

$$\mathcal{A}\vec{e}_i = \vec{e}'_i = \vec{e}_j a_i^j \tag{3.22}$$

თეორემა 3.2: წრფივი ოპერატორის მატრიცის გარდაქმნა

$\mathbf{A}$ არის წრიფი გარდაქმნის ოპერატორის $\mathcal{A}{:}\mathcal{L} \to \mathcal{L}$ მატრიცა $\mathcal{B}_{\mathcal{L}} = [\vec{e}]^{\mathrm{t}}$ ბაზისში. $\mathbf{A}'$ არის მატრიცა $\mathcal{B}'_{\mathcal{L}} = [\vec{e}']^{\mathrm{t}}$ ბაზისში. $\mathbf{C}$ არის გადასვლის მატრიცა $\mathcal{B}'_{\mathcal{L}} = \mathcal{B}_{\mathcal{L}}\mathbf{C}$. მაშინ $\mathcal{A}$ ოპერატორის მატრიცა $\mathcal{B}'$ ბაზისში მოიცემა ფორმულით

$$\mathbf{A}' = \mathbf{C}^{-1}\mathbf{A}\mathbf{C} \tag{3.23}$$

დამტკიცება: ზემოთ დავამტკიცეთ მსგავსი თეორემა როცა გვქონდა ორი სხვადასხვა სივრცე. ახლა ორივე სივრცე ერთი და იგივეა. რაც ნიშნავს რომ $\mathbf{D}$ მატრიცის ნაცვლად გვაქვს $\mathbf{C}$ მატრიცა.

ან შეგვიძლია დავამტკიცოთ იგივე გზით რაც ზემოთ:

$$\mathcal{A}\mathcal{B}' = \mathcal{A}\mathcal{B}\mathbf{C} = \mathcal{B}\mathbf{A}\mathbf{C}$$

მეორე მხრივ

$$\mathcal{A}\mathcal{B}' = \mathcal{B}'\mathbf{A}' = \mathcal{B}\mathbf{C}\mathbf{A}'$$

რაც ნიშნავს

$$\mathcal{B}\mathbf{C}\mathbf{A}' = \mathcal{B}\mathbf{A}\mathbf{C}$$

რადგან ეს უნდა სრულდებოდეს ნებისმიერი ბაზისისთვის

$$\mathbf{C}\mathbf{A}' = \mathbf{A}\mathbf{C}$$

და

$$\mathbf{A}' = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}$$

მაგალითი 3.1: წრფივი გარდაქმნა ობიექტის სისტემაში

ვთქვათ მარსის მოძრაობა მზის გარშემო აღიწერება $\mathbf{A}$ მატრიცით. დედამიწის მოძრაობა მზის გარშემო $\mathbf{C}$ მატრიცით.
მაშინ, მარსის მოძრაობა დედამიწის გარშემო აღიწერება $\mathbf{A}'$ მატრიცით:

$$\mathbf{A}' = \mathbf{C}^{-1}\mathbf{A}\mathbf{C} \tag{3.24}$$

მაგალითი 3.2: წრფივი გარდაქმნა მსოფლიო სისტემაში

(3.23) განტოლებასთან ერთად ხშირად გამოიყენება

$$\mathbf{A} = \mathbf{C}\mathbf{A}'\mathbf{C}^{-1} \tag{3.25}$$

ინტერპრეტაცია არის შემდეგი: ვიცით რა გარდაქმნა მოძრავ საკოორდინატო სისტემის მიმართ, ვპოულობთ გარდაქმნას უძრავი(ლაბორატორიული, მსოფლიო) საკოორდინატო სისტემის მიმართ უძრავი სისტემიდან მოძრავ სისტემაზე გადასვლის მატრიცის საშუალებით.

ვთქვათ მთვარის მოძრაობა დედამიწის გარშემო აღიწერება $\mathbf{A}'$ მატრიცით. დედამიწის მოძრაობა მზის გარშემო $\mathbf{C}$ მატრიცით. მაშინ, მთვარის მოძრაობა მზის გარშემო აღიწერება $\mathbf{A}$ მატრიცით.

! კომპიუტერულ გრაფიკაში ხშირად გამოიყენება გან. (3.25). ადვილია ვიპოვნოთ ობიექტის გარდაქმნა ლოკალური საკოორდინატო სისტემის მიმართ. შემდეგ კი, ვიცით რა როგორ გარდაიქმნება ლოკალური საკოორდინატო სისტემა(დედამიწა) მსოფლიო(მზის) საკოორდინატო სისტემის მიმართ, ვიპოვნით ობიექტის გარდაქმნას(ობიექტის კოორდინატებს) მსოფლიო საკოორდინატო სისტემის მიმართ.

რაც შეეხება განტ. (3.23) ეს გამოიყენება ობიექტის კოორდინატების გამოსათვლელად კამერასთან დაკავშირებულ სისტემაში. $\mathbf{C}$ არის კამერის გარდაქმნის მატრიცა მსოფლიო საკოორდინატო სისტემის მიმართ. $\mathbf{A}$ არის ობიექტის გარდაქმნის მატრიცა მსოფლიო საკოორდინატო სისტემაში. განტ. (3.23) მოგვცემს ობიექტის კოორდინატებს კამერის საკოორდინატო სისტემაში(ამას OpenGL ავტომატურად ითვლის გამოსახულების ფორმირებისას, მაგრამ კოორდინატები კამერის სისტემაში, საჭიროა როცა პროგრამულად გვსურს კამერიდან დაშორების გამოთვლა. მაგ. სხივთა სვლის(Raytrasing) გამოთვლისას).

3.6 აქტიური და პასიური გარდაქმნები

განასხვავებენ ორი ტიპის გარდაქმნებს: აქტიური გარდაქმნები, როცა რაღაც(ობიექტი, ფიზიკური მდგომარეობა) იცვლება და პასიური, როცა ერთი და იგივე ობიექტს(ფიზიკურ მდგომარეობას) აღვწერთ სხვადასხვა საკოორდინატო სისტემიდან. პასიური გარდაქმნის მაგალითებია:

- გალილეის ფარდობითობის პრინციპი, როცა ობიექტს(ფიზიკურ პროცესს) ვუყურებთ მოძრავი საკოორდინატო სისტემიდან.
- ლორენცის გარდაქმნები ფარდობითობის სპეციალურ თეორიაში
- კომპიუტერულ გრაფიკაში კამერის, რის საშუალებითაც ვუყურებთ სცენას (სამყაროს), გარდაქმნები

3.6.1 ვექტორი სხვადასხვა საკოორდინატო სისტემაში ვიცით რომ ნებისმიერი ვექტორი შეგვიძლია გავშალოთ როგორც საბაზისო ვექტორების რაღაც კომბინაცია.

$$\vec{\boldsymbol{x}} = \sum_{i=1}^{N} \vec{\boldsymbol{e}}_i x^i \tag{3.26}$$

ან მატრიცული სახით

$$\vec{\boldsymbol{x}} = [\vec{\boldsymbol{e}}_i]^{\mathrm{t}}[x^i] \tag{3.27}$$

ზემოთ მოყვანილ ტოლობებს (3.10) და (3.11) შეგვიძლია შევხედოთ როგორც საკოორდინატო ბაზისის გარდაქმნას

$$\mathcal{B}' \equiv \mathcal{L}(\mathcal{B}) = \mathcal{B} \cdot \mathbf{C}$$

$$\mathcal{B}' \equiv (\vec{\boldsymbol{e}}_{1'}, \vec{\boldsymbol{e}}_{2'}, \vec{\boldsymbol{e}}_{3'}) = \mathcal{L}([\vec{\boldsymbol{e}}_i]^t) = [\vec{\boldsymbol{e}}_i]^t \cdot \mathbf{C}$$

3D შემთხვევისათვის გვექნება:

$$(\mathcal{L}(\vec{\boldsymbol{e}}_1), \mathcal{L}(\vec{\boldsymbol{e}}_2), \mathcal{L}(\vec{\boldsymbol{e}}_3)) = (\vec{\boldsymbol{e}}_{1'}, \vec{\boldsymbol{e}}_{2'}, \vec{\boldsymbol{e}}_{3'}) = (\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \vec{\boldsymbol{e}}_3) \cdot \begin{pmatrix} c^1_{1'} & c^1_{2'} & c^1_{3'} \\ c^2_{1'} & c^2_{2'} & c^2_{3'} \\ c^3_{1'} & c^3_{2'} & c^3_{3'} \end{pmatrix} \tag{3.28}$$

$\mathbf{C}$ მატრიცას ბაზისის გადასვლის მატრიცას უწოდებენ:

$$\mathbf{C} = \begin{pmatrix} c^1_{1'} & c^1_{2'} & c^1_{3'} \\ c^2_{1'} & c^2_{2'} & c^2_{3'} \\ c^3_{1'} & c^3_{2'} & c^3_{3'} \end{pmatrix} \tag{3.29}$$

სურ 3.2: ვექტორი $\vec{x}$ „მელა-ყურძენი“ არ იცვლება საკოორდინატო სისტემის გარდაქმნისას.

ამ მატრიცის სვეტები წარმოადგენენ ახალი ბაზის ვექტორების კოორდინატებს ძველი საკოორდინატო სისტემის მიმართ(ანუ, როგორ "ჩანს"ახალი ბაზისი ძველ საკოორდინატო სისტემაში).

რა სახე ექნება $\vec{x}$ ვექტორის კოორდინატებს ახალ საკოორდინატო სისტემაში? რადგან $\vec{x}$ არ გარდაიქმნება, ნებისმიერ გარდაქმნილ საკოორდინატო სისტემაში უნდა დარჩეს იგივე სურ.3.2

$$\vec{x} = \vec{e}_i x^i = \vec{e}_{i'} x^{i'} = \vec{e}_i c^i_{i'} x^{i'} \tag{3.30}$$

საიდანაც გვექნება

$$x^i = c^i_{i'} x^{i'} \tag{3.31}$$

და მატრიცული სახით

$$\begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix} = \mathbf{C} \begin{pmatrix} x^{1'} \\ x^{2'} \\ x^{3'} \end{pmatrix} \tag{3.32}$$

საიდანაც

$$\begin{pmatrix} x^{1'} \\ x^{2'} \\ x^{3'} \end{pmatrix} = \mathbf{C}^{-1} \begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix} \tag{3.33}$$

$$x^{i'} = \{\mathbf{C}^{-1}\}^{i'}_j x^j \tag{3.34}$$

$$[x']_{\mathcal{B}'} = \mathbf{C}^{-1}_{\mathcal{B}\to\mathcal{B}'} [x]_{\mathcal{B}} \tag{3.35}$$

რაც ნიშნავს, რომ თუ ბაზისის გადასვლის მატრიცა არის $\mathbf{C}$, ვექტორის კომპონენტები ბაზისის გარდაქმნისას გარდაიქმნება $\mathbf{C}^{-1}$ მატრიცის საშუალებით მარცხნიდან გამრავლებით. ვექტორის კომპონენტების გარდაქმნის ასეთ წესს უწოდებენ კონტრავარიანტულად გარდაქმნას, ხოლო ვექტორს კონტრავარიანტულ ვექტორს (ვექტორის მესამე განმარტება).

3.6.2 პასიური გარდაქმნის ინტერპრეტაცია კომპ. გრაფიკაში ვთქვათ მოცემული გვაქვს ვექტორი $\vec{v} = [\vec{b}]^t[c]$ ბაზისში $[\vec{b}]^t$, და გვაქვს ბაზისი $[\vec{a}]^t$ რომელიც თავის მხრივ მოიცემა $[\vec{b}]^t$ ბაზისის გარდაქმნით

$$[\vec{a}]^t = [\vec{b}]^t \cdot M$$

ანუ

$$[\vec{a}]^t \cdot M^{-1} = [\vec{b}]^t$$

მაშინ მოცემული $\vec{v}$ შეგვიძლია ჩავწეროთ როგორც

$$\vec{v} = [\vec{b}]^t[c] = [\vec{a}]^t\mathbf{M}^{-1}[c]$$

ეს არაა გარდაქმნის ფორმულა, უბრალოდ ერთი და იგივე ვექტორი ჩაწერილია სხვადასხვა ბაზისში. საკოორდინატო ვექტორი $[c]$ წარმოადგენს $\vec{v}$ ვექტორს $[\vec{b}]^t$ საკოორდინატო ბაზისის მიმართ და $M^{-1}[c]$ საკოორდინატო ვექტორი წარმოადგენს იგივე ვექტორს ოღონდ $[\vec{a}]^t$ ბაზისის მიმართ.

მაგალითი 3.3: პასიური გარდაქმნა

ვთქვათ ვექტორი(ვარსკვლავი) მოცემულია მსოფლიო ბაზისის $[\vec{b}]^t$ მიმართ (ეს იქნება მზესთან დაკავშირებული სისტემა), და $[\vec{a}]^t$ ობიექტთან დაკავშირებული ბაზისი(ვთქვათ დედამიწასთან დაკავშირებული სისტემა) ემთხვეოდა მსოფლიო ბაზისს(მზეს). შემდეგ მოვაბრუნეთ ობიექტი ანუ მოვაბრუნეთ მასთან დაკავშირებული ბაზისი (დავუშვათ დედამიწა წანაცვლებულია მზიდან თავის ორბიტაზე და მობრუნდა რაღაც კუთხით მზის გარშემო. წანაცვლება როგორ იქნება მატრიცაში ასახული ამას მოგვიანებით ვნახავთ. ახლა დავუშვათ მატრიცაში გვაქვს მხოლოდ მობრუნება). მაშინ იგივე წერტილს(ვარსკვლავს) გარდაქმნილი (მობრუნებული) ბაზისის მიმართ(დედამიწის მიმართ) ექნება კოორდინატები $\mathbf{M}^{-1}[c]$, სადაც $\mathbf{M}$ არის (დედამიწის) გარდაქმნის მატრიცა მსოფლიოს(მზის) მიმართ.

ზემოთ მოყვანილი პირველი თეორემა შეგვიძლია დავამტკიცოთ **„ალგებრული გზით“.** განმარტების თანახმად $\vec{x} \in \mathcal{V}$-ზე მოქმედება $\mathcal{W}$-დან არის შემდეგი:

$$\mathcal{A}\vec{x} = \mathcal{A}(\vec{v}_i)x^i = [\vec{w}]^{\mathrm{t}}\mathbf{A}[x] \tag{3.36}$$

ისევ, განმარტების თანახმად $\vec{x}' \in \mathcal{V}'$-ზე მოქმედება $\mathcal{W}'$-დან არის შემდეგი

$$\mathcal{A}\vec{x}' = \mathcal{A}(\vec{v}_{i'})x^{i'} = [\vec{w}']^{\mathrm{t}}\mathbf{A}'[x'] \tag{3.37}$$

გავამარტივოთ ბოლო განტოლება. ამისათვის გამოვიყენოთ ბაზისის და კოორდინატების გარდაქმნის ფორმულები $\vec{w}$ ბაზისისთვის და $[x']$ კოორდინატებისთვის

! $$\begin{aligned}[\vec{w}']^{\mathrm{t}} &= [\vec{w}]^{\mathrm{t}}\mathbf{D} \\ [x'] &= \mathbf{C}^{-1}[x]\end{aligned} \tag{3.38}$$

მივიღებთ

$$[\vec{w}']^{\mathrm{t}}\mathbf{A}'[x'] = [\vec{w}]^{t}\mathbf{D}\mathbf{A}'\mathbf{C}^{-1}[x] \tag{3.39}$$

თუ შევადარებთ 3.36 და გავითვალისწინებთ რომ სისტემის გარდაქმნაზე(პასიური გარდაქმნა) ვექტორი არაა დამოკიდებული, მივიღებთ

$$[\vec{w}]^{\mathrm{t}}\mathbf{A}[x] = [\vec{w}]^{t}\mathbf{D}\mathbf{A}'\mathbf{C}^{-1}[x] \tag{3.40}$$

რაც ნიშნავს

$$\mathbf{A} = \mathbf{D}\mathbf{A}'\mathbf{C}^{-1} \tag{3.41}$$

საიდანაც მივიღებთ

$$\mathbf{A}' = \mathbf{D}^{-1}\mathbf{A}\mathbf{C} \tag{3.42}$$

3.7 ევკლიდური სივრცე

გეომეტრიული ვექტორის იდეის განზოგადება ობიექტებზე რომლებიც აკმაყოფილებენ წრფივი სივრცის აქსიომებს გვაძლევს წრფივ სივრცეს. წრფივი სივრცე კარგი ინსტრუმენტია ბევრი ამოცანის აღსაწერად, მაგრამ საკმარისი არაა. მაგალითად, არაფერი გვით-

ქვამს ვექტორის სიგრძეზე. გვაქვს ვექტორები, სხვადასხვა საკოორდინატო სისტემები. წრფივი ასახვები ერთი სივრციდან მეორეში ან იგივე სივრცეში(ოპერატორები, რასაც გარდაქმნები ვუწოდეთ), მაგრამ ვექტორული სიდიდის დასახასიათებლად გჭირდება ვექტორის სიგრძეც. ვექტორების ურთიერთ ორიენტაციის განსასაზღვრად გვჭირდება მათ შორის კუთხეები. ეს ე.წ. *„მეტრიკული“* თვისებები უბრალოდ არაა განმარტებული წრფივი სივრცისათვის. ამიტომ შემოგვაქვს ევკლიდური სივრცის ცნება

განმარტება 3.3: ევკლიდური სივრცე

ნამდვილ წრფივ სივრცეს უწოდებენ ევკლიდურ სივრცეს E_n, თუ $\forall \vec{\boldsymbol{x}}$ და $\vec{\boldsymbol{y}}$ ელემენტს შეგვიძლია შევუსაბამოთ რაღაც წესით რაღაც რეალური რიცხვი $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}})$ რასაც ვუწოდებთ ორი ვექტორის სკალარულ ნამრავლს. სკალარული ნამრავლი უნდა აკმაყოფილებდეს პირობებს:

Ⅰ.1 $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) = (\vec{\boldsymbol{y}}, \vec{\boldsymbol{x}})$

Ⅰ.2 $(\lambda\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) = \lambda(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}})$, სადაც $\lambda \in \mathbb{Z}$

Ⅰ.3 $(\vec{\boldsymbol{x}} + \vec{\boldsymbol{z}}, \vec{\boldsymbol{y}}) = (\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) + (\vec{\boldsymbol{z}}, \vec{\boldsymbol{y}})$

Ⅰ.4 მოცემული გვაქვს $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{x}}) \geq 0$ და $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{x}}) = 0$ მხოლოდ და მხოლოდ მაშინ როცა $\vec{\boldsymbol{x}} = \mathbf{0}$

! ყურადღება მიაქციეთ, რომ ევკლიდური სივრცე 3.3 განმარტებაში

- სივრცე შეიძლება იყოს უსასრულო განზომილებიანი.
- თუ **Ⅰ.4** $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{x}}) \geq 0$ პირობა არ სრულდება ასეთ სივრცეს უწოდებენ ფსევდო ევკლიდურს (ფარდობითობის სპეციალური თეორია).
- თუ სკალარულ ნამრავლს განვმარტავთ კომპლექსურ წრფივ სივრცეში, მაშინ ვღებულობთ *უნიტარულ* სივრცეს (კომპლექსური სიგნალები, კვანტური მექანიკა, ველის თეორია)

3.8 სკალარული ნამრავლი და მეტრიკა

ჩავწეროთ სკალარული ნამრავლი ბაზისის ენაზე:

$$(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) \equiv \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}} = \vec{\boldsymbol{e}}_i x^i \cdot \vec{\boldsymbol{e}}_j y^j = (\vec{\boldsymbol{e}}_i, \vec{\boldsymbol{e}}_j) x^i y^j = g_{ij} x^i y^j = x^i y_i \tag{3.43}$$

სადაც

$$g_{ij} = (\vec{\boldsymbol{e}}_i, \vec{\boldsymbol{e}}_j) \text{ და } x_i = g_{ij} x^j \tag{3.44}$$

ორთონორმირებული ბაზისის შემთხვევაში $(\vec{\boldsymbol{e}}_i, \vec{\boldsymbol{e}}_j) = g_{ij} = \delta_{ij}$ რაც ნიშნავს რომ $x_i = x^i$და

$$(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) = \delta_{ij} x_i y^j = x_i y^i = x_i y_i = x^i y^i \tag{3.45}$$

ჩავწეროთ ზემოთ მოყვანილი განტოლებები მატრიცული სახით

$$\begin{aligned}(\vec{\boldsymbol{x}},\vec{\boldsymbol{y}}) \equiv &\vec{\boldsymbol{x}}\cdot\vec{\boldsymbol{y}} = ([\vec{\boldsymbol{e}}_i]^{\mathrm{t}}[x^i])^{\mathrm{T}}([\vec{\boldsymbol{e}}_j]^{\mathrm{t}}[y^j])\\ =&[x^i]^{\mathrm{t}}[\vec{\boldsymbol{e}}_i]^{\mathrm{t}\mathrm{T}}[\vec{\boldsymbol{e}}_j]^{\mathrm{t}}[y^j]\\ =&(x^1,\cdots,x^n)\begin{pmatrix}\vec{\boldsymbol{e}}_1\cdot\vec{\boldsymbol{e}}_1, & \cdots, & \vec{\boldsymbol{e}}_1\cdot\vec{\boldsymbol{e}}_n\\ \vdots & \vdots & \vdots\\ \vec{\boldsymbol{e}}_n\cdot\vec{\boldsymbol{e}}_1, & \cdots & \vec{\boldsymbol{e}}_n\cdot\vec{\boldsymbol{e}}_n\end{pmatrix}\begin{pmatrix}y^1\\ \vdots\\ y^n\end{pmatrix}\\ =&[x]^{\mathrm{t}}[\mathbf{G}][y]\end{aligned} \tag{3.46}$$

მატრიცას

$$\mathbf{G} = \begin{pmatrix}\vec{\boldsymbol{e}}_1\cdot\vec{\boldsymbol{e}}_1, & \cdots, & \vec{\boldsymbol{e}}_1\cdot\vec{\boldsymbol{e}}_n\\ \vdots, & \vdots, & \vdots\\ \vec{\boldsymbol{e}}_n\cdot\vec{\boldsymbol{e}}_1, & \cdots & \vec{\boldsymbol{e}}_n\cdot\vec{\boldsymbol{e}}_n\end{pmatrix} \tag{3.47}$$

გრამის(Gram) მატრიცა ეწოდება, რაც ფიზიკაში ცნობილია როგორც მეტრიკა $\{\mathbf{G}\}_{ij} = g_{ij}$.

სკალარული ნამრავლის ზოგადი ფორმულა არის განტ. 3.46.

მხოლოდ ევკლიდეს სივრცეში დეკარტეს საკოორდინატო სისტემაში, ორთონორმირებული ბაზისის შემთხვევაში როცა $\mathbf{G}$ ერთეულოვანი მატრიცაა გვაქვს სკოლიდან ცნობილი ფორმულები ვექტორის სიგრძის და სკალარული ნამრავლის შესახებ.

ზემოთ მოყვანილი მსჯელობიდან ასევე ჩანს, რომ ოპერაცია „ინდექსის ჩამოწევა“ ხორციელდება მეტრიკის საშუალებით და მხოლოდ ევკლიდეს სივრცეში დეკარტეს საკოორდინატო სისტემაში, ორთონორმირებული ბაზისის შემთხვევაში ქვედა და ზედა ინდექსიანი კოორდინატები ერთმანეთს ემთხვევა. ზოგადად $x_i = g_{ij}x^j$ ეწოდება „კოვარიანტული“ კოორდინატი.

თუ $(\vec{\boldsymbol{x}},\vec{\boldsymbol{x}})$ არსებობს, მაშინ განმარტებული გვაქვს ვექტორის სიგრძე(ნორმა) როგორც

$$||\vec{\boldsymbol{x}}|| = \sqrt{(\vec{\boldsymbol{x}},\vec{\boldsymbol{x}})} = \sqrt{g_{ij}x^ix^j} \tag{3.48}$$

ნებისმიერი ორი ვექტორისათვის ადგილი აქვს კოში — ბუნიაკოვსკის უტოლობას:

$$(\vec{\boldsymbol{x}},\vec{\boldsymbol{y}})^2 \le (\vec{\boldsymbol{x}},\vec{\boldsymbol{x}})\cdot(\vec{\boldsymbol{y}},\vec{\boldsymbol{y}}) \tag{3.49}$$

რაც საშუალებას იძლევა განიმარტოს ორ ვექტორს შორის კუთხე:

$$\cos\alpha = \frac{(\vec{x}, \vec{y})}{||\vec{x}|| \cdot ||\vec{y}||} \tag{3.50}$$

არა ნულ $\vec{x}$ და $\vec{y}$ ვექტორებს ეწოდებათ ორთოგონალური თუ $(\vec{x}, \vec{y}) = 0$.

თუ მოცემულია რაღაც ბაზისი $[\vec{f}]^{\mathrm{t}}$, მაშინ ყოველთვის შეგვიძლია ავაგოთ ორთოგონალური ბაზისი $[\vec{e}]^{\mathrm{t}}$, როგორც(გრამ — შმიდტის ორთოგონალიზაცია):

$$\begin{aligned} \vec{e}_1 &= \vec{f}_1 \\ \vec{e}_i &= \vec{f}_i - \sum_{k=1}^{i-1} \frac{(\vec{f}_i, \vec{e}_k)}{(\vec{e}_k, \vec{e}_k)} \vec{e}_k, \quad i = 2, 3, \ldots, n \end{aligned} \tag{3.51}$$

დავალება 3.1: ბაზისის ორთოგონალიზაცია

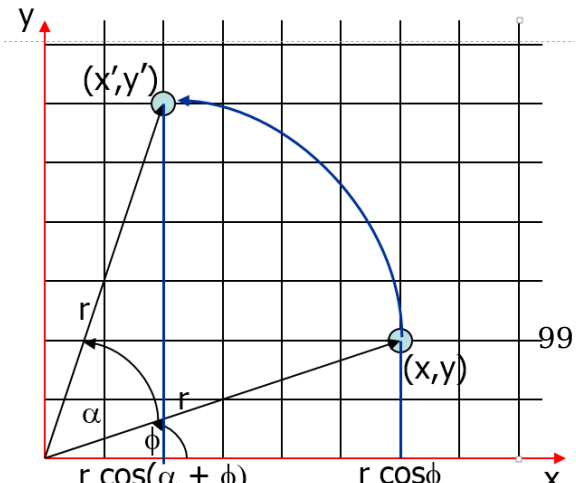
1. (3.51) განტოლებები ჩაწერეთ 3D შემთხვევისათვის.
2. მოცემული $[\vec{f}]^{\mathrm{t}}$ ბაზისისთვის: $\vec{f}_1 = (1,1,1)^{\mathrm{T}}$, $\vec{f}_2 = (1,3,0)^{\mathrm{T}}$ და $\vec{f}_3 = (0,3,-1)^{\mathrm{T}}$ ააგეთ ორთოგონალური ბაზისი $[\vec{e}]^{\mathrm{t}}$.
3. დათვალეთ გრამის მატრიცა ორივე ბაზისისთვის.
4. $\mathcal{F} = [\vec{f}]^{\mathrm{t}}$ ბაზისში მოცემულია ორი ვექტორი კოორდინატებით $[a] = [1,1,1]^{\mathrm{T}}_{\mathcal{F}}$ და $[b] = [2,1,3]^{\mathrm{T}}_{\mathcal{F}}$. გამოთვალეთ კუთხე მათ შორის.

ბაზისის გარდაქმნისას $\mathbf{C}$ მატრიცით, გრამის მატრიცა გარდაიქმნება როგორც

$$\mathbf{G}' = \mathbf{C}^{\mathrm{T}}\mathbf{G}\mathbf{C} \tag{3.52}$$

3.9 აქტიური გარდაქმნები ევკლიდის სივრცეში

როგორც ავღნიშნეთ აქტიური გარდაქმნისას გარდაიქმნება ობიექტი და არა ბაზისი. განვიხილოთ მობრუნება სიბრტყეზე. ჩავთვალოთ რომ დადებითი მიმართულებაა საათის ისრის საწინააღმდეგო მიმართულება. ანუ თუ მობრუნება ხდება α კუთხეზე მობრუნება გვაქვს საათის ისრის მოძრაობის საწინააღმდეგო მიმართულებით სურ.3.3.



$$x = r\cos\phi; y = r\sin\phi \tag{3.53}$$

$$x' = r\cos(\alpha+\phi); y' = r\sin(\alpha+\phi) \tag{3.54}$$

$$\begin{aligned} x' &= r\cos\alpha\cos\phi – r\sin\alpha\sin\phi \\ &= x\cos\alpha – y\sin\alpha \end{aligned} \tag{3.55}$$

$$\begin{aligned} y' &= r\sin\alpha\cos\phi + r\cos\alpha\sin\phi \\ &= x\sin\alpha + y\cos\alpha \end{aligned} \tag{3.56}$$

მიღებული შედეგი შეიძლება ჩაიწეროს მატრიცული სახით

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.57}$$

ესაა ვექტორის კოორდინატების გარდაქმნა.

ვნახოთ როგორ გარდაქმნის ეს გარდაქმნა ბაზისის ვექტორებს. სიმარტივისთვის საბაზისო ვექტორები ავღნიშნოთ $\vec{e}_x$ და $\vec{e}_y$. მაშინ $\vec{e}_x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ და ზემოთ მოყვანილ ფორმულაში თუ ჩავსვამთ ამ კოორდინატებს $\vec{e}_x$ ვექტორისთვის მივიღებთ ახალ კოორდინატებს

$$\begin{pmatrix} {e'_x}^x \\ {e'_x}^y \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} \tag{3.58}$$

ანალოგიურად, $\vec{e}_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ და

$$\begin{pmatrix} {e'_y}^x \\ {e'_y}^y \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix} \tag{3.59}$$

ე.ი. მივიღეთ მობრუნებული ბაზისი $\mathcal{B}'$ საბაზისო ვექტორებით ძველი $\mathcal{B}$ ბაზისის მიმართ

$$\vec{e}'_x = (\vec{e}_x, \vec{e}_y) \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} \quad \text{და} \quad \vec{e}'_y = (\vec{e}_x, \vec{e}_y) \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix} \tag{3.60}$$

ანუ (3.57) გარდაქმნის ფორმულა საბაზისო ვექტორებს გარდაქმნის როგორც

$$\begin{aligned}\vec{e}'_x &= \vec{e}_x \cos\alpha + \vec{e}_y \sin\alpha \\ \vec{e}'_y &= -\vec{e}_x \sin\alpha + \vec{e}_y \cos\alpha\end{aligned} \tag{3.61}$$

ადვილად დავინახავთ, რომ გამოსახულება $\left(\vec{e}'_x, \vec{e}'_y,\right) \cdot \begin{pmatrix} x \\ y \end{pmatrix}$ იგივეა რაც $\left(\vec{e}_x, \vec{e}_y,\right) \cdot \begin{pmatrix} x' \\ y' \end{pmatrix}$

ანუ მივიღეთ, რომ ვექტორის გარდაქმნა შეგვიძლია გავაკეთოთ ორი გზით:

$$\vec{r}\,' = \phi(\vec{r}) = \boxed{(\vec{e}_x, \vec{e}_y) \begin{pmatrix} x' \\ y' \end{pmatrix}} = (\vec{e}_x, \vec{e}_y) \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \boxed{(\vec{e}'_x, \vec{e}'_y) \begin{pmatrix} x \\ y \end{pmatrix}} \tag{3.62}$$

ზოგადი წრფივი გარდაქმნის შემთხვევაში (ანუ აუცილებელი რომ გვქონდეს მობრუნება) მობრუნების მატრიცის ნაცვლად გვექნება მატრიცა [1]

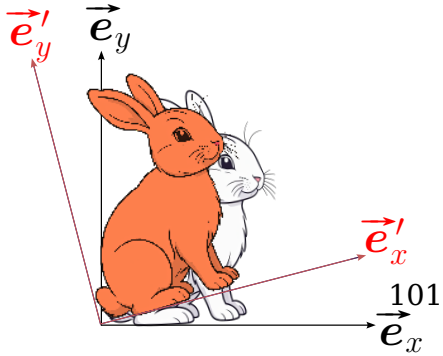
$$\mathbf{C} = \begin{pmatrix} c^1_{1'} & c^1_{2'} \\ c^2_{1'} & c^2_{2'} \end{pmatrix} \tag{3.63}$$

3D გარდაქმნების შემთხვევაში კი გვაქვს მატრიცა (3.29)

$$\mathbf{C} = \begin{pmatrix} c^1_{1'} & c^1_{2'} & c^1_{3'} \\ c^2_{1'} & c^2_{2'} & c^2_{3'} \\ c^3_{1'} & c^3_{2'} & c^3_{3'} \end{pmatrix} \tag{3.64}$$

სადაც მობრუნების შემთხვევაში (ვგულისხმობთ, რომ გვაქვს დეკარტეს ორთონორმირებული საკოორდინატო სისტემა.) $c^j_{i'}$ ელემენტები წარმოადგენენ მიმართულების კოსინუსებს (კუთხის კოსინუსი $\vec{e}_{i'}$ მობრუნებულ საბაზისო ვექტორს და მოუბრუნებელ $\vec{e}_j$ ვექტორს შორის [2], [3], [4])

განტოლებაში (3.62) უკანასკნელი ტოლობა ნიშნავს, რომ მობრუნდა ბაზისი და ვექტორის კოორდინატები დარჩა იგივე (სურ. 3.4). რაც ნიშნავს, რომ აქტიურ გარდაქმნა შეგვიძლია მივიღოთ ორი გზით:



სურ 3.4: ბრუნდება სისტემა და მასთან ერთად კურდღელი

ა) გარდაიქმნება ობიექტის კოორდინატები. ბაზისი რჩება იგივე. კოორდინატები მარცხნიდან მრავლდება გარდაქმნის მატრიცაზე;

ბ) ყოველთვის შეგვიძლია დავუშვათ, რომ ყოველი ობიექტი მოცემულია რაღაც საკუთარ ბაზისში(თანმდევი, საკუთარი საკოორდინატო სისტემა) და გარდაიქმნება ობიექტის საბაზისო ვექტორები(საკუთარი სისტემა), ხოლო ობიექტის კოორდინატები საკუთარი ბაზისის მიმართ არ იცვლება .

თითქოს საკუთარი საკოორდინატო სისტემა არის „ყუთი“, სადაც მყარად დევს ობიექტი. მსოფლიო საკოორდინატო სისტემის მიმართ გარდაიქმნება „ყუთი“ და მასთან ერთად ობიექტი. საკუთარი სისტემის საბაზისო ვექტორები მარჯვნიდან მრავლდება გარდაქმნის მატრიცაზე.

3.10 ორთოგონალური გარდაქმნები

დავუშვათ, გვაქვს ორთონორმირებული ბაზისი და გარდაქმნა რომელიც ბაზისს ტოვებს ორთონორმირებულს. მაშინ (3.52) მივიღებთ, რომ

$$C^{\mathrm{T}}C = 1 \text{ ანუ } C^{\mathrm{T}} = C^{-1} \tag{3.65}$$

ასეთ გარდაქმნას ორთოგონალური გარდაქმნა ეწოდება. ვექტორის სიგრძე არის სკალარი რაც ნიშნავს, რომ მობრუნებისას არ იცვლება.

$$\vec{\boldsymbol{r}}'\vec{\boldsymbol{r}}' = \vec{\boldsymbol{r}}\,\vec{\boldsymbol{r}} \tag{3.66}$$

რაც ნიშნავს

$$\begin{aligned}
([\vec{\boldsymbol{e}}]^t\mathbf{C}[x])^{\mathrm{T}}[\vec{\boldsymbol{e}}]^t\mathbf{C}[x] &= ([\vec{\boldsymbol{e}}]^t[x])^{\mathrm{T}}[\vec{\boldsymbol{e}}]^t[x] \\
[x]^t\mathbf{C}^{\mathrm{T}}[\vec{\boldsymbol{e}}]^{t\mathrm{T}}[\vec{\boldsymbol{e}}]^t\mathbf{C}[x] &= [x]^{t\mathrm{T}}[\vec{\boldsymbol{e}}]^{t\mathrm{T}}[\vec{\boldsymbol{e}}]^t[x] \\
[x]^t\mathbf{C}^{\mathrm{T}}\mathbf{E}\mathbf{C}[x] &= [x]^t\mathbf{E}[x] \\
[x]^t\mathbf{C}^{\mathrm{T}}\mathbf{C}[x] &= [x]^t[x] \\
\mathbf{C}^{\mathrm{T}}\mathbf{C} &= \mathbf{E}
\end{aligned} \tag{3.67}$$

ანუ მობრუნება არის ორთოგონალური გარდაქმნა. თუ გავიხსენებთ მატრიცის დეტერმინანტის თვისებას

$$\begin{aligned} \det(\mathbf{C}^{\mathrm{T}}\mathbf{C}) &= 1 \\ (\det\mathbf{C})^2 &= 1 \\ \det\mathbf{C} &= \pm 1 \end{aligned} \tag{3.68}$$

მატრიცა განი.(3.62) არის Z ღერძის გარშემო α კუთხეზე საათის ისრის მიმართულების საწინააღმდეგოდ მობრუნების მატრიცა $\mathbf{R}_z(\alpha)$. როგორც ვხედავთ დეტერმინანტი მართლაც ტოლია 1. საწინააღმდეგო მიმართულებით მობრუნება იქნება $-\alpha$ კუთხეზე მობრუნება $\mathbf{R}_z(-\alpha)$. ნათელია რომ $\mathbf{R}_z^{-1}(\alpha) = \mathbf{R}_z(-\alpha) = \mathbf{R}_z^{\mathrm{T}}(\alpha)$ და $\det|\mathbf{R}_z(\alpha)| = 1$

რა ორთოგონალურ გარდაქმნას აქვს დეტერმინანტი ტოლი -1? ადვილია ვაჩვენოთ რომ გარდაქმნა

$$\begin{pmatrix} \cos\alpha & \sin\alpha \\ \sin\alpha & -\cos\alpha \end{pmatrix} \tag{3.69}$$

ასევე არის ორთოგონალური გარდაქმნა რომლის დეტერმინანტია -1 და წარმოადგენს მობრუნებას, შემდგომი Y ღერძის არეკვლით. რაც ნიშნავს რომ მარჯვენა სისტემა გახდა მარცხენა. ანუ სისტემამ შეიცვალა ორიენტაცია.

ორთოგონალურ გარდაქმნებს, რომლებიც არ ცვლიან ორიენტაციას საკუთრივი გარდაქმნები ეწოდებათ. რომლებიც ცვლიან ორიენტაციას — არა საკუთრივი. მობრუნება არის საკუთრივი ორთოგონალური გარდაქმნა.

ორ ბაზისს ეწოდება ერთნაირად ორიენტირებული თუ გარდაქმნის (არაა აუცილებელი ეს იყოს მობრუნება) მატრიცის დეტერმინანტი მეტია ნულზე ანუ ბაზისი გარდაქმნილია საკუთრივი გარდაქმნით.

3.11 ნორმალის გარდაქმნა

ხშირად საჭიროა ობიექტის გარდაქმნებთან ერთად ნორმალის გარდაქმნაც. არის ორი გზა: გარდავქმნათ ობიექტი და გარდაქმნილი ობიექტისთვის გადავითვალოთ ნორმალები, და მეორე, თუ ნორმალები უკვე გამოთვლილი გვაქვს და გარდავქმნით ობიექტს, მაშინ არაა აუცილებელი ნორმალების თავიდან გამოთვლა თუ ვიცით მათი გარდაქმნის წესი.

$$\vec{\boldsymbol{n}} \cdot \vec{\boldsymbol{v}} = \vec{\boldsymbol{n}}^{\mathrm{T}} \vec{\boldsymbol{v}} = [n_1 n_2 n_3] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = 0$$

ვთქვათ $\mathbf{M}$, არის $\vec{\boldsymbol{v}}$-ს გარდაქმნის მატრიცა. საძიებელია გარდაქმნის მატრიცა $\mathbf{L}$ ისეთი რომ

$$\vec{\boldsymbol{n}}' \cdot \vec{\boldsymbol{v}}' = 0$$

$$\vec{\boldsymbol{n}}' = \mathbf{L}\vec{\boldsymbol{n}}$$

$$\vec{\boldsymbol{v}}' = \mathbf{M}\vec{\boldsymbol{v}}$$

$$\mathbf{L}\vec{\boldsymbol{n}} \cdot \mathbf{M}\vec{\boldsymbol{v}} = 0$$

ანუ მატრიცული სახით

$$(\mathbf{L}\vec{\boldsymbol{n}})^{\mathrm{T}}\mathbf{M}\vec{\boldsymbol{v}} = 0$$

ეს ტოლობა ძალაში იქნება ნებისმიერი ნორმალისთვის და ვექტორისთვის მაშინ როცა

$$(\mathbf{L}\vec{\boldsymbol{n}})^{\mathrm{T}}\mathbf{M}\vec{\boldsymbol{v}} = \vec{\boldsymbol{n}}^{\mathrm{T}}\mathbf{L}^{\mathrm{T}}\mathbf{M}\vec{\boldsymbol{v}} \Rightarrow \mathbf{L}^{\mathrm{T}}\mathbf{M} = \mathbf{E}$$

ანუ

$$\mathbf{L} = (\mathbf{M}^{-\mathbf{1}})^{\mathrm{T}}$$

დაწვრილებით იხ. [5]

ორთოგონალური გარდაქმნებისას $\mathbf{L} = (\mathbf{M}^{-\mathbf{1}})^{\mathrm{T}} = \mathbf{M}$. ანუ მობრუნებისას ნორმალიც ბრუნდება იგივე კუთხით .$\mathcal{E}_n$

3.12 აფინური სივრცე

წრფივ სივრცეში შემოვიტანეთ სკალარული ნამრავლი და განვმარტეთ ევკლიდური სივრცე E^n (ამბობენ რომ შემოტანილ იქნა ევკლიდური სტრუქტურა), მაგრამ ესეც არაა საკმარისი რეალური სამყაროს აღსაწერად, ვინაიდან გადატანა არაა წრფივი ოპერაცია.

მართლაც წრფივი ასახვიდან (განტ. (3.14))

$$\mathcal{A}(\vec{\boldsymbol{a}} + \vec{\boldsymbol{b}}) = \mathcal{A}\vec{\boldsymbol{a}} + \mathcal{A}\vec{\boldsymbol{b}}, \quad \mathcal{A}(\lambda\vec{\boldsymbol{a}}) = \lambda\mathcal{A}\vec{\boldsymbol{a}} \tag{3.70}$$

გამომდინარეობს, რომ თუ განვმარტავთ გადატანას როგორც

$$\mathcal{T}\vec{\boldsymbol{a}} = \vec{\boldsymbol{a}} + \vec{\boldsymbol{c}} \tag{3.71}$$

მაშინ

$$\mathcal{T}\vec{a} = \vec{a} + \vec{c}$$
$$\mathcal{T}\vec{b} = \vec{b} + \vec{c}$$

მაგრამ

$$\mathcal{T}(\vec{a} + \vec{b}) = \vec{a} + \vec{b} + \vec{c} \neq \mathcal{T}\vec{a} + \mathcal{T}\vec{b} = \vec{a} + \vec{c} + \vec{a} + \vec{c} \tag{3.72}$$

როგორც ვხედავთ წრფივობის პირობა არ სრულდება. ამის მიზეზია შემდეგი: ა) გადატანის ოპერაცია განმარტებულია წერილისთვის და არა ვექტორისთვის, ბ) რადიუს ვექტორის შემოტანისთვის აუცილებელია საკოორდინატო სათავის (ათვლის წერტილის) შემოტანა, ბ) ე.წ. ნულ ვექტორიც რომ ჩავთვალოთ საკოორდინატო სისტემის სათავედ გადატანისას ნულ ვექტორი გადავა არა ნულ ვექტორში, გ) წრფივ სივრცეში არ გვაქვს წერტილის ცნება. ზოგადად წრფივი სივრცე შეიძლება არც იყოს გეომეტრიული ვექტორების სივრცე. მატრიცების, პოლინომების და ა.შ. არა გეომეტრიული წრფივი სივრცის შემთხვევაში აზრიც კი არ აქვს ათვლის სათავის (წერტილის) ცნებას.

გეომეტრიაში რაც ვისწავლეთ სკოლაში გვქონდა წერტილის ცნება. ეს იყო ევკლიდის გეომეტრია რომლის აქსიომატიკაც გადამუშავებულ იქნა ჰილბერტის მიერ. ეს აქსიომატიკა ემყარება წერტილის, წრფის და სიბრტყის ცნებებს და მათი გამოყენებით იგება მდებარეობის, მიკუთვნების, პარალელობის აქსიომები.

კიდევ ერთი აქსიომატიკა შემოტანილ იქნა ჰ. ვეილის მიერ 1918წ [6]. მტკიცდება რომ ვეილის და ჰილბერტის აქსიმები ერთი და იგივე გეომეტრიას იძლევა, მაგრამ ვეილის აქსიომები უფრო „ბუნებრივია“ და ადვილად ზოგადდება $n > 3$ განზომილებიანი სივრცისთვის რაც რთულია ჰილბერტის (ევკლიდის) აქსიომატიური მიდგომის შემთხვევაში. ვეილის აქსიომატიკა ემყარება ვექტორის და წერილის ცნებებს, რის გამოც „წერტილოვან-ვექტორულ“ აქსიომატიკასაც უწოდებენ. ამ გზით განმარტებულ სივრცეს კი „წერტილოვან-ვექტორულ სივრცეს“. სხვაგვარად ამ სივრცეს აფინურ სივრცეს უწოდებენ.

! აფინური ლათინურად `Affinis` ნიშნავს დაკავშირებულს, მსგავსს, მონათესავეს და აღნიშნავს რომ სივრცეში გვაქვს ორი მსგავსი ტიპის ობიექტები: წერტილები და ვექტორები.

წერტილებს ავღნიშნავთ დიდი დახრილი ასო ნიშნებით ხოლო ვექტორებს პატარა. გვაქვს:

- V წერტილების სიმრავლე $\boldsymbol{P}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$
- $\mathcal{K}$ ველზე განმარტებული $\mathcal{L}$ ვექტორული სივრცე ელემენტებით $\vec{\boldsymbol{p}}, \vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}}, \ldots$
- ყოველი ორი წერტილის მოწესრიგებული წყვილის $\boldsymbol{A}, \boldsymbol{B} \in V$, ვუსაბამებთ ვექტორს $\vec{\boldsymbol{x}} \in \mathcal{L}$ $\overrightarrow{\boldsymbol{AB}} = \vec{\boldsymbol{x}}$

განმარტება 3.4: აფინური სივრცე

აფინური სივრცე(V, $\mathcal{L}$) წრფივ $\mathcal{L}$ სივრცეზე ეწოდება რაღაც ბუნების ობიექტების რომლებსაც ვუწოდებთ წერტილებს, ერთობლიობას V, საიდანაც აღებულ მოწესრიგებულ წყვილს $\boldsymbol{A}, \boldsymbol{B}$ შეესაბაამება ვექტორი $\vec{\boldsymbol{x}} \in \mathcal{L}$ პირობით:

1. ყოველი $\boldsymbol{A}$ მოიძებნება ერთი და მხოლოდ ერთი წერტილი $\boldsymbol{B}$ ისეთი რომ $\overrightarrow{\boldsymbol{AB}} = \vec{\boldsymbol{x}}$
2. ნებისმიერი სამი წერტილისათვის $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ სრულდება პირობა $\overrightarrow{\boldsymbol{AB}} + \overrightarrow{\boldsymbol{BC}} = \overrightarrow{\boldsymbol{AC}}$

ზემოთ მოყვანილი განმარტებიდან წრფივი სივრცის განმარტებიდან გამომდინარეობს

- $\overrightarrow{\boldsymbol{AA}} = \vec{\boldsymbol{0}}$
- $\overrightarrow{\boldsymbol{AB}} = -\overrightarrow{\boldsymbol{BA}}$.
- ყოველის სამი $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ წერტილისათვის მოიძებნება წერტილი $\boldsymbol{D}$ ისეთი რომ $\overrightarrow{\boldsymbol{AB}} = \overrightarrow{\boldsymbol{CD}}$.

აფინური სივრცე თუ განმარტებული E^n ევკლიდურ სივრცეზე, მაშინ გვაქვს $\mathcal{E}^n$ აფინური სივრცე. ჰ.ვეილის წიგნში სწორედ ასეთ განმარტებას ვხვდებით. ერთი განსხვავებით, ვეილთან აფინური სივრცის განზომილება პოსტულირებულია და წრფივი სივრცის განზომილება აფინური სივრცის განზომილების ტოლია. ზემოთ მოყვანილ განმარტებაში კი აფინური სივრცის განზომილება მასთან ასოცირებული წრფივი სივრცის განზომილების ტოლია.

ზოგიერთ ლიტერატურაში ევკლიდური სივრცის ქვეშ გულისხმობენ სწორედ აფინურ ევკლიდურ სივრცეს(მაგ. სკოლის კურსის გეომეტრია). ზოგიერთი განასხვავებს წრფივ ევკლიდურ და აფინურ ევკლიდურ სივრცეებს [7], [8]. განმარტების 3.4 პირველი პუნქტი ნიშნავს [9, გვ. 8]

$$\boldsymbol{A} + \vec{\boldsymbol{x}} = \boldsymbol{B} \quad \boldsymbol{A} \text{ წერტილიდან თუ გადავდებთ } \vec{\boldsymbol{x}} \text{ ვექტორს მივიღებთ } \boldsymbol{B} \text{ წერტილს (ვეილი [6])} \tag{3.73}$$

გალიერთან მოყვანილია აფინური სივრცის იგივე განმარტება სახეცვლილი სახით (ესაა ვეილის ორიგინალური განმარტება):

განმარტება 3.5: აფინური სივრცე [9]

აფინური სივრცე ეწოდება სამეულს(ტრიპლეტს) (V, $\mathcal{L}$,+), რომელიც შედგება: არანულოვანი სიმრავლე V(წერტილები), წრფივი სივრცე $\mathcal{L}$(გადატანები, თავისუფალი ვექტორები) და განმარტებულია ქმედება $+ : V \times \mathcal{L} \to V$, რომელიც აკმაყოფილებს პირობებს:

1. $\boldsymbol{A} + \vec{\boldsymbol{0}} = \boldsymbol{A}$ ყოველი $\boldsymbol{A} \in V$
2. $(\boldsymbol{A} + \vec{\boldsymbol{u}}) + \vec{\boldsymbol{v}} = \boldsymbol{A} + (\vec{\boldsymbol{u}} + \vec{\boldsymbol{v}})$ ყოველი $\boldsymbol{A} \in V$ და $\vec{\boldsymbol{u}}, \vec{\boldsymbol{v}} \in \mathcal{L}$
3. ნებისმიერი ორი წერტილისათვის $\boldsymbol{A}, \boldsymbol{B} \in V$ არსებობს ერთადერთი ვექტორი $\vec{\boldsymbol{x}} \in \mathcal{L}$ ისეთი, რომ $\boldsymbol{A} + \vec{\boldsymbol{x}} = \boldsymbol{B}$.

ხშირად ეს უნიკალური ვექტორი აღინიშნება როგორც $\overrightarrow{AB}$ ან $B-A$
ამიტომ შეგვიძლია დავწეროთ

$$B = A + \vec{x}$$

ან

$$B = A + \overrightarrow{AB}$$

ან

$$B = A + (B - A)$$

3.12.1 აფინური სივრცის ვექტორიზაცია დავუშვათ გვაქვს აფინური სივრცე (V,$\mathcal{L}$,+). ავიღოთ რაიმე წერტილი O, მაშინ ნებისმიერი P მოიძებნება ვექტორი $\vec{p} \in \mathcal{L}$ ისეთი რომ $P = O + \vec{p} \equiv \overrightarrow{OP}$. ანუ O წერტილის არჩევა საშუალებას გვაძლევს შევუსაბამოთ ნებისმიერ წერტილს ვექტორი ვექტორული სივრციდან. ვექტორ $\overrightarrow{OP}$ ეწოდება რადიუს ვექტორი. ოპერაცია „წერტილს + ვექტორი“ იქნება „ვექტორს+ვექტორი“ $Q = P + \vec{v} = \overrightarrow{OP} + \vec{v}$. ცხადია ეს გაიგივება(წერტილი/რადიუს ვექტორი) დამოკიდებულია O არჩევაზე. აფინური სივრცის განზომილება წრფივი სივრცის განზომილების ტოლია. n განზომილებიანი აფინური სივრცე ნიშნავს, რომ გვაქვს $n+1$ წერტილი ისეთი რომ $\overrightarrow{OP}_i\, i = \overline{1,\ldots,n}$ ვექტორები ქმნიან $\mathcal{L}$ სივრცის ბაზისს(ასეთ წერტილებს აფინურად დამოუკიდებელი ეწოდება). O წერტილის არჩევა და საბაზისო ვექტორების ერთობლიობა ნიშნავს რომ გვაქვს საკოორდინატო სისტემა. მაგალითად, თუ ავიღებთ ორ წერტილს O და P. $\overrightarrow{OP} \equiv \vec{v}$ მაშინ $O + t \cdot \vec{v}$ სადაც $t \in \mathbb{R}$ მივიღებთ ერთ განზომილებიან აფინურ სივრცეს. თუ ავიღებთ სამ წერტილს, რომლებიც ერთ წრფეზე არ მდებარეობენ, O და P, Q. $\overrightarrow{OP} \equiv \vec{e}_1$, $\overrightarrow{OQ} \equiv \vec{e}_2$ მაშინ $O + t \cdot \vec{e}_1 + \lambda \vec{e}_2$ სადაც $t, \lambda \in \mathbb{R}$ აღწერს წერტილს სიბრტყეზე. ე.ი. მივიღეთ ორ განზომილებიანი აფინურ სივრცე — სიბრტყეს.

აფინური სივრცეების მაგალითია წრფე საკოორდინატო ღერძით. სიბრტყე დეკარტეს მართკუთხა საკოორდინატო სისტემით, ევკლიდური სივრცე დეკარტეს მართკუთხა საკოორდინატო სისტემით.

3.13 წერტილი სხვადასხვა საკოორდინატო სისტემაში

ჩვენ უკვე განვიხილეთ თუ როგორ „ჩანს“ ვექტორი სხვადასხვა ბაზისში. ახლა ვნახოთ როგორ აღიწერება წერტილი საკოორდინატო სისტემის ცვლილებისას.

განვიხილოთ სიბრტყის შემთხვევა. ვთქვათ, გვაქვს ორი დეკარტეს საკოორდინატო სისტემა $\vec{f} = [\vec{e}_1, \vec{e}_2, O]$ და $\vec{f}' = [\vec{e}_{1'}, \vec{e}_{2'}, O']$.

და $\mathbf{C}$ არის ბაზისის გადასვლის მატრიცა $(\vec{e}_{1'}, \vec{e}_{2'}) = (\vec{e}_1, \vec{e}_2)\mathbf{C}$. მაშინ $\overrightarrow{OP} = (\vec{e}_1, \vec{e}_2)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, $\overrightarrow{OO'} = (\vec{e}_1, \vec{e}_2)\begin{pmatrix} O'_1 \\ O'_2 \end{pmatrix}$,

$$\overrightarrow{O'P} = (\vec{e}_{1'}, \vec{e}_{2'})\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix} = (\vec{e}_1, \vec{e}_2)\mathbf{C}\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix}$$

ზემოთ მოყვანილ გაშლებს ბაზისის მიხედვით თუ შევიტანთ გამოსახულებაში

$$\overrightarrow{OP} = \overrightarrow{OO'} + \overrightarrow{O'P}$$

მივიღებთ

$$(\vec{e}_1, \vec{e}_2)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (\vec{e}_1, \vec{e}_2)\begin{pmatrix} O'_1 \\ O'_2 \end{pmatrix} + (\vec{e}_1, \vec{e}_2)\mathbf{C}\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix} = (\vec{e}_1, \vec{e}_2)\left(\mathbf{C}\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix} + \begin{pmatrix} O'_1 \\ O'_2 \end{pmatrix}\right) \tag{3.74}$$

ანუ

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \mathbf{C}\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix} + \begin{pmatrix} O'_1 \\ O'_2 \end{pmatrix} \tag{3.75}$$

და

$$\begin{pmatrix} x_{1'} \\ x_{2'} \end{pmatrix} = \mathbf{C}^{-1}\begin{pmatrix} x_1 - O'_1 \\ x_2 - O'_2 \end{pmatrix} \tag{3.76}$$

ანალოგიური ფორმულები გვექნება 3D შემთხვევაშიც

3.13.1 წერტილების აფინური კომბინაცია. $\overrightarrow{OP}$ რადიუს ვექტორი დამოკიდებულია საკოორდინატო სისტემაზე. ამ გაგებით ის არაა ვექტორი. ვექტორი არაა დამოკიდებული საკოორდინატო სისტემაზე. ორი ვექტორის წრფივი კომბინაცია იძლევა ვექტორს, რომელიც ასევე ცხადია არაა დამოკიდებული საკოორდინატო სისტემის არჩევაზე, განსხვავებით ორი რადიუს ვექტორის წრფივი კომბინაციისგან. მარტივი მაგალითი: გვაქვს ორი წერტილი და შევკრიბეთ მათი რადიუს ვექტორები. სხვა საკოორდინატო სისტემაში ამ ორ წერტილს ექნება სხვა რადიუს ვექტორები და მათი ჯამი იქნება ასევე სხვა ვექტორი.

არის ორი განსაკუთრებული შემთხვევა, როცა წერტილების კომბინაციას აზრი აქვს ანუ დამოუკიდებელია საკოორდინატო სათავის არჩევისგან:

1. ორი წერტილის სხვაობას აზრი აქვს, გვაძლევს ვექტორს $\vec{\boldsymbol{a}} = \boldsymbol{B} - \boldsymbol{A}$. ამ ფაქტის განზოგადებას წარმოადგენს შემდეგი $\sum_{i=1}^{n} \lambda_i \boldsymbol{P}_i$ არის ვექტორი როცა $\sum_{i=1}^{n} \lambda_i = 0$ და არაა დამოკიდებული საკოორდინატო სისტემის არჩევაზე
2. განვიხილოთ ორი წერტილი $\boldsymbol{B}$, $\boldsymbol{A}$ და $\boldsymbol{C}$ ისეთი რომ მდებარეობდეს ამ ორ წერტილის შემაერთებელ მონაკვეთზე. მაშინ $\boldsymbol{C} = (1-\lambda) \cdot \boldsymbol{A} + \cdot \lambda \boldsymbol{B}$. როცა $\lambda = 0$ მაშინ $\boldsymbol{C} = \boldsymbol{A}$, ხოლო როცა $\lambda = 1$ მაშინ $\boldsymbol{C} = \boldsymbol{B}$. როცა $\lambda = 1/2$ მაშინ $\boldsymbol{C} = \frac{\boldsymbol{A}+\boldsymbol{B}}{2}$. ანუ t პარამეტრის ცვლილებით მივიღებთ ნებისმიერ წერტილს მონაკვეთზე $[\boldsymbol{A}, \boldsymbol{B}]$
   ამ ფაქტის განზოგადებას წარმოადგენს შემდეგი $\sum_{i=0}^{n} \lambda_i \boldsymbol{P}_i$ არის წერტილი როცა $\sum_{i=0}^{n} \lambda_i = 1$ და არაა დამოკიდებული საკოორდინატო სისტემის არჩევაზე. ფიზიკიდან ამ შემთხვევის მაგალითია ე.წ. „მასათა ცენტრი“.

ზემოთ მოყვანილი ორივე თეორემა შეგვიძლია მარტივად დავამტკიცოთ.

$$\boldsymbol{P} = \sum_{i=0}^{n} \lambda_i \boldsymbol{P}_i$$

$$\overrightarrow{\boldsymbol{OP}} = \sum_{i=0}^{n} \lambda_i \overrightarrow{\boldsymbol{OP}_i}$$

ავიღოთ რაღაც სხვა წერტილი $\boldsymbol{O}'$ მაშინ

$$\overrightarrow{\boldsymbol{O'P}} = \sum_{i=1}^{n} \lambda_i \overrightarrow{\boldsymbol{O'O}} + \sum_{i=0}^{n} \lambda_i \overrightarrow{\boldsymbol{OP}_i} \tag{3.77}$$

თუ დავუშვებთ, რომ $\sum_{i=0}^{n} \lambda_i = 0$, მაშინ ბოლო ტოლობიდან მივიღებთ

$$\overrightarrow{\boldsymbol{O'P}} = \sum_{i=0}^{n} \lambda_i \overrightarrow{\boldsymbol{OP}_i} = \overrightarrow{\boldsymbol{OP}} \tag{3.78}$$

ანუ მივიღეთ რაღაც რაც არაა დამოკიდებული საკოორდინატო სათავეზე. ე.ი. მიღებული შედეგი ვექტორია(და არა რადიუს ვექტორი ანუ წერტილი)

თუ დავუშვებთ, რომ $\sum_{i=0}^{n} \lambda_i = 1$, მაშინ ტოლობიდან 3.77 მივიღებთ

$$\overrightarrow{\boldsymbol{O'P}} = \overrightarrow{\boldsymbol{O'O}} + \sum_{i=0}^{n} \lambda_i \overrightarrow{\boldsymbol{OP}_i} \tag{3.79}$$

$$\boldsymbol{O'} + \overrightarrow{\boldsymbol{O'P}} = \boldsymbol{O'} + \overrightarrow{\boldsymbol{O'O}} + \overrightarrow{\boldsymbol{OP}} = \boldsymbol{O} + \overrightarrow{\boldsymbol{OP}} \tag{3.80}$$

ე.ი. ეს წერტილი არ ყოფილა დამოკიდებული ათვლის წერტილის არჩევაზე.

3.14 ზოგადი აფინური გარდაქმნები

გავიხსენოთ რა არის წრფივი გარდაქმნა. წრფივი გარდაქმნისას გარდაიქმნება ბაზისი ხოლო კოორდინატების მნიშვნელობები ახალ ბაზისში არის იგივე რაც იყო ძველ ბაზისში.

აფინური გარდაქმნა არის ორი გარდაქმნის კომპოზიცია – წრფივი გარდაქმნა + გადატანა ($\widehat{\mathcal{A}} = \{\widehat{L}, \widehat{\mathrm{T}}_d\}$). ამიტომ გვექნება

$$\begin{aligned}
\widehat{\mathcal{A}}\boldsymbol{P} = \phi(\boldsymbol{P}) &= \vec{\boldsymbol{d}} + L(\boldsymbol{P}) \\
\widehat{\mathcal{A}}\boldsymbol{P} = \phi(\overrightarrow{\boldsymbol{OP}}) &= \overrightarrow{\boldsymbol{OO'}} + L(\overrightarrow{\boldsymbol{O'P}}) \\
\overrightarrow{\boldsymbol{OP'}} &= \overrightarrow{\boldsymbol{OO'}} + L(\overrightarrow{\boldsymbol{O'P}}) \\
\vec{\boldsymbol{f}}_{\mathcal{B}}[x'] &= \overrightarrow{\boldsymbol{OO'}}_{\mathcal{B}} + \vec{\boldsymbol{f}}_{\mathcal{B}'}[x]
\end{aligned}$$

კავშირი ძველ და ახალ საკოორდინატო ბაზისებს შორის წრფივი გარდაქმნისას გვაქვს

$$\vec{\boldsymbol{f}}_{\mathcal{B}'} = \vec{\boldsymbol{f}}_{\mathcal{B}}\mathbf{C} \tag{3.81}$$

ხოლო

$$\overrightarrow{\boldsymbol{OO'}}_{\mathcal{B}} = \vec{\boldsymbol{f}}_{\mathcal{B}}[d] \tag{3.82}$$

ამიტომ გვექნება

$$\vec{\boldsymbol{f}}_{\mathcal{B}}[x'] = \vec{\boldsymbol{f}}_{\mathcal{B}}[d] + \vec{\boldsymbol{f}}_{\mathcal{B}}\mathbf{C}[x]$$

ანუ კოორდინატებისთვის გვექნება

$$[x'] = [d] + \mathbf{C}[x] \tag{3.83}$$

მატრიცული სახით

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} + \mathbf{C} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{3.84}$$

აქედან ჩანს, რომ წრფივი განტოლებების სისტემის ამოხსნის ამოცანა აფინური გარდაქმნების ენაზე ყოფილა შემდეგი: ვიპოვნოთ $\overrightarrow{\boldsymbol{OP}}$ ისეთი, რომ მოცემული აფინური გარდაქმნით $\widehat{\mathcal{A}} \equiv \{\widehat{L}, \vec{\boldsymbol{d}}\}$ აისახებოდეს $\overrightarrow{\boldsymbol{OP}'}$ ვექტორში.

აფინური გარდაქმნის თვისებები:

- საკოორდინატო სათავე არაა აუცილებელი რომ დარჩეს ადგილზე.
- წრფე აისახება წრფეში.
- პარალელური წრფეები აისახება პარალელურ წრფეებში.
- მონაკვეთებს შორის პროპორციები არ იცვლება.
- სიბრტყე აისახება სიბრტყეში.
- პარალელური სიბრტყეები აისახება პარალელურ სიბრტყეებში.
- აფინური გარდაქმნების კომბინაცია არის აფინური გარდაქმნა.
- პარალელოგრამის ფართობი (პარალელეპიპედის მოცულობა) აფინური $\mathbf{C}$ გარდაქმნისას გარდაიქმნება როგორც

$$\begin{aligned} S' &= \det \mathbf{C} \cdot S \\ V' &= \det \mathbf{C} \cdot V \end{aligned} \tag{3.85}$$

- ფართობებს და მოცულობებს შორის პროპორციები არ იცვლება.

3.15 წერტილის და ვექტორის აფინური გარდაქმნა

ზემოთ მოყვანილი ზოგადი აფინური გარდაქმნა ჩავწეროთ როგორც

$$\widehat{\mathcal{A}}\overrightarrow{\boldsymbol{OX}} = \overrightarrow{\boldsymbol{OY}} = \widehat{L}\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}} \tag{3.86}$$

აფინური გარდაქმნა განმარტებული გვაქვს წერტილისთვის და რადგან აფინური გარდაქმნები ყოველთვის განმარტებული გვაქვს რაღაც საკოორდინატო სისტემაში $\overrightarrow{\boldsymbol{OX}} \equiv \vec{\boldsymbol{r}}$, $\overrightarrow{\boldsymbol{OX_1}} \equiv \vec{\boldsymbol{r}}_1$ და ა.შ. ($\vec{\boldsymbol{r}}_i$ წერტილის რადიუს ვექტორია).

ანუ მიღებული შედეგები ძალაშია წერტილებისთვის(რადიუს ვექტორებისთვის). შეგვიძლია განვიხილოთ ვექტორის აფინური გარდაქმნაც. ამ შემთხვევაში ვექტორი უნდა განვიხილოთ როგორც ორ წერტილს შორის სხვაობა($\boldsymbol{X}_2 = \boldsymbol{X}_1 + \vec{\boldsymbol{x}}$ რაც იგივეა სათავის არჩევის შემდეგ $\overrightarrow{\boldsymbol{OX}_2} \equiv \vec{\boldsymbol{r}}_2$, $\overrightarrow{\boldsymbol{OX}_1} \equiv \vec{\boldsymbol{r}}_1$ და $\overrightarrow{\boldsymbol{OX}_2} = \overrightarrow{\boldsymbol{OX}_1} + \vec{\boldsymbol{x}}$ სადაც $\overrightarrow{\boldsymbol{X_1X_2}} = \vec{\boldsymbol{r}}_2 - \vec{\boldsymbol{r}}_1 \equiv \vec{\boldsymbol{x}}$) და ვექტორების შემთხვევაში ტრანსლიაციური ნაწილი (გადატანა) ნულია. მართლაც

$$\widehat{\mathcal{A}}\overrightarrow{\boldsymbol{X_1X_2}} = \widehat{\mathcal{A}}(\overrightarrow{\boldsymbol{OX}_2} - \overrightarrow{\boldsymbol{OX}_1}) = \widehat{L}\overrightarrow{\boldsymbol{OX}_2} + \vec{\boldsymbol{d}} - (\widehat{L}\overrightarrow{\boldsymbol{OX}_1} + \vec{\boldsymbol{d}}) = \widehat{L}\overrightarrow{\boldsymbol{X_1X_2}} \tag{3.87}$$

ეს გასაგებიცაა. ვექტორში ყოველთვის ვგულისხმობთ თავისუფალ ვექტორს და თავისუფალი ვექტორი გადატანისას არ იცვლება. ამის მათემატიკური ასახვაა რაც მივიღეთ — აფინური გარდაქმნა ვექტორზე მოქმედებს მხოლოდ „წრფივი გარდაქმნის ნაწილით“ ანუ

$$\widehat{\mathcal{A}}\vec{\boldsymbol{x}} = \widehat{L}\vec{\boldsymbol{x}} \tag{3.88}$$

წერტილების შეკრების შემთხვევაში, როგორც ზემოთ ავღნიშნეთ აზრი აქვს მხოლოდ წერტილების აფინურ კომბინაციას, რაც გვაძლევს ან ვექტორს ან ბარიცენტრულ წერტილს.

ვექტორების ჯამისთვის გვექნება

$$\widehat{\mathcal{A}}(\vec{\boldsymbol{x}}_1 + \vec{\boldsymbol{x}}_2) = \widehat{L}(\vec{\boldsymbol{x}}_1 + \vec{\boldsymbol{x}}_2) \tag{3.89}$$

3.16 მრავალჯერადი აფინური გარდაქმნები

განვიხილოთ ორი აფინური გარდაქმნის კომპოზიცია

$$\widehat{\mathcal{A}}_1\boldsymbol{X} = \boldsymbol{X}_1 = \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_1 \tag{3.90}$$

და

$$\begin{aligned}\boldsymbol{X}_2 = \widehat{\mathcal{A}}_2\widehat{\mathcal{A}}_1\boldsymbol{X} = \widehat{\mathcal{A}}_2\boldsymbol{X}_1 = \widehat{\mathcal{A}}_2(\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_1) = \\ = \widehat{L}_2(\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_1) + \vec{\boldsymbol{d}}_2 = \\ = \widehat{L}_2\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_2\vec{\boldsymbol{d}}_1 + \vec{\boldsymbol{d}}_2\end{aligned}$$

3.17 შებრუნებული აფინური გარდაქმნა

რადგან აფინური გარდაქმნა არის

$$\widehat{\mathcal{A}}\boldsymbol{X} = \boldsymbol{Y} \equiv \overrightarrow{\boldsymbol{OY}} = \widehat{L}\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}} \tag{3.91}$$

შებრუნებული იქნება

$$\boldsymbol{X} = \widehat{\mathcal{A}}^{-1}\boldsymbol{Y} = \widehat{L}^{-1}(\overrightarrow{\boldsymbol{OY}} - \vec{\boldsymbol{d}}) \tag{3.92}$$

მართლაც, დავუბრუნდეთ მრავალჯერადი გარდაქმნების ფორმულას

$$\widehat{\mathcal{A}}_2\widehat{\mathcal{A}}_1\overrightarrow{\boldsymbol{OX}} = \widehat{L}_2\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_2\vec{\boldsymbol{d}}_1 + \vec{\boldsymbol{d}}_2 \tag{3.93}$$

დავუშვათ $\widehat{\mathcal{A}}_2$ არის $\widehat{\mathcal{A}}_1^{-1}$ მაშინ 3.92 და ზემოთ მოყვანილი განტოლების გამოყენებით გვექნება

$$\begin{aligned}\widehat{\mathcal{A}}_1^{-1}\widehat{\mathcal{A}}_1\overrightarrow{\boldsymbol{OX}} &= \widehat{\mathcal{A}}_1^{-1}(\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_1)\\ &= \widehat{L}_1^{-1}(\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_1 - \vec{\boldsymbol{d}}_1) = \widehat{L}_1^{-1}\widehat{L}_1\overrightarrow{\boldsymbol{OX}} = \overrightarrow{\boldsymbol{OX}} = \boldsymbol{X}\end{aligned} \tag{3.94}$$

ორი აფინური გარდაქმნის კომპოზიცია არის აფინური გარდაქმნა ზემოთ მიღებული შედეგი შგვიძლია ჩავწეროთ როგორც

$$\widehat{\mathcal{A}}_2\widehat{\mathcal{A}}_1\overrightarrow{\boldsymbol{OX}} = \widehat{\mathcal{A}}_3\{\widehat{L}_2\widehat{L}_1, \widehat{L}_2\vec{\boldsymbol{d}}_1 + \vec{\boldsymbol{d}}_2\}\overrightarrow{\boldsymbol{OX}} \tag{3.95}$$

თუ ისევ დავუშვებთ რომ $\widehat{\mathcal{A}}_2$ არის $\widehat{\mathcal{A}}_1^{-1}$ მაშინ $\widehat{\mathcal{A}}_3$ უნდა იყოს იგივური გარდაქმნა ანუ სრულდება პირობები

$$\begin{aligned}&\widehat{L}_2 = \widehat{L}_1^{-1}\\ &\widehat{L}_2\vec{\boldsymbol{d}}_1 + \vec{\boldsymbol{d}}_2 = 0 \Rightarrow \vec{\boldsymbol{d}}_2 = -\widehat{L}_1^{-1}\vec{\boldsymbol{d}}_1\end{aligned} \tag{3.96}$$

ე.ი.

$$\widehat{\mathcal{A}}_1^{-1} \equiv \widehat{\mathcal{A}}\{\widehat{L}_1^{-1}, -\widehat{L}_1^{-1}\vec{\boldsymbol{d}}_1\} \tag{3.97}$$

3.18 წრფივი გარდაქმნა და გადატანა

აფინური გარდაქმნის განმარტებიდან გამომდინარე $\boldsymbol{X}-$ზე ჯერ სრულდება წრფივი გარდაქმნა, ხოლო შემდეგ გადატანა. ზოგიერთ წიგნში აფინური გარდაქმნის აღსანიშნავად შეიძლება შეგვხვდეს აღნიშვნა $\widehat{\mathcal{A}}\boldsymbol{X} \equiv \phi(\boldsymbol{X}) \equiv \widehat{\mathrm{T}}_d\widehat{L}\overrightarrow{\boldsymbol{OX}}$. რაც ნიშნავს, რომ $\boldsymbol{OX}-$ზე ჯერ სრულდება წრფივი გარდაქმნა მატრიცით $\widehat{L}$ ხოლო შემდეგ ტრანსლიაცია (გადატანა) $\widehat{\mathrm{T}}_d$ ვექტორით $\vec{\boldsymbol{d}}$ (ანუ $\widehat{\mathrm{T}}_d\overrightarrow{\boldsymbol{OX}} = \overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}$).

ვანახოთ, რომ წრფივი გარდაქმნა და გადატანა არ კომუტირებენ (ანუ ოპერაციების თანმიმდევრობის შეცვლა არ გვაძლევს იგივე შედეგს). წარმოვიდგინოთ, გვაქვს ორი აფინური გარდაქმნა პირველში გვაქვს მხოლოდ წრფივი ნაწილი($\widehat{\mathcal{A}}_1 = \{\widehat{L}_1, \vec{\boldsymbol{d}}_1 = \vec{\boldsymbol{0}}\}$), ხოლო

მეორეში მხოლოდ გადატანა $\widehat{\mathcal{A}}_2 = \{\widehat{E}, \vec{\boldsymbol{d_2}}\}$($\widehat{E}$ აღნიშნავს იგივური წრფივი გარდაქმნას). მაშინ განტ. 3.93-ში გავითვალისწინოთ ზემოთ მოყვანილი $\widehat{\mathcal{A}}_1$ და $\widehat{\mathcal{A}}_1$ სახე და გვექნება

$$\begin{aligned}\widehat{\mathcal{A}}_2\widehat{\mathcal{A}}_1\overrightarrow{\boldsymbol{OX}} =& \widehat{L}_2\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_2\vec{\boldsymbol{d}}_1 + \vec{\boldsymbol{d}}_2\\ =& \widehat{E}\widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{E}\vec{\mathbf{0}} + \vec{\boldsymbol{d}}_2\\ =& \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_2\end{aligned} \tag{3.98}$$

შედეგში ახალი არაფერია. ჯერ წრფივი გარდაქმნა ხოლო შემდეგ გადატანა ეს არის აფინური გარდაქმნის განმარტება და სწორედ ესაა ზემოთ მიღებული ტოლობა. გადატანა/წრფივი გარდაქმნის ენაზე ეს შეგვიძლია ჩავწეროთ როგორც

$$\widehat{\mathrm{T}}_{d_2}\widehat{L}_1\overrightarrow{\boldsymbol{OX}} = \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_2 \tag{3.99}$$

ახლა ვნახოთ

$$\widehat{\mathcal{A}}_1\widehat{\mathcal{A}}_2\overrightarrow{\boldsymbol{OX}} \tag{3.100}$$

გვექნება

$$\begin{aligned}\widehat{\mathcal{A}}_1\widehat{\mathcal{A}}_2\overrightarrow{\boldsymbol{OX}} =& \widehat{L}_1\widehat{L}_2\overrightarrow{\boldsymbol{OX}} + \widehat{L}_1\vec{\boldsymbol{d}}_2 + \vec{\boldsymbol{d}}_1\\ =& \widehat{L}_1\widehat{E}\overrightarrow{\boldsymbol{OX}} + \widehat{L}_1\vec{\boldsymbol{d}}_2 + \vec{\mathbf{0}}\\ =& \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_1\vec{\boldsymbol{d}}_2\end{aligned}$$

ისევ გადატანა/წრფივი გარდაქმნის ენაზე ეს შეგვიძლია ჩავწეროთ როგორც

$$\widehat{L}_1\widehat{\mathrm{T}}_{d_2}\overrightarrow{\boldsymbol{OX}} = \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_1\vec{\boldsymbol{d}}_2 = \widehat{L}_1(\overrightarrow{\boldsymbol{OX}} + \vec{\boldsymbol{d}}_2) \tag{3.101}$$

აფინური გარდაქმნის კომპოზიცია ისევ არის აფინური გარდაქმნა. ახლა დავსვათ ამოცანა: ვიპოვნოთ ისეთი $\widehat{\mathcal{A}}_3$, რომ $\widehat{\mathcal{A}}_1\widehat{\mathcal{A}}_2\overrightarrow{\boldsymbol{OX}} = \widehat{L}_1\overrightarrow{\boldsymbol{OX}} + \widehat{L}_1\vec{\boldsymbol{d}}_2 = \widehat{\mathcal{A}}_3\overrightarrow{\boldsymbol{OX}}$ რადგან

$$\widehat{\mathcal{A}}_i\widehat{\mathcal{A}}_j = \widehat{\mathcal{A}}\{\widehat{L}_i\widehat{L}_j, \widehat{L}_i\vec{\boldsymbol{d}}_j + \vec{\boldsymbol{d}}_i\} \tag{3.102}$$

ადვილად ვაჩვენებთ, რომ ეს იქნება აფინური გარდაქმნა წრფივი ნაწილით $\widehat{L}_1$ და წანაცვლების ვექტორით $\widehat{L}_1\vec{\boldsymbol{d}}_2$

$$\widehat{\mathcal{A}}_3 \equiv \widehat{\mathcal{A}}_1\widehat{\mathcal{A}}_2 = \widehat{\mathcal{A}}_3\{\widehat{L}_1\widehat{L}_2, \widehat{L}_1\vec{\boldsymbol{d}}_2 + \vec{\boldsymbol{d}}_1\} = \widehat{\mathcal{A}}_1\{\widehat{L}_1, \vec{\mathbf{0}}\}\widehat{\mathcal{A}}_2\{\widehat{E}, \vec{\boldsymbol{d_2}}\} = \widehat{\mathcal{A}}_3\{\widehat{L}_1, \widehat{L}_1\vec{\boldsymbol{d}}_2\} \tag{3.103}$$

მომავალში შეიძლება დაგვჭირდეს რთულ ტოლობაში $\widehat{\mathrm{T}}_{d_2}\widehat{L}_1$ ტიპის ნამრავლში ოპერაციების გადანაცვლება. შეგვიძლია ვნახოთ რომ

$$\widehat{\mathrm{T}}_{d_2}\widehat{L}_1 = \widehat{L}_1\widehat{\mathrm{T}}_{\widehat{L}_1^{-1}d_2} \tag{3.104}$$

და

$$\widehat{L}_1\widehat{\mathrm{T}}_{d_2} = \widehat{\mathrm{T}}_{\widehat{L}_1 d_2}\widehat{L}_1 \tag{3.105}$$

თავი 4

OpenGL შესავალი

კომპიუტერული გრაფიკა მნიშვნელოვან როლს თამაშობს ისეთ სხვადასხვა დარგებში როგორიცაა არქიტექტურა CAD(Computer Aided Design), ვიზუალიზაცია მედიცინაში, თამაშების და ფილმების ინდუსტრია, სამეცნიერო კვლევების და შედეგების ვიზუალიზაცია, ინფორმაციის ვიზუალიზაცია.

ზოგადი დანიშნულების პროგრამირების ენები უზრუნველყოფენ რაღაც დონით გრაფიკულ ფუნქციებს და შესაძლებლობებს, მაგრამ არსებობენ სპეციალიზირებული გრაფიკული ბიბლიოთეკები GL, OpenGL, VRML, Java2D, Java3D. გრაფიკული ფუნქციების ერთობლიობას ხშირად უწოდებენ კომპიუტერული გრაფიკის პროგრამული უზრუნველყოფის პროგრამირების ინტერფეისს (Computer Graphics Application Programming Interface (CG API)).

4.1 OpenGL პრიმიტივები

ცხრილი 4.1: **OpenGL** ძირითადი ფუნქციები

1. OpenGL პრიმიტივები	
1.1. `glBegin('არგუმენტი')` და `glEnd()`	
`'არგუმენტი'`	აღწერა
გრძელდება შემდეგ გვერდზე	

GL_POINTS	Treats each vertex as a single point. Vertex n defines point n. N points are drawn.
GL_LINES	Treats each pair of vertices as an independent line segment. Vertices 2n - 1 and 2n define line n. N/2 lines are drawn.
GL_LINE_STRIP	Draws a connected group of line segments from the first vertex to the last. Vertices n and n+1 define line n. N - 1 lines are drawn.
GL_LINE_LOOP	Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n + 1 define line n. The last line, however, is defined by vertices N and 1. N lines are drawn.
GL_TRIANGLES	Treats each triplet of vertices as an independent triangle. Vertices 3n - 2, 3n - 1, and 3n define triangle n. N/3 triangles are drawn.
GL_TRIANGLE_STRIP	Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n, vertices n, n + 1, and n + 2 define triangle n. For even n, vertices n + 1, n, and n + 2 define triangle n. N - 2 triangles are drawn.
GL_TRIANGLE_FAN	Draws a connected group of triangles. one triangle is defined for each vertex presented after the first two vertices. Vertices 1, n + 1, n + 2 define triangle n. N - 2 triangles are drawn.
GL_QUADS	Treats each group of four vertices as an independent quadrilateral. Vertices 4n - 3, 4n - 2, 4n - 1, and 4n define quadrilateral n. N/4 quadrilaterals are drawn.

გრძელდება შემდეგ გვერდზე

GL_QUAD_STRIP	Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2n - 1, 2n, 2n + 2, and 2n + 1 define quadrilateral n. N/2 - 1 quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.
GL_POLYGON	Draws a single, convex polygon. Vertices 1 through N define this polygon.

1.2. glBegin/glEnd შორის შეიძლება გამოყენებულ იქნას ბრძანებები:

. gl Vertex 2 f ('არგუმენტები')	⚪ აღნიშნავს რომ გვაქვს ფუნქცია ეკუთვნის **OpenGL** ბიბლიოთეკას ⚪ ფუნქციის სახელი ⚪ არგუმენტების რაოდენობა შეიძლება იყოს {2,3,4} ამ სპეციფიკატორი შემდეგ შეიძლება იყოს ასევე v რაც ნიშნავს, რომ არგუმენტი უნდა იყოს მასივი. ⚪ რა ტიპის არგუმენტი შეიძლება იყოს. შეიძლება იყოს {f,d,i} ⚪ 'არგუმენტები' აქ გადაეცემა არგუმენტების ჩამოწათვალი(ან მასივი), რაც შეესაბამება მოცემულ ფუნქციას იგივე ეხება ქვემოთ მოყვანილ ფუნქციებსაც.
glNormal	ნორმალის გადაცემა
glColor	ფერი
glMaterial glTexCoord	ნივთიერება
glEvalCoord	
glEvalPoint	
glMaterial	
glEdgeFlag	
glCallList ან glCallLists	წინასწარ შექმნილი glCallList გადაცემა დასახატად.

თუ გადაეცემა ფუნქცია, რომელიც არ იცის glBegin-glEnd მაშინ შეცდომა ჩაიწერება ლოგ ში და ფუნქცია იგნორირდება

გრძელდება შემდეგ გვერდზე

glEdgeFlag

4.2 2D გარდაქმნები

გარდაქმნები შეიძლება იყოს შემდეგი:

1. გადატანა (ტრანსლიაცია)
2. მობრუნება
3. არეკვლა
4. სკალირება
5. ძვრა

გადატანა

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix} \tag{4.1}$$

ერთგვაროვანი სკალირება:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \alpha \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \alpha x \\ \alpha y \end{pmatrix} \tag{4.2}$$

არაერთგვაროვანი სკალირება:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha x \\ \beta y \end{pmatrix} \tag{4.3}$$

ძვრა

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} h_x y \\ h_y x \end{pmatrix} \tag{4.4}$$

როგორც ვხედავთ გადატანისას თითოეულ კოორდინატს ემატება რიცხვი. სკალირებისას მრავლდება რიცხვზე და მობრუნებისას კოორდინატები მრავლდება მატრიცაზე. ანუ თუ გვექნება რაიმე გეომეტრიული ობიექტი და ობიექტი რომელსაც შეგვიძლია ვუწოდოთ გარდაქმნა მოცემული ობიექტს უნდა ჰქონდეს გარდაქმნის ობიექტიც. გარდაქმნის ობიექტში უნდა იყოს სკალირების, მობრუნების და გადატანის წევრები და ფუნქციები.

მაგრამ რა ხდება როცა რამდენჯერმე ვასრულებთ გადატანას, მობრუნებას, სკალირებას ან ამ გარდაქმნების ნებისმიერ კომბინაციას?

გადატანა, სკალირების შემთხვევაში ადვილია, უბრალოდ შევცვლით შესაბამის მნიშვნელობებს. მობრუნების შემთხვევაშიც ყოველთვის შეიძლება ავაგოთ ახალი მატრიცა მოცემული ახალი კუთხისთვის, თუ წინასწარ ვიცით ეს კუთხე. მაგრამ არის შემთხვევები როცა ობიექტი გადის გარდაქმნის რამდენიმე ეტაპს და საჭიროა არა მარტო პირველადი მდგომარეობის ცოდნა არამედ ყველა ეტაპზე მდგომარეობის ცოდნაც. კითხვაზე პასუხი შემდეგია: მობრუნებების შემთხვევაში მრავალჯერადი მობრუნებისას შესაბამისი მატრიცები უნდა გამრავლდეს. მაგ. თუ $\mathbf{Q}_1(\alpha_1)$,$\mathbf{Q}_2(\alpha_2)$,$\mathbf{Q}_3(-\alpha_3)$ მობრუნებებია მიღებული შედეგი იქნება:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{Q}_3(-\alpha_3)\mathbf{Q}_2(\alpha_2)\mathbf{Q}_1(\alpha_1) \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.5}$$

ანუ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{Q}(\beta) \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.6}$$

სადაც

$$\mathbf{Q}(\beta) = \mathbf{Q}_3(-\alpha_3)\mathbf{Q}_2(\alpha_2)\mathbf{Q}_1(\alpha_1) \tag{4.7}$$

მობრუნებების მატრიცების ნამრავლია. ანუ ჩვენ შეგვიძლია რამდენჯერმე კი არ გარდავქმნათ ობიექტი არამედ ავაგოთ ერთი მობრუნების მატრიცა რომელზეც გამრავლება მოგვცემს საჭირო შედეგს.

მაგრამ ისევ ცალკე გვაქვს გადატანები და სკალირებები. თუკი რომელიმე მობრუნებებს შორის მოხდა გადატანა და ან სკალირება ესეც გათვალისწინებული უნდა იყოს. მაგ. თუ პირველი და მეორე მობრუნების მერე მოხდა გადატანა T_1 და T_2 მაშინ:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{Q}_3(-\alpha_3)\mathrm{T}_2\mathbf{Q}_2(\alpha_2)\mathrm{T}_1\mathbf{Q}_1(\alpha_1) \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.8}$$

და ყველაფერი სწორია, მაგრამ ასეთი მიდგომით შეუძლებელი იქნება გარდაქმნების (უფრო სწორედ საბოლოო შედეგის) ერთი მატრიცით წარმოდგენა.

თუმცა ასეთი მიდგომაც სწორია. მაგალითად თუ წერთ გეომეტრიის რედაქტორს მოცემული ობიექტს ექნება შესაბამისი მობრუნების, სკალირების, გადატანის წევრები. თქვენ ცვლით ამ წევრებს და ყოველი შეცვლისას ახატინებთ გრაფიკულ ძრავას ობიექტს (თუმცა ამ

მიდგომის დროსაც, როცა გამოიყენებთ **OpenGL** ბრძანებებს, ძრავა თავად აკეთებს საბოლოო მატრიცას რომელზე გამრავლდება წერტილის კოორდინატები).

ასევე გასათვალისწინებელია გარდაქმნების თანმიმდევრობა. ობიექტის რედაქტირებისას ხშირად იღებენ ასეთ თანმიმდევრობას:

1. სკალირება
2. მობრუნება
3. გადატანა

ანუ თუ ჩავთვლით რომ რედაქტორში შექმნილი ან გარე ფაილიდან ჩატვირთული ობიექტი არის კოორდინატის სათავეში, ჯერ ხდება ამ ობიექტის სკალირება, შემდეგ მობრუნება და მხოლოდ ამის შემდეგ გადატანა.

მოცემული მიდგომა არ იძლევა საშუალებას, მაგალითად ჯერ მოვახდინოთ გადატანა და შემდეგ მობრუნება მოცემული საკოორდინატო ღერძის/ან ღერძების გარშემო. ცხადია იგივე ეფექტი შეიძლება მიღწეულ იქნას ზემოთ მოყვანილი მეთოდითაც

ანუ შეიძლება მოინახოს ისეთი გადატანა და მობრუნება რომ ჯერ მოვაბრუნოთ ობიექტი საკოორდინატო სათავეში და შემდეგ გადავიტანოთ სასურველ ადგილას.

მაგრამ არსებობს სხვა მიდგომაც რომელიც საშუალებას იძლევა განვიხილოთ ობიექტის გარდაქმნა ერთი გარდაქმნის მატრიცის საშუალებით. ამ მატრიცაში „შენახულია“ როგორც მობრუნება ასევე გადატანაც და სკალირებაც.

4.3 2D ერთგვაროვანი კოორდინატები. აფინური სივრცის ჩადება 3D სივრცეში.

ვთქვათ გარდაქმნა მოიცემა შემდეგი სახით:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.9}$$

მაშინ თუ გარდაქმნის მატრიცის ელემენტებს ჩავწერთ როგორც მობრუნების ელემენტებს, გვექნება მობრუნების მატრიცა.

თუ გვსურს სკალირება მაშინ:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.10}$$

ძვრა

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & sh_x \\ sh_y & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.11}$$

არეკვლა Y მიმართ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.12}$$

არეკვლა (0,0) მიმართ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.13}$$

ანუ მობრუნება, სკალირება, ძვრა შეიძლება გვქონდეს ერთ მატრიცაში.

მაგრამ შეუძლებელია 2D გადატანა მოცემულ იყოს როგორც $2 \bigotimes 2$ მატრიცაზე გამრავლება:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix} \tag{4.14}$$

იმისათვის რომ გადატანაც ჩართული იყოს გარდაქმნის მატრიცაში და წარმოდგენილ იქნას როგორც მატრიცაზე გამრავლება შემოაქვთ ე.წ. ერთგვაროვანი კოორდინატები.

ვუმატებთ სივრცეს კიდევ ერთ განზომილებას მესამე კოორდინატით w (ანუ 2D შემთხვევისათვის იქნება 3 განზომილება) მაშინ მოცემული წერტილი შეგვიძლია წარმოვიდგინოთ როგორც

$$[wx, wy, w]^{\mathrm{T}}, w \neq 0$$

ან თუ დავანორმირებთ

$$[x, y, 1]^{\mathrm{T}}, w = 1$$

ანუ 2D (x, y) წერტილი ერთგვაროვან 3D კოორდინატებში ესაა წრფე რომელიც გადის საკოორდინატო სისტემის სათავეზე, მაგრამ არ შეიცავს მას (პირობა $w \neq 0$)

ერთგვაროვან კოორდინატებში გარდაქმნა შეიძლება ჩაიწეროს შემდეგი სახით:

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix} \tag{4.15}$$

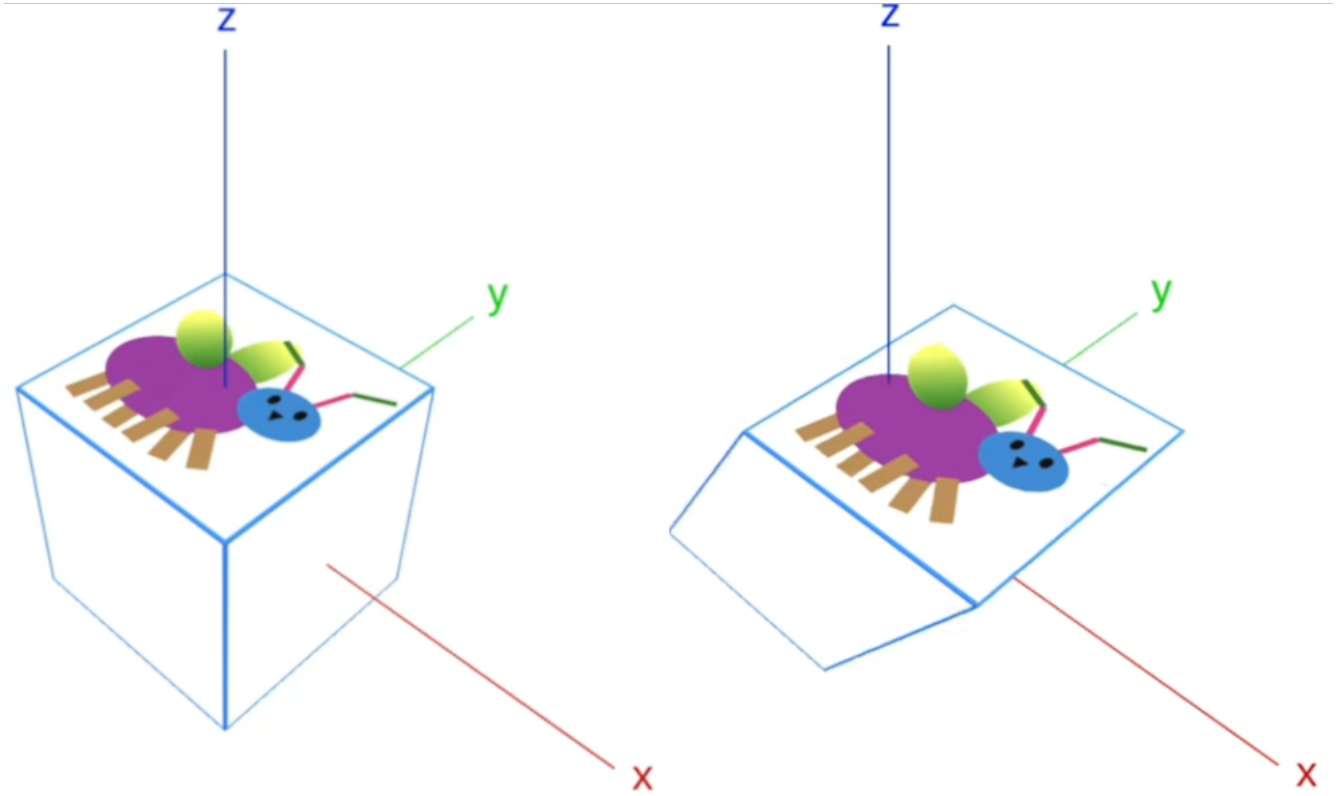
ადვილად შეიძლება შემოწმდეს რომ მატრიცა

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

გვაძლევს გადატანას x გასწვრივ ფორმ.(4.14)

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.16}$$

მაგრამ რატომ მოგვცა ერთი კოორდინატის დამატებამ გადატანა? ზემოთ მოყვანილი მატრიცა არის ძვრა 3D შემთხვევაში სურ. 4.2



სურ 4.2 გადატანა 2D, როგორც ძვრა 3D

თუ არ დავხატავთ კუბს და დავხატავდით მხოლოდ 2D სურათს და დავზედავთ სივრცეს ზემოდან დავინახავთ რომ სურათი წანაცვლდა x გასწვრივ.

ასევე მნიშვნელოვანია რომ მესამე კოორდინატი ტოლია ერთისა. ეს ნიშნავს, რომ ორგანზომილებიანი სივრცე ჩადებულია 3D სივრცეში და წარმოადგენს z=1 სიბრტყეს. ნებისმიერი წრფივი გარდაქმნა რომლის შედეგად გარადქმნილი ობიექტი დარჩება z=1 სიბრტყეზე იქნება 2D აფინური გარდაქმნა ამ სიბრტყისთვის.

y გასვრივ გადატანას მოგვცემს მატრიცა

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.17}$$

ხოლო სკალირებას მოგვცემს მატრიცა

$$\mathbf{S} = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.18}$$

შესაბამისად α კუთხეზე მობრუნებას მოგვცემს მატრიცა

$$\mathbf{Q}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

მობრუნებას ექნება სახე:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.19}$$

ანუ ზოგადად გვაქვს მატრიცა

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \tag{4.20}$$

სადაც $a, e > 1$ იძლევა სკალირებას, ხოლოდ c, f გადატანას.

რაც შეეხება b, d იძლევა x ძვრას y და y ძვრას x მიმართ.

ერთგვაროვან კოორდინატებში x ძვრას (ინგლისურად უწოდებენ „sharing“) გვაძლევს მატრიცა

$$\mathbf{H}_x = \begin{pmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

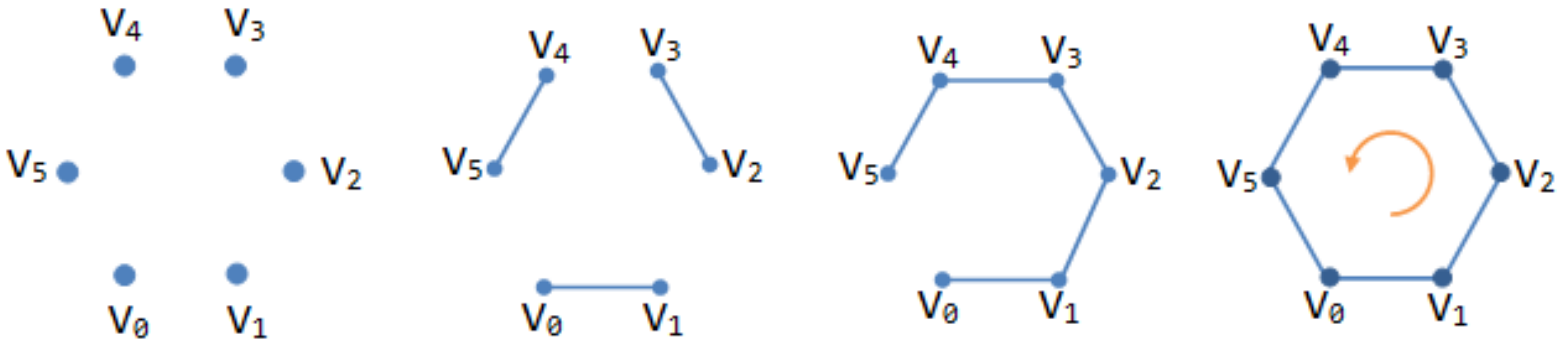
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x + sh_x y \\ y \\ 1 \end{pmatrix}$$

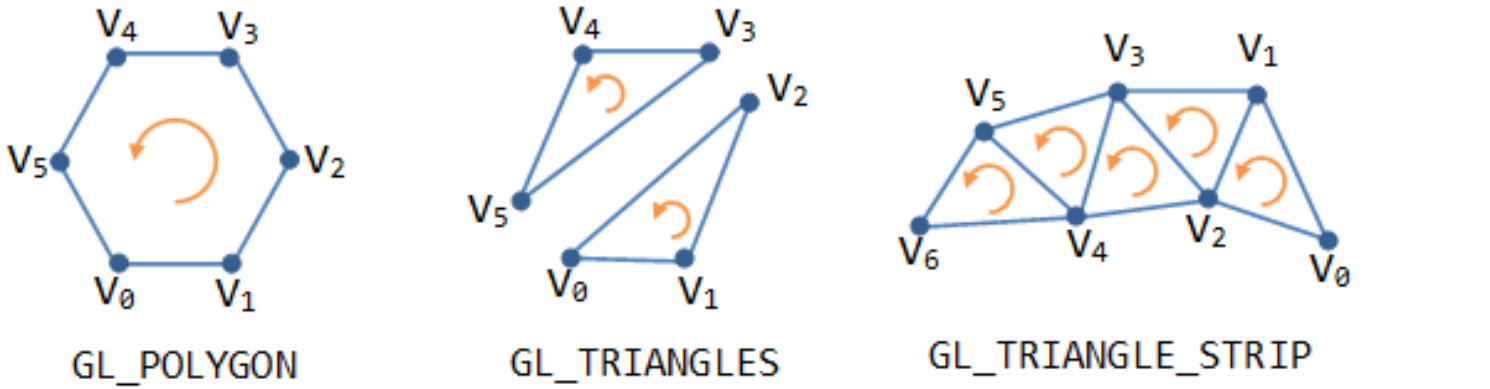
ხოლო y გასწვრივ

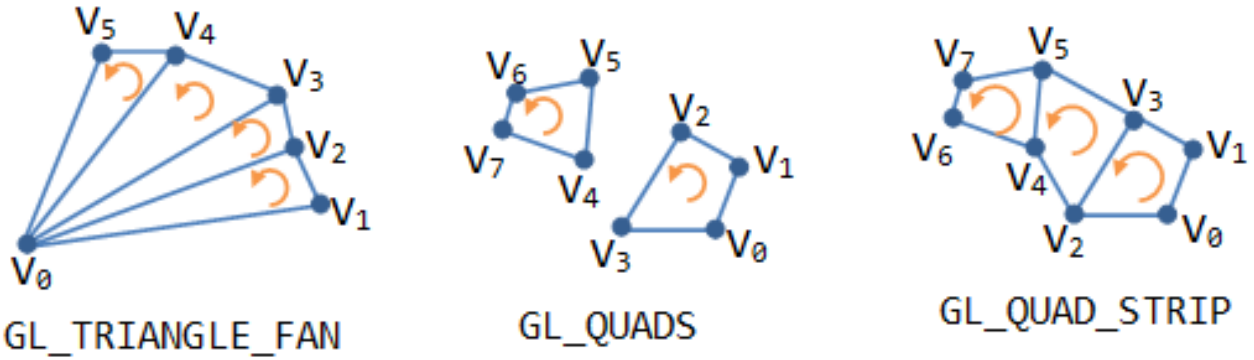
$$\mathbf{H}_y = \begin{pmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y + sh_y x \\ 1 \end{pmatrix}$$

4.4 ხედვის პორტში ასახვა


V4 V3
V5 V2
V0 V1
GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP


GL_POLYGON
GL_TRIANGLES
V6 V5 V4 V3 V2 V1 V0
GL_TRIANGLE_STRIP


GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP

ლიტერატურა

[1] N. V. Yefimov. *Quadratic Forms and Matrices*. Academic Press Inc., 1 edition, 1964. :Ефимов Н.В. - Квадратичные формы и матрицы (2012, ФИЗМАТЛИТ).

[2] ნ. მუსხელიშვილი. *ანალიზური გეომეტრიის კურსი*. საბჭოთა საქართველო, თბილისი. მე-4 გამ., 1962.

[3] M. A. Akivis and V. V. Goldberg. *Tensor Calculus with Applications*. World Scientific, 3 edition, 2003. :Акивис М. А., Гольдберг В. В. - Тензорное исчисление-Физматлит (2005).

[4] გ. ჭილაშვილი. *ვექტორული და ტენზორული ალრიცხვის ელემენტები*. თ.ს.უ., თბილისი, 1982.

[5] M. Pharr, W. Jakob, and G. Humphreys. Physically Based Rendering: From Theory to Implementation, 2023. URL `https://pbr-book.org/4ed/Geometry_and_Transformations/Applying_Transformations#Normals`.

[6] H. Weyl. *Space-Time-Matter*. Dover pub., 4 edition, 1952. : Вейль Г.(Weyl H.) - Пространство. Время. Материя (2004).

[7] M. Berger. *Geometry I*. Springer, 1987.

[8] A. O. Remizov I. R. Shafarevich. *Linear Algebra and Geometry*. Springer, 2013. :Шафаревич И.Р., Ремизов А.О. - Линейная алгебра и геометрия (2009, Физматлит).

[9] J. Gallier. *Geometric Methods and Applications for Computer Science and Engineerin*. Springer New York, NY, 2 edition, 2013. URL `https://www.cis.upenn.edu/~jean/gbooks/geom2-v2.html`.

[10] OpenGL. *OpenGL FAQ*. URL `https://www.opengl.org/archives/resources/faq/technical/transformations.htm`.

[11] M. Segal and K. Akeley. *The OpenGL™graphics System: A Specification (Version 2.1)*. 2006. URL `https://registry.khronos.org/OpenGL/specs/gl/glspec21.pdf`.

[12] Hearn D. D.,, Baker P.,, and Carithers Warren,. *Computer Graphics with OpenGL*. Pearson New International Edition, 4th edition, New York – London, 2013.