ГУАП

КАФЕДРА 14

КУРСОВАЯ РАБОТА (ПРОЕКТ)
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

доц., канд. техн. наук		А.В. Шахомиров
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

Элементарные преобразования трехмерных объектов с удалением невидимых линий и поверхностей и построением теней от заданного источника света

по дисциплине: Компьютерная графика

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 1042		Н.В. Корзун
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2022

Оглавление

1. Постановка задачи

1) Множество многогранников пространственной сцены: Параллелепипед и призма
2) Алгоритм закраски: Закраска линиями
3) Алгоритм удаления невидимых линий и поверхностей: Алгоритм “Художника”
4) Алгоритм построения тени: Построение тени на землю (источник света на конечном расстоянии)

2. Формализация

Данный проект писался, основываясь на различной литературе по линейной алгебре и геометрии.
Каждая фигура задается в нормализованных координатах, которые позволяют создать некий базовый шаблон фигуры, умножение на который любого числа, в дальнейшем, позволит получить данную фигуру в любом размере.
Отрисовка фигур происходит с помощью средств SDL.
Управление происходит клавишами указанными в консоли при запуске программы:

- W – движение объекта по оси Y в отрицательную сторону
- S – движение объекта по оси Y в положительную сторону
- A – движение объекта по оси X в отрицательную сторону
- D – движение объекта по оси X в положительную сторону
- Q – движение объекта по оси Z в отрицательную сторону
- E – движение объекта по оси Z в положительную сторону

- Стрелка вверх– поворот по оси Y в отрицательную сторону относительно центра объекта
- Стрелка вверх – поворот по оси Y в положительную сторону относительно центра объекта
- Стрелка вверх – поворот по оси X в отрицательную сторону относительно центра объекта
- Стрелка вверх – поворот по оси X в положительную сторону относительно центра объекта
- Z – поворот по оси Z в отрицательную сторону относительно центра фигуры
- X – поворот по оси Z в положительную сторону относительно центра объекта

- \+ - увеличение объекта
- \- - уменьшение объекта

  Коэффициенты и параметры сдвига, увеличения и углы поворота задаются для каждого объекта отдельно, но для простоты разработки, они задаются в коде при инициализации объекта.

- L – включить/выключить флаг управления светом
- P – включить/выключить флаг управления плоскостью, работает при включенном управлении светом.
- Цифры используются для выбора номера объекта.

В программе можно отрисовывать тени на несколько плоскостей, также плоскости можно двигать и крутить, плоскости задаются 3 точками.
Источник света можно двигать.

Фигур в программе может быть произвольное количество, они задаются в одной функции – add_figures().

Фигура состоит из точек, а грани задаются треугольниками, так чтобы описать параллелепипед нужно восемь точек и 12 полигонов – по два на грань.

Тени строятся с большим числом допущений, но это не мешает продемонстрировать их работу. Допущение состоит в следующем: предполагается, что полигон тень которого строится лежит между источником света, и плоскостью на которую строится тень. Добавление полного построения теней планируется в одном из будущих обновлений проекта: *https://github.com/nikoloskorzun/KG1*. Вычисления для построения всех вариантов теней выполнены, так например описан(в комментариях в исходном коде) способ решения проблемы бесконечных теней.

Ввиду примитивности и «млопригодности» заданного алгоритма отрисовки невидимых поверхностей, в программе исбользуется кабинетная проекция при отрисовке с закраской и изометрическая проекция при отрисовке «скелетов» фигур, в случае изометрической проекции происходит отрисовка только фигур.

Алгоритм художника используется везде даже при отрисовки плоскостей (у них для этого есть 3 заданных точки).

Работа с памятью осуществляется динамически.

Архитектура проекта пыталась разрабатываться так, чтобы быть легко модифицируемой, для этого есть множество инструментов. Но ввиду желания упростить себе работу некоторые из этих инструментов не используются. Например множественное использование числовых констант в функции add_figures класса Screen.

3. Листинг программы

3.1. Source.cpp

```
#include <iostream>
#include <SDL.h>
//#include "template_functions.h"
#define DEBUG 1
#include "screen.h"
using namespace std;
/*
Вариант 9.
Параллелепипед и призма
Построчное заполнение
Алгоритм «художника»
Построение «на землю» (источник света на конечном расстоянии)
*/
/*
Плоскости (произвольное количество), на которые падает тени.
Фигуры (произвольное количество):
Параллелепипед      F[0]
Треугольная призма  F[1]
Истоник света (один) L
Плоскость на которую проецируем тени P[0]
Объекты P[], L, F[] могут свободно двигаться
*/
/*
* P - плоскость
* L - источник света
* T - треугольный полигон: t1, t2, t3 - точки
*
* Ki - точка теневого полигона
тень - многоугольник - 3х или 4-х.
Так как тени могут быть бесконечными, то введем R = 1000 допустим - это будет
максимальная длинна бесконечной тени,
{1. Идеальный
    L---Ti---P или P---Ti---L
    Пересечение этой прямой с P - искомая точка Ki
2. Источник света между плоскостью и точкой полигона
    Ti---L---P или P---L---Ti
    Тень не создается для этой точки.
3. Плоскость между источником света и точкой полигона
    L---P---Ti или T---P---L
    Тень не создается для этой точки.
4. Плоскость параллельна прямой между источником света и точкой полигона
    L---Ti не пересекает P}
Все возможные случаи:
их 20.
T1T2T3
Хорошие варианты:
111Если для всех 3-х точек случай 1, то теневой полигон K можно строить
222Если для всех 3-х точек случай 2, то тени нет
333Если для всех 3-х точек случай 3, то тени нет
444Если для всех 3-х точек случай 4, то тени нет
Теперь мерзость:
11 2:
     Здесь возникает бесконечно длинная тень.
     12K1 K2
     x = точка пересечения прямой 3-L с плоскостью
     прямые x-K1 и x-K2 - образуют контур бесконечной тени
     3 берем точку но прямой x-K1 удаленную(в противоположную сторону от x) от K1 на R
     4 берем точку но прямой x-K2 удаленную(в противоположную сторону от x) от K2 на R
11 3:
    Нужно строить тень 4-х угольника
    12 K1 K2
```

    34 точки пересечения прямой 1-3 и плоскости и 2-3 и плоскости
11 4:
     Здесь возникает бесконечно длинная тень.
     12K1 K2
     x = прямая 3-L
     параллельно переносим x в K1 и K2
     34 также как от L к 3 в ту же сторону откладывать R
1 22:
     Здесь возникает бесконечно длинная тень.
     1 K1
     x1 = точка пересечения прямой 2-L с плоскостью
     x2 = точка пересечения прямой 3-L с плоскостью
     2 на прямой x1-K1 откладываем точку удаленную от x1 (в противоположную сторону от K1) на R
     3 на прямой x2-K1 откладываем точку удаленную от x2 (в противоположную сторону от K1) на R
3 22:
     Здесь возникает бесконечно длинная тень.

     x1 = точка пересечения прямой 2-L с плоскостью
     x2 = точка пересечения прямой 3-L с плоскостью

     x3 = точка пересечения прямой 1-L с плоскостью
     1 точка пересечения прямой 2-1 с плоскостью
     2 точка пересечения прямой 3-1 с плоскостью
     3 на прямой x1-x3 откладываем точку удаленную от x1 (в противоположную сторону от x3) на R
     4 на прямой x2-x3 откладываем точку удаленную от x2 (в противоположную сторону от x3) на R
4 22:
    Повезло тень не создается
1 33:
     1 K1
     2 точка пересечения прямой 2-1 с плоскостью
     3 точка пересечения прямой 3-1 с плоскостью
2 33:
     x1 = точка пересечения прямой 1-L с плоскостью
     1 x2 = точка пересечения прямой 2-1 с плоскостью
     2 x3 = точка пересечения прямой 3-1 с плоскостью
     3 на прямой x1-x2 откладываем точку удаленную от x2 (в противоположную сторону от x1) на R
     4 на прямой x1-x3 откладываем точку удаленную от x3 (в противоположную сторону от x1) на R
4 33:
     x1 = прямая 1-L
     1 x2 = точка пересечения прямой 2-1 с плоскостью
     2 x3 = точка пересечения прямой 3-1 с плоскостью
     параллельно переносим x1 в x2 и x3
     34 также как от L к 1 в ту же сторону откладывать R
1 44:
     1 K1
     x1 = прямая l-2
     x2 = прямая l-3
     параллельно переносим x1 и x2 в K1
     2 также как от L к 2 в ту же сторону откладывать R
     3 также как от L к 3 в ту же сторону откладывать R
2 44:

        Повезло тень не создается
3 44:
     1 точка пересечения 1-2 и плоскости
     2 точка пересечения 1-3 и плоскости
     x3 = точка пересечения l и 1
     x1 = прямая l-2
     x2 = прямая l-3

```
     параллельно переносим x1 и x2 в x3
     3 также как от L к 2 в ту же сторону откладывать R
     4 также как от L к 3 в ту же сторону откладывать R
123:
     1 K1
     2 x1 = пересечение прямой 2-3 и плоскости
     3 x2 = пересечение прямой 1-3 и плоскости

     x3 = точка пересечения l-2 и плоскости
     4 на прямой x3-K1 откладываем точку удаленную от K1 (в противоположную сторону от
x3) на R
     5 на прямой x3-x1 откладываем точку удаленную от x1 (в противоположную сторону от
x3) на R
124:

     1 K1
     x3 = пересечение 1-2 и плоскости через L параллельной Плоскости
     x1 = прямая l-x3
     x2 = прямая l-3
     параллельно переносим x1 и x2 в K1
     2 также как от L к x3 в ту же сторону откладывать R
     3 также как от L к 3 в ту же сторону откладывать R

134:
     1 точка пересечения 1-2 и плоскости
     2 x2 = точка пересечения 3-2 и плоскости
     3 K1
     x1 = прямая l-3
     параллельно переносим x1 в x2
     4 также как от L к 3 в ту же сторону откладывать R
234:
     1 x3 = точка пересечения 1-2 и плоскости
     2 x2 = точка пересечения 3-2 и плоскости
     x4 = точка (прямая 1-l пересечение с плоскостью)

    3 на прямой x3-x4 откладываем точку удаленную от x3 (в противоположную сторону от x4)
на R
     x1 = прямая l-3
     параллельно переносим x1 в x2
     4 также как от L к 3 в ту же сторону откладывать R
*/
int main(int argc, char* argv[])
{
    srand(1);
    Screen s(800, 800);
    s.add_figures();
    s.cycle();
    return 0;

}
```

3.2. screen.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "figure.h"
#include "light_system.h"
#include "polygone.h"
#include "shadow_polygone.h"
#include "template_functions.h"
/*
*/
using namespace std;
class Screen
```

```
{
public:

    Screen(uint32_t width, uint32_t height);
    void add_figures();
    int cycle();
    ~Screen();

private:
    SDL_Window* win;
    SDL_Renderer* ren = NULL;
    uint32_t width = NULL;
    uint32_t height = NULL;
    Figure* figures = NULL;
    Light_system* light_system; // она одна
    Base_polygone** polygones;
    Polygone** polygones_for_shadow;
    Shadow_polygone** shadow_polygones_for_shadow;
    //Polygone* polygones;
    //Shadow_polygone* shadow_polygones;
    size_t polygone_count;

    void draw_poligones(SDL_Renderer* ren);
    void painter_algorithm();

};
```

3.3. screen.cpp

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "screen.h"
#include "figure.h"
#include "polygone.h"
#include "template_functions.cpp"
using namespace std;
static int double_compare(const void* p1, const void* p2)
{
   double a = (*((Base_polygone**)(p1)))->z;
   double b = (*((Base_polygone**)(p2)))->z;
   //cout << a<<" " << b << endl;

   if (a < b) return -1;
   if (a == b) return 0;
   return 1;

}
Screen::Screen(uint32_t width, uint32_t height) {
   cout << "START!\n";
   if (SDL_Init(SDL_INIT_VIDEO) < 0) {
      SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "Couldn't initialize SDL: %s",
SDL_GetError());
      return;
   }
   if (SDL_CreateWindowAndRenderer(width, height, SDL_WINDOW_RESIZABLE, &win,
&ren)) {
```

```
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "Couldn't create window and
renderer: %s", SDL_GetError());
        return;
    }
    this->width = width;
    this->height = height;
}
void Screen::add_figures()//это функция говна, ее можно переписать и сделать
инициализацию фигур нормальной, но зачем?
{
    this->polygones = new Base_polygone * [(12 + 8) * 2 + 1 + 1];
    this->polygone_count = (12 + 8) * 2 + 1 + 1;
    polygones_for_shadow = new Polygone * [12 + 8];
    shadow_polygones_for_shadow = new Shadow_polygone * [12 + 8];
    size_t i = 0;
    this->figures = new Figure[2];
    size_t** i_ = allocate_memory_for_N_M_array<size_t>(12, 3);
    i_[0][0] = 0;
    i_[0][1] = 1;
    i_[0][2] = 3;
    i_[1][0] = 0;
    i_[1][1] = 2;
    i_[1][2] = 3;
    i_[2][0] = 4;
    i_[2][1] = 5;
    i_[2][2] = 7;
    i_[3][0] = 4;
    i_[3][1] = 6;
    i_[3][2] = 7;
    //
    i_[4][0] = 0;
    i_[4][1] = 1;
    i_[4][2] = 4;
    i_[5][0] = 1;
    i_[5][1] = 4;
    i_[5][2] = 5;
    i_[6][0] = 2;
    i_[6][1] = 3;
    i_[6][2] = 6;
    i_[7][0] = 3;
    i_[7][1] = 6;
    i_[7][2] = 7;
    //
    i_[8][0] = 0;
    i_[8][1] = 2;
    i_[8][2] = 6;
    i_[9][0] = 0;
    i_[9][1] = 4;
    i_[9][2] = 6;
    i_[10][0] = 1;
    i_[10][1] = 3;
    i_[10][2] = 7;
```

```
i_[11][0] = 1;
i_[11][1] = 5;
i_[11][2] = 7;
double** AB = allocate_memory_for_N_M_array<double>(8, 4);
AB[0][0] = 0;
AB[0][1] = 0;
AB[0][2] = 0;
AB[0][3] = 1;
AB[1][0] = 0;
AB[1][1] = 100;
AB[1][2] = 0;
AB[1][3] = 1;
AB[2][0] = 100;
AB[2][1] = 0;
AB[2][2] = 0;
AB[2][3] = 1;
AB[3][0] = 100;
AB[3][1] = 100;
AB[3][2] = 0;
AB[3][3] = 1;
AB[4][0] = 0;
AB[4][1] = 0;
AB[4][2] = 200;
AB[4][3] = 1;
AB[5][0] = 0;
AB[5][1] = 100;
AB[5][2] = 200;
AB[5][3] = 1;
AB[6][0] = 100;
AB[6][1] = 0;
AB[6][2] = 200;
AB[6][3] = 1;
AB[7][0] = 100;
AB[7][1] = 100;
AB[7][2] = 200;
AB[7][3] = 1;
this->figures[0].set(8, AB);
this->figures[0].associate_figure_with_polygones(this->polygones, i_, 12);
free_memory_for_N_M_array<size_t>(i_, 12, 3);
free_memory_for_N_M_array<double>(AB, 8, 4);
i_ = allocate_memory_for_N_M_array<size_t>(8, 3);
i_[0][0] = 0;
i_[0][1] = 1;
i_[0][2] = 2;
i_[1][0] = 3;
i_[1][1] = 4;
i_[1][2] = 5;
i_[2][0] = 0;
i_[2][1] = 1;
i_[2][2] = 3;
i_[3][0] = 1;
i_[3][1] = 3;
```

```
i_[3][2] = 4;
i_[4][0] = 1;
i_[4][1] = 2;
i_[4][2] = 5;
i_[5][0] = 1;
i_[5][1] = 4;
i_[5][2] = 5;
i_[6][0] = 0;
i_[6][1] = 2;
i_[6][2] = 3;
i_[7][0] = 2;
i_[7][1] = 3;
i_[7][2] = 5;
AB = allocate_memory_for_N_M_array<double>(6, 4);
AB[0][0] = 0;
AB[0][1] = 0;
AB[0][2] = 0;
AB[0][3] = 1;
AB[1][0] = 100;
AB[1][1] = 50;
AB[1][2] = 0;
AB[1][3] = 1;
AB[2][0] = 0;
AB[2][1] = 100;
AB[2][2] = 0;
AB[2][3] = 1;
AB[3][0] = 0;
AB[3][1] = 0;
AB[3][2] = 200;
AB[3][3] = 1;
AB[4][0] = 100;
AB[4][1] = 50;
AB[4][2] = 200;
AB[4][3] = 1;
AB[5][0] = 0;
AB[5][1] = 100;
AB[5][2] = 200;
AB[5][3] = 1;
this->figures[1].set(6, AB);
this->figures[1].associate_figure_with_polygones(this->polygones, i_, 8, 12);
for (i = 0; i < 12 + 8; i++)
{
   polygones_for_shadow[i] = (Polygone*)this->polygones[i];
}
free_memory_for_N_M_array<size_t>(i_, 8, 3);
free_memory_for_N_M_array<double>(AB, 6, 4);
double** l = allocate_memory_for_N_M_array<double>(1, 4);
l[0][0] = 200;
l[0][1] = 200;
l[0][2] = 5000;
l[0][3] = 1;
double*** p = new double** [1];
```

```
    p[0] = allocate_memory_for_N_M_array<double>(3, 4);
    p[0][0][0] = 0;
    p[0][0][1] = 0;
    p[0][0][2] = -1000;
    p[0][0][3] = 1;
    p[0][1][0] = 100;
    p[0][1][1] = 0;
    p[0][1][2] = -1000;
    p[0][1][3] = 1;
    p[0][2][0] = 0;
    p[0][2][1] = 100;
    p[0][2][2] = -1000;
    p[0][2][3] = 1;
    this->light_system = new Light_system(l, 1, p);
    this->light_system->associate_plane_with_polygones(this->polygones, 12 + 8);
    this->light_system->associate_light_source_with_polygones(this->polygones, 12 + 8 + 1);
    Shadow_polygone* sp_t;
    for (i = 0; i < 12 + 8; i++)
    {
    sp_t = new Shadow_polygone;
    polygones[12 + 8 + 1 + 1 + i] = sp_t;
    shadow_polygones_for_shadow[i] = sp_t;
    }
    free_memory_for_N_M_array<double>(l, 1, 4);
    free_memory_for_N_M_array<double>(p[0], 3, 4);//lol
    delete[] p;

}
int Screen::cycle()
{
    cout << "Control:\n";
    cout << "\n\tMove:\n";
    cout << "\t[w] - y--\n";
    cout << "\t[a] - x--\n";
    cout << "\t[s] - y++\n";
    cout << "\t[d] - x++\n";
    cout << "\t[q] - z--\n";
    cout << "\t[e] - z++\n";
    cout << "\n\tRotate:\n";
    cout << "\t[^] - y- rotate\n";
    cout << "\t[v] - x- rotate\n";
    cout << "\t[>] - y+ rotate\n";
    cout << "\t[<] - x+ rotate\n";
    cout << "\t[z] - z- rotate\n";
    cout << "\t[x] - z+ rotate\n";
    cout << "\n\tScale:\n";
    cout << "\t[+] - increase size\n";
    cout << "\t[-] - decrease size\n";
    cout << "\n\tChoice:\n";
    cout << "\t[l] - on/off light control\n";
    cout << "\t[p] - if light control ON plane control ON/OFF\n";
    cout << "\n\tlight control OFF\n";
```

```
cout << "\t[1] - choice figure 1\n";
cout << "\t[2] - choice figure 2\n";
cout << "\n\tlight control ON\n";
cout << "\t[1] - choice plane 1\n";
cout << "\t[2] - choice plane 2\n";

SDL_Event event;
size_t figure_choice = 0;
size_t plane_choice = 0;
int plane_flag = 1;
int exit = 1;
int light_exit = 1;
int proj_flag = 0;
while (exit == 1)
{
   while (SDL_PollEvent(&event))
   {
      if (event.type == SDL_QUIT)
      {
         exit = 0;
      }
      if ((event.type == SDL_KEYDOWN))
      {
         switch (event.key.keysym.sym)
         {
         case SDLK_l:
            //управление "светом"
            light_exit = 1;
            while(light_exit)
            {
               while (SDL_PollEvent(&event))
               {
                  if (event.type == SDL_QUIT)
                  {
                     exit = 0;
                     light_exit = 0;
                  }
                  if ((event.type == SDL_KEYDOWN))
                  {
                     switch (event.key.keysym.sym)
                     {
                     case SDLK_l:
                        light_exit = 0;
                        break;
                     case SDLK_p:
                        if (plane_flag)
                           plane_flag = 0;
                        else
                           plane_flag = 1;
                        break;
                     case SDLK_w:
                        if (plane_flag)
```

```
            this->light_system->move_up_plane(plane_choice);
        else
            this->light_system->move_up_light();
        break;
    case SDLK_s:
        if (plane_flag)
            this->light_system->move_down_plane(plane_choice);
        else
            this->light_system->move_down_light();
        break;
    case SDLK_a:
        if (plane_flag)
            this->light_system->move_left_plane(plane_choice);
        else
            this->light_system->move_left_light();
        break;
    case SDLK_d:
        if (plane_flag)
            this->light_system->move_right_plane(plane_choice);
        else
            this->light_system->move_right_light();
        break;
    case SDLK_q:
        if (plane_flag)
            this->light_system->move_forward_plane(plane_choice);
        else
            this->light_system->move_forward_light();
        break;
    case SDLK_e:
        if (plane_flag)
            this->light_system->move_back_plane(plane_choice);
        else
            this->light_system->move_back_light();
        break;
    case SDLK_UP:
        this->light_system->rotate_x_positive_plane(plane_choice);
        break;
    case SDLK_DOWN:
        this->light_system->rotate_x_negative_plane(plane_choice);
        break;
    case SDLK_RIGHT:
        this->light_system->rotate_y_positive_plane(plane_choice);
        break;
    case SDLK_LEFT:
        this->light_system->rotate_y_negative_plane(plane_choice);
        break;
    case SDLK_z:
        this->light_system->rotate_z_positive_plane(plane_choice);
        break;
    case SDLK_x:
        this->light_system->rotate_z_negative_plane(plane_choice);
        break;
```

```
            case SDLK_1:
                if (plane_flag)
                    plane_choice = 0;
                break;
            case SDLK_2:
                if (plane_flag)
                    plane_choice = 1;
                break;
            }

            SDL_SetRenderDrawColor(ren, 0x00, 0x00, 0x00, 0x00);
            SDL_RenderClear(ren);
            draw_poligones(ren);
            SDL_RenderPresent(ren);

        }
    }

}
break;

case SDLK_0:
    if(proj_flag)
        proj_flag = 0;
    else
        proj_flag = 1;
    break;
case SDLK_1:
    figure_choice = 0;
    break;
case SDLK_2:
    figure_choice = 1;
    break;
case SDLK_w:
    this->figures[figure_choice].move_up();

    break;
case SDLK_a:

    this->figures[figure_choice].move_left();
    break;
case SDLK_s:

    this->figures[figure_choice].move_down();
    break;
case SDLK_d:
    this->figures[figure_choice].move_right();

    break;
case SDLK_q:

    this->figures[figure_choice].move_forward();
```

```
                break;
            case SDLK_e:
                this->figures[figure_choice].move_back();

                break;

            case SDLK_UP:
                this->figures[figure_choice].rotate_x_positive();
                break;
            case SDLK_DOWN:
                this->figures[figure_choice].rotate_x_negative();
                break;
            case SDLK_RIGHT:
                this->figures[figure_choice].rotate_y_positive();
                break;
            case SDLK_LEFT:
                this->figures[figure_choice].rotate_y_negative();
                break;
            case SDLK_z:
                this->figures[figure_choice].rotate_z_positive();
                break;
            case SDLK_x:
                this->figures[figure_choice].rotate_z_negative();
                break;
            case SDLK_KP_PLUS:
                this->figures[figure_choice].scale_up();
                break;
            case SDLK_KP_MINUS:
                this->figures[figure_choice].scale_down();
                break;
            }
            //draw
            SDL_SetRenderDrawColor(ren, 0x00, 0x00, 0x00, 0x00);
            SDL_RenderClear(ren);

            draw_poligones(ren);
            SDL_RenderPresent(ren);
        }
    }
  }
  return 0;
}
Screen::~Screen() {
  cout << "THE END!\n";
  if (ren)
  {
    SDL_DestroyRenderer(ren);
  }
  if (win)
  {
    SDL_DestroyWindow(win);
  }
```

```
    if (polygones)
    {
        for (size_t i = 0; i < polygone_count; i++)
            delete polygones[i];
        delete[] polygones;
    }
    if (polygones_for_shadow)
        delete[] polygones_for_shadow;
    if (shadow_polygones_for_shadow)
        delete[] shadow_polygones_for_shadow;
    if (figures)
        delete[] figures;

    if (light_system)
        delete light_system;
    SDL_Quit();
}
void Screen::draw_poligones(SDL_Renderer* ren)
{

    light_system->shadows_create(12 + 8, polygones_for_shadow,
shadow_polygones_for_shadow);
    painter_algorithm();

    for (register size_t i = 0; i < polygone_count; i++)
        polygones[i]->draw(ren);
}
void Screen::painter_algorithm()
{
    register size_t i;
    for (i = 0; i < polygone_count; i++)
    {
        polygones[i]->set_z();
       // cout << polygones[i]->z<<endl;
    }
    qsort(polygones, polygone_count, sizeof(Base_polygone*), double_compare);
}
```

3.4. figure.h

```
#pragma once
#include <iostream>
#include "polygone.h"
using namespace std;
// Класс фигуры чистая математика без использования sdl и прочего...
class Figure {
public:
    Figure();
    void set(size_t n, double** coords);
    Figure(size_t n, double** coords);
    ~Figure();

    void associate_figure_with_polygones(Base_polygone** polygone_array, size_t **rule,
size_t polugones_count, size_t first_elem_pos = 0);
    void associate_figure_proj_with_polygones(Polygone* polygone_array, size_t **rule,
size_t polugones_count, size_t first_elem_pos = 0);
    void move_up();
```

```
    void move_down();
    void move_left();
    void move_right();
    void move_forward();
    void move_back();
    void rotate_y_positive();
    void rotate_y_negative();
    void rotate_x_positive();
    void rotate_x_negative();
    void rotate_z_positive();
    void rotate_z_negative();
    void scale_up();
    void scale_down();
    void create_projection();
    //#TODO прописать сеттеры и геттеры
    void set_DX(double offset = 5);
    void set_DY(double offset = 5);
    void set_DZ(double offset = 5);
private:
    double** f_proj;
    double** m_move;
    double** m_rotate_x;
    double** m_rotate_y;
    double** m_rotate_z;
    double** m_scale;
    double** m_proj;
    double** f;
    size_t N;
    size_t M; // = 4 always
    double DX;
    double DY;
    double DZ;
    double ANGLE;
    double SCALE_FACTOR;

    inline void create_m_move();
    inline void create_m_scale();
    inline void create_m_rotate_x();
    inline void create_m_rotate_y();
    inline void create_m_rotate_z();

    inline void create_m_projection();
    inline void rotate_x(double k);
    inline void rotate_y(double k);
    inline void rotate_z(double k);
    inline void scaling(double k);
    void multing(double** lin, double** matrix);
};
```

3.5. figure.cpp

```
#pragma once
#include <iostream>
#include "figure.h"
#include "polygone.h"
#include "template_functions.cpp"
using namespace std;
void Figure::create_projection()
{
   //поворот по Y и X
   register size_t i;
   register size_t j;
   for (i = 0; i < this->N; i++)
```

```
        for (j = 0; j < this->M; j++)
        {
            this->f_proj[i][j] = f[i][j];
        }
    multing(f_proj, m_proj);
}
Figure::Figure()
{
    this->f = NULL;
    this->m_move = NULL;
    this->m_scale = NULL;
    this->m_rotate_x = NULL;
    this->m_rotate_y = NULL;
    this->m_rotate_z = NULL;
    this->N = NULL;
    this->M = NULL;
    this->DX = NULL;
    this->DY = NULL;
    this->DZ = NULL;
    this->ANGLE = NULL;
    this->SCALE_FACTOR = NULL;
}
void Figure::associate_figure_with_polygones(Base_polygone** polygone_array, size_t** rule,
size_t polugones_count, size_t first_elem_pos)
{
    Polygone* p;
    for (size_t i = 0; i < polugones_count; i++)
    {
        p = new Polygone;
        p->associate(this->f[rule[i][0]], this->f[rule[i][1]], this->f[rule[i][2]]);
        polygone_array[i + first_elem_pos] = p;

    }
}
void Figure::associate_figure_proj_with_polygones(Polygone* polygone_array, size_t** rule,
size_t polugones_count, size_t first_elem_pos)
{
    cout << "ERROR a_f_p_w_p\n\n";
    for (size_t i = 0; i < polugones_count; i++)
    {
        polygone_array[i + first_elem_pos].associate(this->f_proj[rule[i][0]], this-
>f_proj[rule[i][1]], this->f_proj[rule[i][2]]);
    }
}
void Figure::set(size_t n, double** coords)
/*
*/
{
    register size_t i;
    register size_t j;
    this->DX = 5;
    this->DY = 5;
```

```
    this->DZ = 5;
    this->ANGLE = M_PI / 36;
    this->SCALE_FACTOR = 1.1;
    //allocate memory for arr, and fill this array
    this->N = n;
    this->M = 4;
    this->f = allocate_memory_for_N_M_array<double>(this->N, this->M);
    create_m_move();
    create_m_scale();
    create_m_rotate_x();
    create_m_rotate_y();
    create_m_rotate_z();
    create_m_projection();
    for (i = 0; i < this->N; i++)
        for (j = 0; j < this->M; j++)
        {
            this->f[i][j] = coords[i][j];
        }
    this->f_proj = allocate_memory_for_N_M_array<double>(this->N, this->M);
}
Figure::Figure(size_t n, double** coords)
{
    this->set(n, coords);
}
Figure::~Figure()
{
    if (this->f)
    {
        free_memory_for_N_M_array<double>(this->f, this->N, this->M);
    }
    if (this->m_move)
    {
        free_memory_for_N_M_array<double>(this->m_move, this->M, this->M);
    }
    if (this->m_scale)
    {
        free_memory_for_N_M_array<double>(this->m_scale, this->M, this->M);
    }
    if (this->m_rotate_x)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_x, this->M, this->M);
    }
    if (this->m_rotate_y)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_y, this->M, this->M);
    }
    if (this->m_rotate_z)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_z, this->M, this->M);
    }
    if (this->m_proj)
    {
```

```
        free_memory_for_N_M_array<double>(this->m_proj, this->M, this->M);
    }

}
void Figure::move_up()
{
    m_move[3][0] = 0; //
    m_move[3][1] = -this->DY; //
    m_move[3][2] = 0; //
    multing(this->f, m_move);
}
void Figure::move_down()
{
    m_move[3][0] = 0; //
    m_move[3][1] = this->DY; //
    m_move[3][2] = 0; //
    multing(this->f, m_move);
}
void Figure::move_left()
{
    m_move[3][0] = -this->DX; //
    m_move[3][1] = 0; //
    m_move[3][2] = 0; //
    multing(this->f, m_move);
}
void Figure::move_right()
{
    m_move[3][0] = this->DX; //
    m_move[3][1] = 0; //
    m_move[3][2] = 0; //
    multing(this->f, m_move);
}
void Figure::move_forward()
{
    m_move[3][0] = 0; //
    m_move[3][1] = 0; //
    m_move[3][2] = -this->DZ; //
    multing(this->f, m_move);
}
void Figure::move_back()
{
    m_move[3][0] = 0; //
    m_move[3][1] = 0; //
    m_move[3][2] = this->DZ; //
    multing(this->f, m_move);
}
void Figure::rotate_y_positive()
{
    rotate_y(1);
}
void Figure::rotate_y_negative()
{
```

```
    rotate_y(-1);
}
void Figure::rotate_x_positive()
{
    rotate_x(1);
}
void Figure::rotate_x_negative()
{
    rotate_x(-1);
}
void Figure::rotate_z_positive()
{
    rotate_z(1);
}
void Figure::rotate_z_negative()
{
    rotate_z(-1);
}
void Figure::scale_up()
{
    scaling(0);
}
void Figure::scale_down()
{
    scaling(-1);
}
inline void Figure::create_m_move()
{
    this->m_move = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_move[0][0] = 1;
    this->m_move[0][1] = 0;
    this->m_move[0][2] = 0;
    this->m_move[0][3] = 0;
    this->m_move[1][0] = 0;
    this->m_move[1][1] = 1;
    this->m_move[1][2] = 0;
    this->m_move[1][3] = 0;
    this->m_move[2][0] = 0;
    this->m_move[2][1] = 0;
    this->m_move[2][2] = 1;
    this->m_move[2][3] = 0;
    this->m_move[3][3] = 1;
}
inline void Figure::create_m_scale()
{
    this->m_scale = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_scale[0][1] = 0;
    this->m_scale[0][2] = 0;
    this->m_scale[0][3] = 0;
    this->m_scale[1][0] = 0;
    this->m_scale[1][2] = 0;
    this->m_scale[1][3] = 0;
```

```
    this->m_scale[2][0] = 0;
    this->m_scale[2][1] = 0;
    this->m_scale[2][3] = 0;
    this->m_scale[3][3] = 1;
}
inline void Figure::create_m_rotate_x()
{
    this->m_rotate_x = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_x[0][0] = 1;
    this->m_rotate_x[0][1] = 0;
    this->m_rotate_x[0][2] = 0;
    this->m_rotate_x[0][3] = 0;
    this->m_rotate_x[1][0] = 0;
    this->m_rotate_x[1][1] = cos(this->ANGLE);
    this->m_rotate_x[1][3] = 0;
    this->m_rotate_x[2][0] = 0;
    this->m_rotate_x[2][2] = cos(this->ANGLE);
    this->m_rotate_x[2][3] = 0;
    this->m_rotate_x[3][0] = 0;
    this->m_rotate_x[3][3] = 1;
}
inline void Figure::create_m_rotate_y()
{
    this->m_rotate_y = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_y[0][0] = cos(this->ANGLE);
    this->m_rotate_y[0][1] = 0;
    this->m_rotate_y[0][3] = 0;
    this->m_rotate_y[1][0] = 0;
    this->m_rotate_y[1][1] = 1;
    this->m_rotate_y[1][2] = 0;
    this->m_rotate_y[1][3] = 0;
    this->m_rotate_y[2][1] = 0;
    this->m_rotate_y[2][2] = cos(this->ANGLE);
    this->m_rotate_y[2][3] = 0;
    this->m_rotate_y[3][1] = 0;
    this->m_rotate_y[3][3] = 1;
}
inline void Figure::create_m_rotate_z()
{
    this->m_rotate_z = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_z[0][0] = cos(this->ANGLE);
    this->m_rotate_z[0][2] = 0;
    this->m_rotate_z[0][3] = 0;
    this->m_rotate_z[1][1] = cos(this->ANGLE);
    this->m_rotate_z[1][2] = 0;
    this->m_rotate_z[1][3] = 0;
    this->m_rotate_z[2][0] = 0;
    this->m_rotate_z[2][1] = 0;
    this->m_rotate_z[2][2] = 1;
    this->m_rotate_z[2][3] = 0;
    this->m_rotate_z[3][2] = 0;
    this->m_rotate_z[3][3] = 1;
```

```
}
inline void Figure::create_m_projection()
{
   this->m_proj = allocate_memory_for_N_M_array<double>(this->M, this->M);
   this->m_proj[0][0] = 0.7;
   this->m_proj[0][1] = -0.4;
   this->m_proj[0][2] = 0;
   this->m_proj[0][3] = 0;
   this->m_proj[1][0] = 0;
   this->m_proj[1][1] = 0.8;
   this->m_proj[1][2] = 0;
   this->m_proj[1][3] = 0;
   this->m_proj[2][0] = 0.7;
   this->m_proj[2][1] = 0.4;
   this->m_proj[2][2] = 1;
   this->m_proj[2][3] = 0;

   this->m_proj[3][0] = 0;
   this->m_proj[3][1] = 0;
   this->m_proj[3][2] = 0;
   this->m_proj[3][3] = 1;
}
//функция поворота вокруг оси X
inline void Figure::rotate_x(double k)
{
   register double yc = 0.0;
   register double zc = 0.0;
   register size_t i;
   for (i = 0; i < this->N; i++)
   {
      yc += this->f[i][1];
   }
   yc = yc / this->N;
   for (i = 0; i < this->N; i++)
   {
      zc += this->f[i][2];
   }
   zc = zc / this->N;
   this->m_rotate_x[1][2] = sin(k * this->ANGLE);
   this->m_rotate_x[2][1] = -sin(k * this->ANGLE);
   this->m_rotate_x[3][1] = yc * (1 - cos(this->ANGLE)) + zc * sin(k * this->ANGLE);
   this->m_rotate_x[3][2] = zc * (1 - cos(this->ANGLE)) - yc * sin(k * this->ANGLE);
   multing(this->f, this->m_rotate_x);
}
//функция поворота вокруг оси Y
inline void Figure::rotate_y(double k)
{
   register double xc = 0.0;
   register double zc = 0.0;
   register size_t i;
   for (i = 0; i < this->N; i++)
   {
```

```
        xc += this->f[i][0];
    }
    xc = xc / this->N;
    for (i = 0; i < this->N; i++)
    {
        zc += this->f[i][2];
    }
    zc = zc / this->N;
    this->m_rotate_y[0][2] = -sin(k * this->ANGLE);
    this->m_rotate_y[2][0] = sin(k * this->ANGLE);
    this->m_rotate_y[3][0] = xc * (1 - cos(this->ANGLE)) - zc * sin(k * this->ANGLE);
    this->m_rotate_y[3][2] = zc * (1 - cos(this->ANGLE)) + xc * sin(k * this->ANGLE);
    multing(this->f, this->m_rotate_y);
}
//функция поворота вокруг оси Z
inline void Figure::rotate_z(double k)
{
    register double xc = 0.0;
    register double yc = 0.0;
    register size_t i;
    for (i = 0; i < this->N; i++)
    {
        xc += this->f[i][0];
    }
    xc = xc / this->N;
    for (i = 0; i < this->N; i++)
    {
        yc += this->f[i][1];
    }
    yc = yc / this->N;
    this->m_rotate_z[0][1] = sin(k * this->ANGLE);
    this->m_rotate_z[1][0] = -sin(k * this->ANGLE);
    this->m_rotate_z[3][0] = xc * (1 - cos(this->ANGLE)) + yc * sin(k * this->ANGLE);
    this->m_rotate_z[3][1] = yc * (1 - cos(this->ANGLE)) - xc * sin(k * this->ANGLE);
    multing(this->f, this->m_rotate_z);
}
inline void Figure::scaling(double k)
{
    register double xc = 0.0;
    register double yc = 0.0;
    register double zc = 0.0;
    register size_t i;
    for (i = 0; i < this->N; i++)
    {
        xc += this->f[i][0];
    }
    xc = xc / this->N;
    for (i = 0; i < this->N; i++)
    {
        yc += this->f[i][1];
    }
    yc = yc / this->N;
```

```
    for (i = 0; i < this->N; i++)
    {
        zc += this->f[i][2];
    }
    zc = zc / this->N;
    double scale_koeff = (k == 0) ? (this->SCALE_FACTOR) : (1 / this->SCALE_FACTOR);
    this->m_scale[3][0] = xc * (1 - scale_koeff);
    this->m_scale[3][1] = yc * (1 - scale_koeff);
    this->m_scale[3][2] = zc * (1 - scale_koeff);
    this->m_scale[0][0] = scale_koeff;
    this->m_scale[1][1] = scale_koeff;
    this->m_scale[2][2] = scale_koeff;
    multing(this->f, this->m_scale); //умножение матрицы отрезка на матрицу
масштабирования
}
void Figure::multing(double** lin, double** matrix)
{
    double** res = allocate_memory_for_N_M_array<double>(this->N, this->M);
    register size_t i;
    register size_t j;
    register size_t k;
    for (i = 0; i < this->N; i++)
        for (j = 0; j < this->M; j++)
            res[i][j] = 0;
    for (i = 0; i < this->N; i++)
    {
        for (j = 0; j < this->M; j++)
        {
            for (k = 0; k < this->M; k++)
            {
                res[i][j] += (lin[i][k] * matrix[k][j]);
            }
        }
    }
    //return res
    for (i = 0; i < this->N; i++)
    {
        for (j = 0; j < this->M; j++)
        {
            lin[i][j] = res[i][j];
        }
    }
    free_memory_for_N_M_array<double>(res, this->N, this->M);
    return;
}
```

3.6. light_system.h

```
#pragma once
#include <iostream>
#include "base_polygone.h"
#include "shadow_polygone.h"
#include "polygone.h"
#include "plane.h"
#include "light_source.h"
```

```
#include "template_functions.h"
using namespace std;
//
class Light_system
{
private:
    // Математическая оснастка
    typedef struct point
    {
        double x;
        double y;
        double z;
    };
    typedef struct vector_
    {
        double x;
        double y;
        double z;
    };
    typedef struct line
    {
        point p;
        vector_ s;
    };
    typedef struct plane
    {
        double A;
        double B;
        double C;
        double D;
    };
    line point_point2line(point p1, point p2)
    {
        //прямая по 2 точкам
        return line{ p2, vector_{p1.x - p2.x, p1.y - p2.y, p1.z - p2.z} };
    }
    line paral_line_through_point(line l, point p)
    {
        //прямая параллельная данной через точку
        return line{ p, l.s };
    }
    point intersection_line_plane(line l, plane p)
    {
        if ((p.A * (l.s.x) + p.B * (l.s.y) + p.C * (l.s.z)) == 0)
        {
            cout << "ERR div by zero!\n\n";
            return point{ 0,0,0 };
        }
        double t = (-(p.A * (l.p.x) + p.B * (l.p.y) + p.C * (l.p.z) + p.D))/ (p.A *
(l.s.x) + p.B * (l.s.y) + p.C * (l.s.z));
        return point{l.p.x + l.s.x*t, l.p.y + l.s.y * t, l.p.z + l.s.z * t};
        //О***ть вот так можно было, вот это кайф реально
        double** m = allocate_memory_for_N_M_array<double>(3, 3);
        m[0][0] = l.s.y;
        m[0][1] = -l.s.x;
        m[0][2] = 0;
        m[1][0] = 0;
        m[1][1] = l.s.z;
        m[1][2] = -l.s.y;
        m[2][0] = p.A;
        m[2][1] = p.B;
        m[2][2] = p.C;
        double* v = new double[3];
        v[0] = l.s.y * l.p.x - l.s.x * l.p.y;
        v[1] = l.s.z * l.p.y - l.s.y * l.p.z;
```

```
        v[2] = -p.D;
        double* ans = fucking_Cramer_LS_solver(m, v, 3);
        point P{ ans[0], ans[1], ans[2] };
        delete[] ans;
        delete[] v;
        free_memory_for_N_M_array<double>(m, 3, 3);
        return P;
    }
    plane plane_with_3_points(point p1, point p2, point p3)
    {
        return plane{ p1.y * (p2.z - p3.z) + p2.y * (p3.z - p1.z) + p3.y * (p1.z - p2.z),
            p1.x * (p3.z - p2.z) + p2.x * (p1.z - p3.z) + p3.x * (p2.z - p1.z),
            p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y) + p3.x * (p1.y - p2.y),
            -(p1.x * (p2.y * p3.z - p3.y * p2.z) + p2.x * (p3.y * p1.z - p1.y * p3.z) +
p3.x * (p1.y * p2.z - p2.y * p1.z)) };
    }

    double determinant(double** mat, size_t n)
    {
        if (n == 3)
            return (mat[0][0]*mat[1][1]*mat[2][2] - mat[0][0] * mat[1][2] * mat[2][1] -
mat[0][1] * mat[1][0] * mat[2][2] + mat[0][1] * mat[1][2] * mat[2][0] + mat[0][2] *
mat[1][0] * mat[2][1] - mat[0][2] * mat[1][1] * mat[2][0]);

        cout << "ERROR:DETERMINANT!\n";
        return 0;
    }
    double* fucking_Cramer_LS_solver(double** mat, double* vec, size_t n)
    {
        if (n != 3)
        {
            cout << "Cramer Error\n";
            return NULL;
        }
        double* ans = new double[n];
        double det = determinant(mat, n);
        double** mat_ = allocate_memory_for_N_M_array<double>(n, n);
        mat_[0][0] = vec[0];
        mat_[0][1] = vec[1];
        mat_[0][2] = vec[2];
        mat_[1][0] = mat[1][0];
        mat_[1][1] = mat[1][1];
        mat_[1][2] = mat[1][2];
        mat_[2][0] = mat[2][0];
        mat_[2][1] = mat[2][1];
        mat_[2][2] = mat[2][2];
        double det1 = determinant(mat_, n);
        mat_[0][0] = mat[0][0];
        mat_[0][1] = mat[0][1];
        mat_[0][2] = mat[0][2];
        mat_[1][0] = vec[0];
        mat_[1][1] = vec[1];
        mat_[1][2] = vec[2];
        double det2 = determinant(mat_, n);
        mat_[1][0] = mat[1][0];
        mat_[1][1] = mat[1][1];
        mat_[1][2] = mat[1][2];
        mat_[2][0] = vec[0];
        mat_[2][1] = vec[1];
        mat_[2][2] = vec[2];
        double det3 = determinant(mat_, n);
        if (det == 0)
            det = 0.000001;
        ans[0] = det1 / det;
        ans[1] = det2 / det;
```

```
        ans[2] = det3 / det;
        free_memory_for_N_M_array<double>(mat_, n, n);
        return ans;
    }
public:
    Light_system(double **coord_light, size_t amount_planes, double*** coord_planes);
    //Light_system(size_t n, double** coords);
    ~Light_system();
    void move_up_plane(size_t choice);
    void move_down_plane(size_t choice);
    void move_left_plane(size_t choice);
    void move_right_plane(size_t choice);
    void move_forward_plane(size_t choice);
    void move_back_plane(size_t choice);
    void move_up_light();
    void move_down_light();
    void move_left_light();
    void move_right_light();
    void move_forward_light();
    void move_back_light();
    void associate_plane_with_polygones(Base_polygone** polygone_array, size_t
polugones_pos);
    void associate_light_source_with_polygones(Base_polygone** polygone_array, size_t
polugones_pos);
    void shadows_create(size_t amount_polygones, Polygone** polygones, Shadow_polygone**
shadow_polygones);

    void rotate_y_positive_plane(size_t choice);
    void rotate_y_negative_plane(size_t choice);
    void rotate_x_positive_plane(size_t choice);
    void rotate_x_negative_plane(size_t choice);
    void rotate_z_positive_plane(size_t choice);
    void rotate_z_negative_plane(size_t choice);

private:
    size_t create_shadow(size_t number_of_plane, Polygone * polygone, double** vertexes);
    double** m_move;
    double** m_rotate_x;
    double** m_rotate_y;
    double** m_rotate_z;
    double** light_point;//1*4
    double*** planes;//count_of_planes*3*4
    size_t count_of_planes;
    size_t N;// = 3 always
    size_t M; // = 4 always
    double DX;
    double DY;
    double DZ;
    double ANGLE;
    inline void create_m_move();
    inline void create_m_rotate_x();
    inline void create_m_rotate_y();
    inline void create_m_rotate_z();
    inline void rotate_x(double k, size_t number_of_plane);
    inline void rotate_y(double k, size_t number_of_plane);
    inline void rotate_z(double k, size_t number_of_plane);
    void multing(double** lin, double** matrix);
};
```

3.7. light_system.cpp

```
#pragma once
#include "light_system.h"
inline void Light_system::create_m_move()
{
```

```
    this->m_move = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_move[0][0] = 1;
    this->m_move[0][1] = 0;
    this->m_move[0][2] = 0;
    this->m_move[0][3] = 0;
    this->m_move[1][0] = 0;
    this->m_move[1][1] = 1;
    this->m_move[1][2] = 0;
    this->m_move[1][3] = 0;
    this->m_move[2][0] = 0;
    this->m_move[2][1] = 0;
    this->m_move[2][2] = 1;
    this->m_move[2][3] = 0;
    this->m_move[3][3] = 1;
}
inline void Light_system::create_m_rotate_x()
{
    this->m_rotate_x = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_x[0][0] = 1;
    this->m_rotate_x[0][1] = 0;
    this->m_rotate_x[0][2] = 0;
    this->m_rotate_x[0][3] = 0;
    this->m_rotate_x[1][0] = 0;
    this->m_rotate_x[1][1] = cos(this->ANGLE);
    this->m_rotate_x[1][3] = 0;
    this->m_rotate_x[2][0] = 0;
    this->m_rotate_x[2][2] = cos(this->ANGLE);
    this->m_rotate_x[2][3] = 0;
    this->m_rotate_x[3][0] = 0;
    this->m_rotate_x[3][3] = 1;
}
inline void Light_system::create_m_rotate_y()
{
    this->m_rotate_y = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_y[0][0] = cos(this->ANGLE);
    this->m_rotate_y[0][1] = 0;
    this->m_rotate_y[0][3] = 0;
    this->m_rotate_y[1][0] = 0;
    this->m_rotate_y[1][1] = 1;
    this->m_rotate_y[1][2] = 0;
    this->m_rotate_y[1][3] = 0;
    this->m_rotate_y[2][1] = 0;
    this->m_rotate_y[2][2] = cos(this->ANGLE);
    this->m_rotate_y[2][3] = 0;
    this->m_rotate_y[3][1] = 0;
    this->m_rotate_y[3][3] = 1;
}
inline void Light_system::create_m_rotate_z()
{
    this->m_rotate_z = allocate_memory_for_N_M_array<double>(this->M, this->M);
    this->m_rotate_z[0][0] = cos(this->ANGLE);
    this->m_rotate_z[0][2] = 0;
```

```
    this->m_rotate_z[0][3] = 0;
    this->m_rotate_z[1][1] = cos(this->ANGLE);
    this->m_rotate_z[1][2] = 0;
    this->m_rotate_z[1][3] = 0;
    this->m_rotate_z[2][0] = 0;
    this->m_rotate_z[2][1] = 0;
    this->m_rotate_z[2][2] = 1;
    this->m_rotate_z[2][3] = 0;
    this->m_rotate_z[3][2] = 0;
    this->m_rotate_z[3][3] = 1;
}
inline void Light_system::rotate_x(double k, size_t number_of_plane) {
    register double yc = 0.0;
    register double zc = 0.0;
    register size_t i;
    for (i = 0; i < this->N; i++)
    {
        yc += this->planes[number_of_plane][i][1];
    }
    yc = yc / this->N;
    for (i = 0; i < this->N; i++)
    {
        zc += this->planes[number_of_plane][i][2];
    }
    zc = zc / this->N;
    this->m_rotate_x[1][2] = sin(k * this->ANGLE);
    this->m_rotate_x[2][1] = -sin(k * this->ANGLE);
    this->m_rotate_x[3][1] = yc * (1 - cos(this->ANGLE)) + zc * sin(k * this->ANGLE);
    this->m_rotate_x[3][2] = zc * (1 - cos(this->ANGLE)) - yc * sin(k * this->ANGLE);
    multing(planes[number_of_plane], this->m_rotate_x);
}
inline void Light_system::rotate_y(double k, size_t number_of_plane) {
    register double xc = 0.0;
    register double zc = 0.0;
    register size_t i;
    for (i = 0; i < this->N; i++)
    {
        xc += this->planes[number_of_plane][i][0];
    }
    xc = xc / this->N;
    for (i = 0; i < this->N; i++)
    {
        zc += this->planes[number_of_plane][i][2];
    }
    zc = zc / this->N;
    this->m_rotate_y[0][2] = -sin(k * this->ANGLE);
    this->m_rotate_y[2][0] = sin(k * this->ANGLE);
    this->m_rotate_y[3][0] = xc * (1 - cos(this->ANGLE)) - zc * sin(k * this->ANGLE);
    this->m_rotate_y[3][2] = zc * (1 - cos(this->ANGLE)) + xc * sin(k * this->ANGLE);
    multing(this->planes[number_of_plane], this->m_rotate_y);
}
inline void Light_system::rotate_z(double k, size_t number_of_plane) {
```

```
    register double xc = 0.0;
    register double yc = 0.0;
    register size_t i;
    for (i = 0; i < this->N; i++)
    {
        xc += this->planes[number_of_plane][i][0];
    }
    xc = xc / this->N;
    for (i = 0; i < this->N; i++)
    {
        yc += this->planes[number_of_plane][i][1];
    }
    yc = yc / this->N;
    this->m_rotate_z[0][1] = sin(k * this->ANGLE);
    this->m_rotate_z[1][0] = -sin(k * this->ANGLE);
    this->m_rotate_z[3][0] = xc * (1 - cos(this->ANGLE)) + yc * sin(k * this->ANGLE);
    this->m_rotate_z[3][1] = yc * (1 - cos(this->ANGLE)) - xc * sin(k * this->ANGLE);
    multing(this->planes[number_of_plane], this->m_rotate_z);
}
void Light_system::move_up_plane(size_t choice)
{
    m_move[3][0] = 0; //
    m_move[3][1] = -this->DY; //
    m_move[3][2] = 0; //
    multing(this->planes[choice], m_move);
}
void Light_system::move_down_plane(size_t choice)
{
    m_move[3][0] = 0; //
    m_move[3][1] = this->DY; //
    m_move[3][2] = 0; //
    multing(this->planes[choice], m_move);
}
void Light_system::move_left_plane(size_t choice)
{
    m_move[3][0] = -this->DX; //
    m_move[3][1] = 0; //
    m_move[3][2] = 0; //
    multing(this->planes[choice], m_move);
}
void Light_system::move_right_plane(size_t choice)
{
    m_move[3][0] = this->DX; //
    m_move[3][1] = 0; //
    m_move[3][2] = 0; //
    multing(this->planes[choice], m_move);
}
void Light_system::move_forward_plane(size_t choice)
{
    m_move[3][0] = 0; //
    m_move[3][1] = 0; //
    m_move[3][2] = -this->DZ; //
```

```
    multing(this->planes[choice], m_move);
}
void Light_system::move_back_plane(size_t choice)
{
    m_move[3][0] = 0; //
    m_move[3][1] = 0; //
    m_move[3][2] = this->DZ; //
    multing(this->planes[choice], m_move);
}
void Light_system::move_up_light()
{
    this->light_point[0][1] -= this->DY;
}
void Light_system::move_down_light()
{
    this->light_point[0][1] += this->DY;
}
void Light_system::move_left_light()
{
    this->light_point[0][0] -= this->DX;
}
void Light_system::move_right_light()
{
    this->light_point[0][0] += this->DX;
}
void Light_system::move_forward_light()
{
    this->light_point[0][2] -= this->DZ;
}
void Light_system::move_back_light()
{
    this->light_point[0][2] += this->DZ;
}
void Light_system::shadows_create(size_t amount_polygones, Polygone** polygones,
Shadow_polygone** shadow_polygones)
{
    //Здесь нужно вызвать функцию которая "создает" теневой полигон для каждого
полигона
    size_t iter = 0;
    size_t count_of_vertex;
    double** vertexes = allocate_memory_for_N_M_array<double>(5, 3);;
    for (size_t i = 0; i < count_of_planes; i++)
    {
        for (size_t j = 0; j < amount_polygones; j++)
        {
            count_of_vertex = create_shadow(i, polygones[j], vertexes);
            //cout << vertexes[0][0] << endl;
            shadow_polygones[iter]->set_vertexes(count_of_vertex, vertexes);
            iter++;
        }
    }
    free_memory_for_N_M_array<double>(vertexes, 5, 3);
```

```
}
size_t Light_system::create_shadow(size_t number_of_plane, Polygone* polygone, double**
vertexes)
{
   point L = { light_point[0][0], light_point[0][1], light_point[0][2] };
   point P1 = { polygone->get_vertex1()[0], polygone->get_vertex1()[1], polygone-
>get_vertex1()[2] };
   point P2 = { polygone->get_vertex2()[0], polygone->get_vertex2()[1], polygone-
>get_vertex2()[2] };
   point P3 = { polygone->get_vertex3()[0], polygone->get_vertex3()[1], polygone-
>get_vertex3()[2] };
   /*

   {
1. Идеальный
   L---Ti---P или P---Ti---L
   Пересечение этой прямой с P - искомая точка Ki
2. Источник света между плоскостью и точкой полигона
   Ti---L---P или P---L---Ti
   Тень не создается для этой точки.
3. Плоскость между источником света и точкой полигона
   L---P---Ti или T---P---L
   Тень не создается для этой точки.
4. Плоскость параллельна прямой между источником света и точкой полигона
   L---Ti не пересекает P}

   */

   line line_L_P1 = point_point2line(L, P1);
   line line_L_P2 = point_point2line(L, P2);

   line line_L_P3 = point_point2line(L, P3);
   plane plane_ = plane_with_3_points(point{
planes[number_of_plane][0][0],planes[number_of_plane][0][1],planes[number_of_plane][0][2]}
, point{
planes[number_of_plane][1][0],planes[number_of_plane][1][1],planes[number_of_plane][1][2]
}, point{
planes[number_of_plane][2][0],planes[number_of_plane][2][1],planes[number_of_plane][2][2]
});

   point p1 = intersection_line_plane(line_L_P1, plane_);
   point p2 = intersection_line_plane(line_L_P2, plane_);
   point p3 = intersection_line_plane(line_L_P3, plane_);
   //cout << line_L_P1.p.x << " " << line_L_P1.s.x << endl;
   //cout << line_L_P1.p.y << " " << line_L_P1.s.y << endl;
   //cout << line_L_P1.p.z << " " << line_L_P1.s.z << endl;
#ifdef DEBUG
   cout << "PLANE: " << plane_.A << " " << plane_.B << " " << plane_.C << " " << plane_.D
<< endl;
   cout << "point pol1: " << P1.x << " " << P1.y << " " << P1.z << " " << endl;
   cout << "point pol2: " << P2.x << " " << P2.y << " " << P2.z << " " << endl;
   cout << "point pol3: " << P3.x << " " << P3.y << " " << P3.z << " " << endl;
```

```
    cout << "point light: " << L.x << " " << L.y << " " << L.z << " " << endl;

    cout << "Line1: (" << line_L_P1.p.x << "; " << line_L_P1.p.y<<"; " << line_L_P1.p.z << ")
(" << line_L_P1.s.x << "; " << line_L_P1.s.y << "; " << line_L_P1.s.z << ")" << endl;
    cout << "Line2: (" << line_L_P2.p.x << "; " << line_L_P2.p.y << "; " << line_L_P2.p.z << ")
(" << line_L_P2.s.x << "; " << line_L_P2.s.y << "; " << line_L_P2.s.z << ")" << endl;
    cout << "Line3: (" << line_L_P3.p.x << "; " << line_L_P3.p.y << "; " << line_L_P3.p.z << ")
(" << line_L_P3.s.x << "; " << line_L_P3.s.y << "; " << line_L_P3.s.z << ")" << endl;
    cout << "point intersect: " << p1.x << " " << p1.y << " " << p1.z << " " << endl;
    cout << "point intersect: " << p2.x << " " << p2.y << " " << p2.z << " " << endl;
    cout << "point intersect: " << p3.x << " " << p3.y << " " << p3.z << " " << endl<<endl;
#endif // DEBUG
    vertexes[0][0] = p1.x;
    vertexes[0][1] = p1.y;
    vertexes[0][2] = p1.z;
    vertexes[1][0] = p2.x;
    vertexes[1][1] = p2.y;
    vertexes[1][2] = p2.z;
    vertexes[2][0] = p3.x;
    vertexes[2][1] = p3.y;
    vertexes[2][2] = p3.z;
    //int type_vertex_1;
   //#TODO
    //
    //
    //
    return 3;
}
void Light_system::rotate_y_positive_plane(size_t choice)
{
    rotate_y(1, choice);
}
void Light_system::rotate_y_negative_plane(size_t choice)
{
    rotate_y(-1, choice);
}
void Light_system::rotate_x_positive_plane(size_t choice)
{
    rotate_x(1, choice);
}
void Light_system::rotate_x_negative_plane(size_t choice)
{
    rotate_x(-1, choice);
}
void Light_system::rotate_z_positive_plane(size_t choice)
{
    rotate_z(1, choice);
}
void Light_system::rotate_z_negative_plane(size_t choice)
{
    rotate_z(-1, choice);
}
```

```
void Light_system::associate_plane_with_polygones(Base_polygone** polygone_array, size_t
polugones_pos)
{
   Plane* p;

   for (size_t i = 0; i < count_of_planes; i++)
   {
      p = new Plane;
      p->associate(this->planes[i]);
      polygone_array[i + polugones_pos] = p;
   }
   /*Polygone* p;
   for (size_t i = 0; i < polugones_count; i++)
   {
      p = new Polygone;
      p->associate(this->f[rule[i][0]], this->f[rule[i][1]], this->f[rule[i][2]]);
      polygone_array[i + first_elem_pos] = p;
   }
   */
}
void Light_system::associate_light_source_with_polygones(Base_polygone** polygone_array,
size_t polugones_pos)
{
   Light_source* l;
   l = new Light_source;
   l->associate(this->light_point);
   polygone_array[polugones_pos] = l;
   /*
   Polygone* p;
   for (size_t i = 0; i < polugones_count; i++)
   {
      p = new Polygone;
      p->associate(this->f[rule[i][0]], this->f[rule[i][1]], this->f[rule[i][2]]);
      polygone_array[i + first_elem_pos] = p;
   }
   */
}
void Light_system::multing(double** lin, double** matrix)
{
   double** res = allocate_memory_for_N_M_array<double>(this->N, this->M);
   register size_t i;
   register size_t j;
   register size_t k;
   for (i = 0; i < this->N; i++)
      for (j = 0; j < this->M; j++)
         res[i][j] = 0;
   for (i = 0; i < this->N; i++)
   {
      for (j = 0; j < this->M; j++)
      {
         for (k = 0; k < this->M; k++)
         {
```

```
                res[i][j] += (lin[i][k] * matrix[k][j]);
            }
        }
    }
    //return res
    for (i = 0; i < this->N; i++)
    {
        for (j = 0; j < this->M; j++)
        {
            lin[i][j] = res[i][j];
        }
    }
    free_memory_for_N_M_array<double>(res, this->N, this->M);
    return;
}
Light_system::Light_system(double** coord_light, size_t amount_planes, double***
coord_planes)
{
    register size_t i;
    register size_t j;
    register size_t k;
    this->DX = 5;
    this->DY = 5;
    this->DZ = 5;
    this->ANGLE = M_PI / 36;
    this->N = 3;
    this->M = 4;
    this->count_of_planes = amount_planes;
    this->light_point = allocate_memory_for_N_M_array<double>(1, this->M);

    for (i = 0; i < 4; i++)
        this->light_point[0][i] = coord_light[0][i];
    create_m_move();
    create_m_rotate_x();
    create_m_rotate_y();
    create_m_rotate_z();
    this->planes = new double**[this->count_of_planes];
    for(k = 0; k< this->count_of_planes; k++)
    {
        this->planes[k] = allocate_memory_for_N_M_array<double>(this->N, this->M);
        for (i = 0; i < this->N; i++)
            for (j = 0; j < this->M; j++)
            {
                this->planes[k][i][j] = coord_planes[k][i][j];
            }
    }
}
Light_system::~Light_system()
{
    if (this->light_point)
    {
        free_memory_for_N_M_array<double>(this->light_point, 1, this->M);
```

```
    }
    for (size_t k = 0; k < this->count_of_planes; k++)
    {
        if (this->planes[k])
        {
            free_memory_for_N_M_array<double>(this->planes[k], this->N, this->M);
        }
    }
    if (this->planes)
    {
        delete[] this->planes;
    }
    if (this->m_move)
    {
        free_memory_for_N_M_array<double>(this->m_move, this->M, this->M);
    }
    if (this->m_rotate_x)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_x, this->M, this->M);
    }
    if (this->m_rotate_y)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_y, this->M, this->M);
    }
    if (this->m_rotate_z)
    {
        free_memory_for_N_M_array<double>(this->m_rotate_z, this->M, this->M);
    }
}
```

3.8. template_functions.h

```
#pragma once
#include <iostream>
typedef struct Color
{
    uint32_t r : 8;
    uint32_t g : 8;
    uint32_t b : 8;
    uint32_t a : 8;
};
template<typename T>
inline T** allocate_memory_for_N_M_array(size_t n, size_t m);
template<typename T>
inline void free_memory_for_N_M_array(T** arr, size_t n, size_t m);
```

3.9. template_functions.cpp

```
#pragma once
#include <iostream>
template<typename T>
inline T** allocate_memory_for_N_M_array(size_t n, size_t m)
{
    register size_t i;
    T** temp = new T * [n];
    for (i = 0; i < n; i++)
        temp[i] = new T[m];
    return temp;
}
```

```
template<typename T>
inline void free_memory_for_N_M_array(T** arr, size_t n, size_t m)
{
    register size_t i;
    for (i = 0; i < n; i++)
        delete[] arr[i];
    delete[] arr;
}
```

3.10. base_polygone.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "template_functions.h"
using namespace std;
class Base_polygone
    //Полигон - это всегда треугольник
{
public:
    double z;
    Base_polygone() {};
    virtual ~Base_polygone() {};
    virtual void set_z() = 0;
    virtual void set_color(Color color_) = 0;
    virtual Color get_color() = 0;
    virtual void draw(SDL_Renderer* ren) = 0;
protected:
    Color c;
};
```

3.11. polygone.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "base_polygone.h"
#include "template_functions.h"
using namespace std;
class Polygone : public Base_polygone
    //Полигон - это всегда треугольник
{
public:
    Polygone(const Polygone& polygone_copy);

    Polygone();

    Polygone& operator=(const Polygone& right);
    ~Polygone();
    virtual void set_z() override;
    void associate(double* d1, double* d2, double* d3);
    virtual void set_color(Color color_) override;
    virtual Color get_color() override;
    virtual void draw(SDL_Renderer* ren) override;
    void draw_proj(SDL_Renderer* ren);
    double* get_vertex1();
    double* get_vertex2();
    double* get_vertex3();
private:
    double *pointer_vertex_1;
    double *pointer_vertex_2;
    double *pointer_vertex_3;
};
```

3.12. polygone.cpp

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "polygone.h"
using namespace std;
Polygone::Polygone(const Polygone& polygone_copy)
{
    this->c = polygone_copy.c;

    this->z = polygone_copy.z;
    this->pointer_vertex_1 = polygone_copy.pointer_vertex_1;
    this->pointer_vertex_2 = polygone_copy.pointer_vertex_2;
    this->pointer_vertex_3 = polygone_copy.pointer_vertex_3;
}
void Polygone::set_color(Color color_)
{
    this->c = color_;
}
Color Polygone::get_color()
{
    return this->c;
}
Polygone::Polygone()
{

    this->z = NULL;

    this->c.r = rand() % 256;
    this->c.g = rand() % 256;
    this->c.b = rand() % 256;
    this->c.a = 255;
}
Polygone& Polygone::operator=(const Polygone& right) {
    //#TODO
    //проверка на самоприсваивание
    if (this == &right) {
        return *this;
    }
    this->pointer_vertex_1 = right.pointer_vertex_1;
    this->pointer_vertex_2 = right.pointer_vertex_2;
    this->pointer_vertex_3 = right.pointer_vertex_3;
    this->z = right.z;

    this->c = right.c;
    return *this;
}
Polygone::~Polygone()
{
    return;
}
void Polygone::set_z()
{
    /*
    //cout << pointer_vertex_1[0] << "; " << pointer_vertex_1[1] << "; " <<
pointer_vertex_1[2] << endl;
    //cout << pointer_vertex_2[0] << "; " << pointer_vertex_2[1] << "; " <<
pointer_vertex_2[2] << endl;
    //cout << pointer_vertex_3[0] << "; " << pointer_vertex_3[1] << "; " <<
pointer_vertex_3[2] << endl;
    */
    //z = (pointer_vertex_1[2] + pointer_vertex_2[2] + pointer_vertex_3[2]) / 3;
    //return;
    double az = pointer_vertex_1[2], bz = pointer_vertex_2[2], cz = pointer_vertex_3[2];
    double mz = (az + cz) / 2, pz = (az + bz) / 2;
```

```
    double kz = (bz + cz) / 2;
    z = (az + bz + cz + mz + pz + kz) / 6;
   // cout << z<< "   " << c.r <<":" << c.g <<":" << c.b << endl<<endl;
        return;
    /*

               A

           p
       B       Z   m

              k
                     C


    */

    if (bz != cz)
        this->z = (cz * mz - bz * pz) / (mz + cz - bz - pz);
    else
        if (az != bz)
            this->z = (bz * kz - az * mz) / (kz + bz - az - mz);
        else
            if (az != cz)
                this->z = (cz * kz - az * pz) / (kz + cz - az - pz);
            else
                this->z = az;
    //cout << az << " " << bz << " " << cz << "\n" << this->z << "\n";
    //
    //this->z = (f[0][2] + f[1][2] + f[2][2]) / 3;
}
void Polygone::associate(double* d1, double* d2, double* d3)
{
    this->pointer_vertex_1 = d1;
    this->pointer_vertex_2 = d2;
    this->pointer_vertex_3 = d3;
}
double* Polygone::get_vertex1()
{
    return this->pointer_vertex_1;
}
double* Polygone::get_vertex2()
{
    return this->pointer_vertex_2;
}
double* Polygone::get_vertex3()
{
    return this->pointer_vertex_3;
}
void  Polygone::draw_proj(SDL_Renderer* ren)
{
}
void  Polygone::draw(SDL_Renderer* ren)
{
    /*
    double a = f[0][1] * (f[1][2] - f[2][2]) + f[1][1] * (f[2][2] - f[0][2]) + f[2][1] *
(f[0][2] - f[1][2]);
    double b = f[0][0] * (f[2][2] - f[1][2]) + f[1][0] * (f[0][2] - f[2][2]) + f[2][0] *
(f[1][2] - f[0][2]);
    //double c = f[0][0] * (f[1][1] - f[2][1]) + f[1][0] * (f[2][1] - f[0][1]) + f[2][0]
* (f[0][1] - f[1][1]);
    double x1 = (f[0][0] + f[1][0] + f[2][0]) / 3;
    double y1 = (f[0][1] + f[1][1] + f[2][1]) / 3;
    double x2 = x1 + a;
    double y2 = y1 + b;
```

```
    Нормаль
   SDL_RenderDrawLine(ren, x1, y1, x2, y2);
   */
    SDL_SetRenderDrawColor(ren, c.r, c.g, c.b, c.a);
    SDL_RenderDrawLine(ren, pointer_vertex_1[0], pointer_vertex_1[1],
pointer_vertex_2[0], pointer_vertex_2[1]);
    SDL_RenderDrawLine(ren, pointer_vertex_1[0], pointer_vertex_1[1],
pointer_vertex_3[0], pointer_vertex_3[1]);
    SDL_RenderDrawLine(ren, pointer_vertex_3[0], pointer_vertex_3[1],
pointer_vertex_2[0], pointer_vertex_2[1]);
    //  return;

    double* d_;
    if (pointer_vertex_1[0] > pointer_vertex_2[0])
    {
        d_ = pointer_vertex_1;
        pointer_vertex_1 = pointer_vertex_2;
        pointer_vertex_2 = d_;
    }
    if (pointer_vertex_1[0] > pointer_vertex_3[0])
    {
        d_ = pointer_vertex_1;
        pointer_vertex_1 = pointer_vertex_3;
        pointer_vertex_3 = d_;
    }
    if (pointer_vertex_2[0] > pointer_vertex_3[0])
    {
        d_ = pointer_vertex_2;
        pointer_vertex_2 = pointer_vertex_3;
        pointer_vertex_3 = d_;
    }

    // x1= f[2][0]   f[2][1]
    // x2= f[1][0]   f[1][1]
     ////((f[1][0] - f[0][0]) == 0) ? f[0][1] : ((f[1][1] - f[0][1]) * (x - f[0][0]) /
(f[1][0] - f[0][0]) + f[0][1])) lc
     ////((f[2][0] - f[0][0]) == 0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) /
(f[2][0] - f[0][0]) + f[0][1])) lr`
         ////((f[1][0] - f[2][0]) == 0) ? f[2][1] : ((f[1][1] - f[2][1]) * (x - f[2][0]) /
(f[1][0] - f[2][0]) + f[2][1])) rc
    for (double x = pointer_vertex_1[0]; x < pointer_vertex_2[0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((pointer_vertex_2[0] - pointer_vertex_1[0]) == 0) ?
pointer_vertex_1[1] : ((pointer_vertex_2[1] - pointer_vertex_1[1]) * (x -
pointer_vertex_1[0]) / (pointer_vertex_2[0] - pointer_vertex_1[0]) +
pointer_vertex_1[1])), x, (((pointer_vertex_3[0] - pointer_vertex_1[0]) == 0) ?
pointer_vertex_1[1] : ((pointer_vertex_3[1] - pointer_vertex_1[1]) * (x -
pointer_vertex_1[0]) / (pointer_vertex_3[0] - pointer_vertex_1[0]) +
pointer_vertex_1[1])));
    }
    for (double x = pointer_vertex_2[0]; x <= pointer_vertex_3[0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((pointer_vertex_2[0] - pointer_vertex_3[0]) == 0) ?
pointer_vertex_3[1] : ((pointer_vertex_2[1] - pointer_vertex_3[1]) * (x -
pointer_vertex_3[0]) / (pointer_vertex_2[0] - pointer_vertex_3[0]) +
pointer_vertex_3[1])), x, (((pointer_vertex_3[0] - pointer_vertex_1[0]) == 0) ?
pointer_vertex_1[1] : ((pointer_vertex_3[1] - pointer_vertex_1[1]) * (x -
pointer_vertex_1[0]) / (pointer_vertex_3[0] - pointer_vertex_1[0]) +
pointer_vertex_1[1])));
    }
    /*
    for (double x = f[0][0]; x < f[1][0]; x++)
    {
```

```
        SDL_RenderDrawLine(ren, x, (((f[1][0] - f[0][0]) == 0) ? f[0][1] : ((f[1][1] -
f[0][1]) * (x - f[0][0]) / (f[1][0] - f[0][0]) + f[0][1])), x, (((f[2][0] - f[0][0]) ==
0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) / (f[2][0] - f[0][0]) + f[0][1])));
    }
    for (double x = f[1][0]; x <= f[2][0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((f[1][0] - f[2][0]) == 0) ? f[2][1] : ((f[1][1] -
f[2][1]) * (x - f[2][0]) / (f[1][0] - f[2][0]) + f[2][1])), x, (((f[2][0] - f[0][0]) ==
0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) / (f[2][0] - f[0][0]) + f[0][1])));
    }
    */


}
```

3.13. light_source.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "base_polygone.h"
#include "template_functions.h"
using namespace std;
class Light_source: public Base_polygone
    //Полигон - это всегда треугольник
{
public:
    Light_source(const Light_source& polygone_copy);
    Light_source();
    Light_source& operator=(const Light_source& right);
    ~Light_source();
    virtual void set_z() override;
    virtual void set_color(Color color_) override;
    virtual Color get_color() override;
    virtual void draw(SDL_Renderer* ren) override;
    void associate(double** l_source);
private:
    double** l;
};
```

3.14. light_source.cpp

```
#pragma once
#include "light_source.h"
using namespace std;
Light_source::Light_source(const Light_source& polygone_copy)
{
	this->c = polygone_copy.c;
	this->z = polygone_copy.z;
	this->l = polygone_copy.l;
}
Light_source::Light_source()
{
	this->z = NULL;
	this->c.r = 255;
	this->c.g = 255;
	this->c.b = 0;
	this->c.a = 255;
}
Light_source& Light_source::operator=(const Light_source& right)
{
	if (this == &right) {
		return *this;
	}
	this->c = right.c;
	this->z = right.z;
```

```
	this->l = right.l;
	return *this;
}
Light_source::~Light_source()
{

}
void Light_source::set_z()
{
	this->z = this->l[0][2];
}
void Light_source::set_color(Color color_)
{

	this->c.r = 255;
	this->c.g = 255;
	this->c.b = 255;
	this->c.a = 255;
}
Color Light_source::get_color()
{
	return this->c;
}
void Light_source::draw(SDL_Renderer* ren)
{
	SDL_SetRenderDrawColor(ren, c.r, c.g, c.b, c.a);
	SDL_RenderDrawPoint(ren, this->l[0][0], this->l[0][1]);
	//cout << this->l[0][0] << " " << this->l[0][1] << endl;
}
void Light_source::associate(double** l_source)
{
	this->l = l_source;
}
```

3.15. plane.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "base_polygone.h"
#include "template_functions.h"
using namespace std;
class Plane : public Base_polygone
    //Полигон - это всегда треугольник
{
public:

    Plane(const Plane& polygone_copy);
    Plane();
    Plane& operator=(const Plane& right);
    ~Plane();

    virtual void set_z() override;
    virtual void set_color(Color color_) override;
    virtual Color get_color() override;
    virtual void draw(SDL_Renderer* ren) override;
    void associate(double** plane_coord);

private:
    double* pointer_vertex_1;
    double* pointer_vertex_2;
    double* pointer_vertex_3;
};
```

3.16. plane.cpp

```
#pragma once
#include "plane.h"
using namespace std;
Plane::Plane(const Plane& polygone_copy)
{
    this->pointer_vertex_1 = polygone_copy.pointer_vertex_1;
    this->pointer_vertex_2 = polygone_copy.pointer_vertex_2;
    this->pointer_vertex_3 = polygone_copy.pointer_vertex_3;
    this->z = polygone_copy.z;
    this->c = polygone_copy.c;
}
    Plane::Plane()

    {
        this->z = NULL;
        this->c = {100, 100, 100, 255};
    }
    Plane& Plane::operator=(const Plane& right) {
        if (this == &right) {
            return *this;
        }
        this->pointer_vertex_1 = right.pointer_vertex_1;
        this->pointer_vertex_2 = right.pointer_vertex_2;
        this->pointer_vertex_3 = right.pointer_vertex_3;
        this->z = right.z;
        this->c = right.c;
        return *this;
    }
    Plane::~Plane()
    {
    }
    void Plane::set_z()
    {
        z =( this->pointer_vertex_1[2] + this->pointer_vertex_2[2] + this-
>pointer_vertex_3[2])/3 -1; //-1 нужен для корректного отображения теней
    }
     void Plane::set_color(Color color)
     {
         this->c.r = 255;
         this->c.g = 255;
         this->c.b = 255;
         this->c.a = 255;
     }
     Color Plane::get_color()
     {
         return c;
     }
     void Plane::draw(SDL_Renderer* ren)
     {
         //return;
         SDL_SetRenderDrawColor(ren, c.r, c.g, c.b, c.a);
         SDL_RenderClear(ren);
         //no
     }
    void Plane::associate(double** plane_coord)

    {
        this->pointer_vertex_1 = plane_coord[0];
        this->pointer_vertex_2 = plane_coord[1];
        this->pointer_vertex_3 = plane_coord[2];
    }
```

3.17. shadow_polygone.h

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "template_functions.h"
#include "base_polygone.h"
using namespace std;
//Самый плохой (слишком много допущений) класс
class Shadow_polygone : public Base_polygone
    //Теневой полигон - это многоугольник чаще треугольник но может быть 4-х и 5-ти
угольник
    //логика хранения координат в теневом полигоне выбивается из логики хранения
привычных координат. здесь мы не храним указатели на координаты а сразу храним
координаты, потомучто так логичнее всего.
{
public:
    Shadow_polygone(const Shadow_polygone& polygone_copy);
    Shadow_polygone();
    Shadow_polygone& operator=(const Shadow_polygone& right);
    ~Shadow_polygone();
    virtual void set_z() override;
    virtual void set_color(Color color_) override;
    virtual Color get_color() override;
    void set_vertexes(size_t count, double** v);
    virtual void draw(SDL_Renderer* ren) override;
private:

    size_t count_of_vertex;
    double **coords;//5*3 always - bad code
};
```

3.18. shadow_polygone.cpp

```
#pragma once
#include <iostream>
#include <SDL.h>
#include "shadow_polygone.h"
using namespace std;
void Shadow_polygone::set_vertexes(size_t count, double** v)
{
    for (register size_t i = 0; i < count; i++)
        for (register size_t j = 0; j < 3; j++)
            this->coords[i][j] = v[i][j];

    this->count_of_vertex = count;
}
Shadow_polygone::Shadow_polygone(const Shadow_polygone& polygone_copy)
{
    this->c = polygone_copy.c;
    //cout << "copy error\n";
    this->z = polygone_copy.z;

    this->coords = allocate_memory_for_N_M_array<double>(5, 3);
    for (register size_t i = 0; i < 5; i++)
        for (register size_t j = 0; j < 3; j++)
            this->coords[i][j] = polygone_copy.coords[i][j];
    this->count_of_vertex = polygone_copy.count_of_vertex;
    //cout << "copy ERROR\n";
}
Shadow_polygone::Shadow_polygone()
{
    this->z = NULL;
    this->coords = allocate_memory_for_N_M_array<double>(5, 3);

    this->count_of_vertex = 0;
```

```
    this->c.r =255;
    this->c.g = 255;
    this->c.b = 255;
    this->c.a = 255;
}
Shadow_polygone& Shadow_polygone::operator=(const Shadow_polygone& right) {
    //проверка на самоприсваивание
    if (this == &right) {
        return *this;
    }
    this->z = right.z;
    this->coords = allocate_memory_for_N_M_array<double>(5, 3);
    for (register size_t i = 0; i < 5; i++)
        for (register size_t j = 0; j < 3; j++)
            this->coords[i][j] = right.coords[i][j];
    this->count_of_vertex = right.count_of_vertex;
    this->c  = right.c;
    return *this;
}
Shadow_polygone::~Shadow_polygone()
{
    if (coords)
        free_memory_for_N_M_array<double>(coords, 5, 3);
    return;
}
void Shadow_polygone::set_z()
{

    if (!count_of_vertex)
        return;
    double az = coords[0][2], bz = coords[1][2], cz = coords[2][2];
    z = (az + bz + cz) / 3;
    return;
}
void Shadow_polygone::set_color(Color color)
{
    this->c.r = 255;
    this->c.g = 255;
    this->c.b = 255;
    this->c.a = 255;
}
Color Shadow_polygone::get_color()
{
    return c;
}
void  Shadow_polygone::draw(SDL_Renderer* ren)
{
   // return;
    /*
    double a = f[0][1] * (f[1][2] - f[2][2]) + f[1][1] * (f[2][2] - f[0][2]) + f[2][1] *
(f[0][2] - f[1][2]);
    double b = f[0][0] * (f[2][2] - f[1][2]) + f[1][0] * (f[0][2] - f[2][2]) + f[2][0] *
(f[1][2] - f[0][2]);
    //double c = f[0][0] * (f[1][1] - f[2][1]) + f[1][0] * (f[2][1] - f[0][1]) + f[2][0]
* (f[0][1] - f[1][1]);
    double x1 = (f[0][0] + f[1][0] + f[2][0]) / 3;
    double y1 = (f[0][1] + f[1][1] + f[2][1]) / 3;
    double x2 = x1 + a;
    double y2 = y1 + b;
    Нормаль
   SDL_RenderDrawLine(ren, x1, y1, x2, y2);
   */
    if (this->count_of_vertex != 3)
        return;
```

```
    SDL_SetRenderDrawColor(ren, c.r, c.g, c.b, c.a);
    SDL_RenderDrawLine(ren, coords[0][0], coords[0][1], coords[1][0], coords[1][1]);
    SDL_RenderDrawLine(ren, coords[0][0], coords[0][1], coords[2][0], coords[2][1]);
    SDL_RenderDrawLine(ren, coords[2][0], coords[2][1], coords[1][0], coords[1][1]);
    //return;
    double* d_;
    if (coords[0][0] > coords[1][0])
    {
        d_ = coords[0];
        coords[0] = coords[1];
        coords[1] = d_;
    }
    if (coords[0][0] > coords[2][0])
    {
        d_ = coords[0];
        coords[0] = coords[2];
        coords[2] = d_;
    }
    if (coords[1][0] > coords[2][0])
    {
        d_ = coords[1];
        coords[1] = coords[2];
        coords[2] = d_;
    }
    // x1= f[2][0]   f[2][1]
    // x2= f[1][0]   f[1][1]
     //(((f[1][0] - f[0][0]) == 0) ? f[0][1] : ((f[1][1] - f[0][1]) * (x - f[0][0]) /
(f[1][0] - f[0][0]) + f[0][1])) lc
     //(((f[2][0] - f[0][0]) == 0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) /
(f[2][0] - f[0][0]) + f[0][1])) lr`
         //(((f[1][0] - f[2][0]) == 0) ? f[2][1] : ((f[1][1] - f[2][1]) * (x - f[2][0]) /
(f[1][0] - f[2][0]) + f[2][1])) rc
    for (double x = coords[0][0]; x < coords[1][0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((coords[1][0] - coords[0][0]) == 0) ? coords[0][1] :
((coords[1][1] - coords[0][1]) * (x - coords[0][0]) / (coords[1][0] - coords[0][0]) +
coords[0][1])), x, (((coords[2][0] - coords[0][0]) == 0) ? coords[0][1] : ((coords[2][1]
- coords[0][1]) * (x - coords[0][0]) / (coords[2][0] - coords[0][0]) + coords[0][1])));
    }
    for (double x = coords[1][0]; x <= coords[2][0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((coords[1][0] - coords[2][0]) == 0) ? coords[2][1] :
((coords[1][1] - coords[2][1]) * (x - coords[2][0]) / (coords[1][0] - coords[2][0]) +
coords[2][1])), x, (((coords[2][0] - coords[0][0]) == 0) ? coords[0][1] : ((coords[2][1]
- coords[0][1]) * (x - coords[0][0]) / (coords[2][0] - coords[0][0]) + coords[0][1])));
    }
    /*
    for (double x = f[0][0]; x < f[1][0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((f[1][0] - f[0][0]) == 0) ? f[0][1] : ((f[1][1] -
f[0][1]) * (x - f[0][0]) / (f[1][0] - f[0][0]) + f[0][1])), x, (((f[2][0] - f[0][0]) ==
0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) / (f[2][0] - f[0][0]) + f[0][1])));
    }
    for (double x = f[1][0]; x <= f[2][0]; x++)
    {
        SDL_RenderDrawLine(ren, x, (((f[1][0] - f[2][0]) == 0) ? f[2][1] : ((f[1][1] -
f[2][1]) * (x - f[2][0]) / (f[1][0] - f[2][0]) + f[2][1])), x, (((f[2][0] - f[0][0]) ==
0) ? f[0][1] : ((f[2][1] - f[0][1]) * (x - f[0][0]) / (f[2][0] - f[0][0]) + f[0][1])));
    }
    */
}
```

4. Примеры работы

Плоскость на которую падает тень лежит перпендикулярно взгляду, на среднем расстоянии. Фигуры при инициализации лежат в друг друге. Источник света лежит ближе всего к нам.

Все изменения смотреть относительно предыдущего рисунка

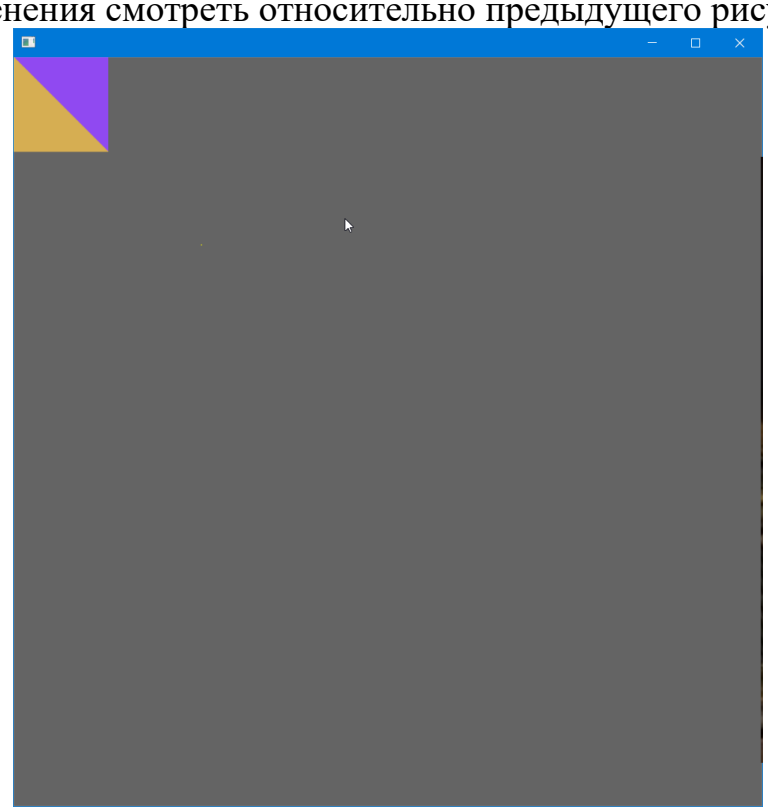

Рисунок 1 – Отрисовка двух фигур

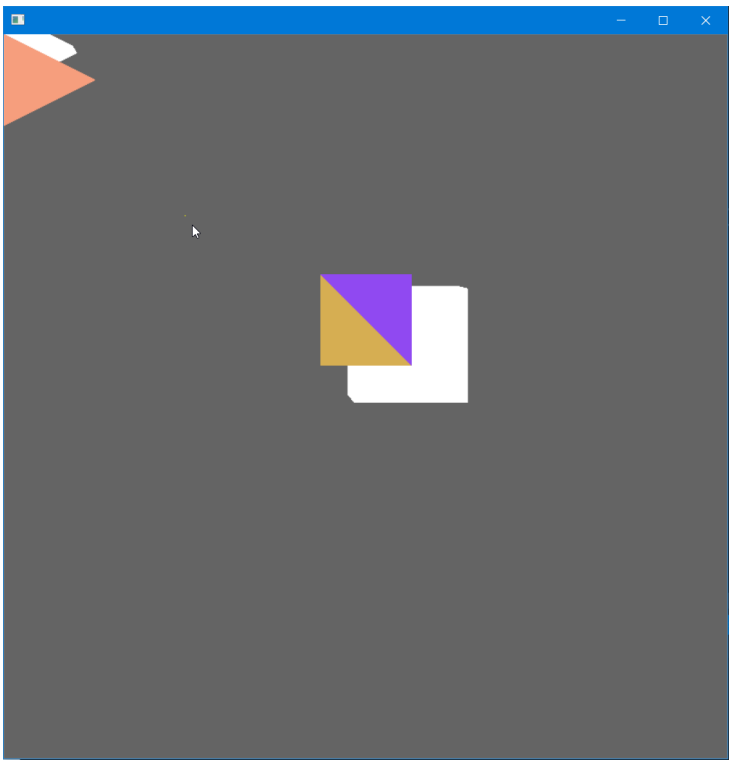

Рисунок 2 – движение одной из фигур по двум осям

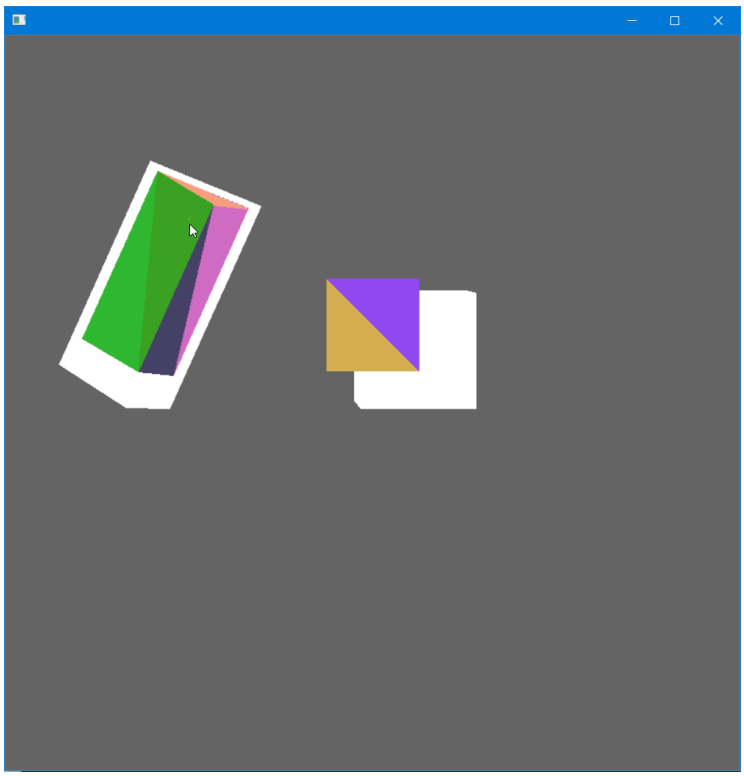

Рисунок 3 – переключение на другую фигуру, движение ее по двум осям и поворот ее по двум осям.

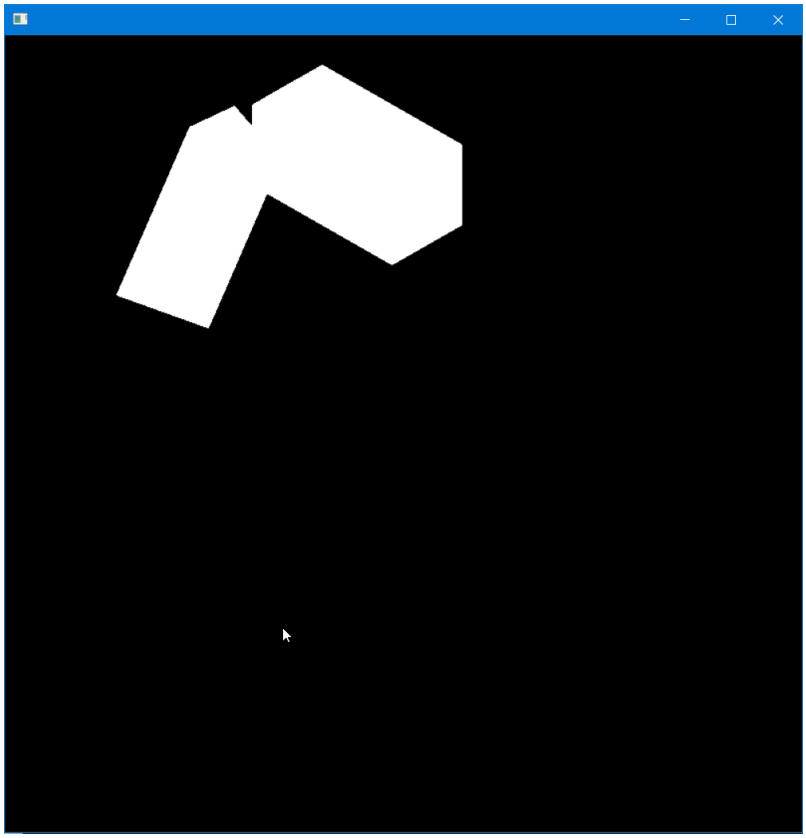

Рисунок 4 – Отображение проекции фигур

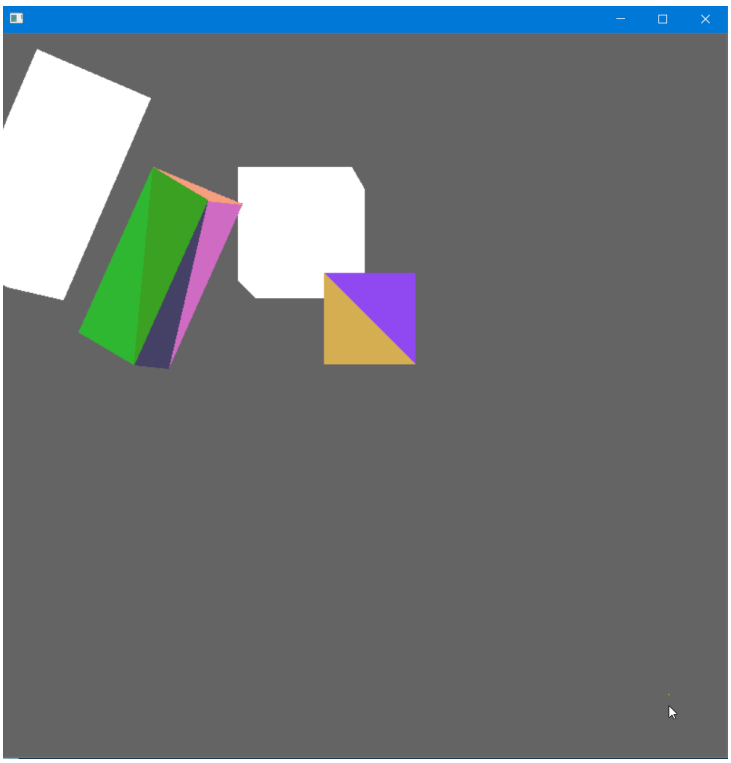

Рисунок 5 – двигаем источник света по двум осям

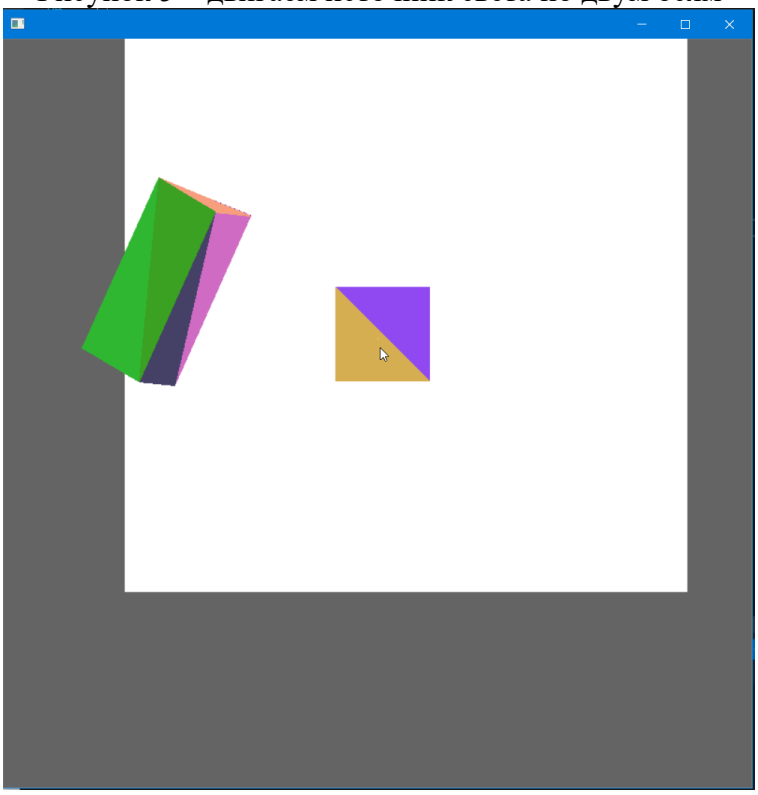

Рисунок 7 – двигаем источник света по трем осям ближе к одной из фигур

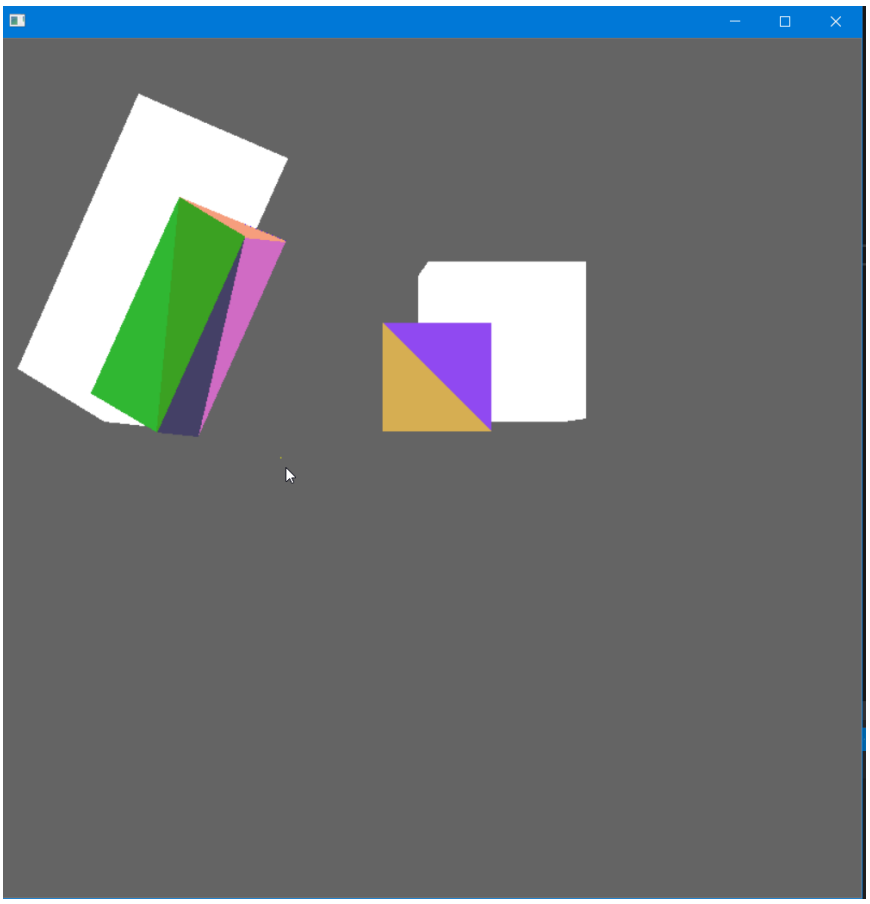

Рисунок 8 – отдаляем источник света

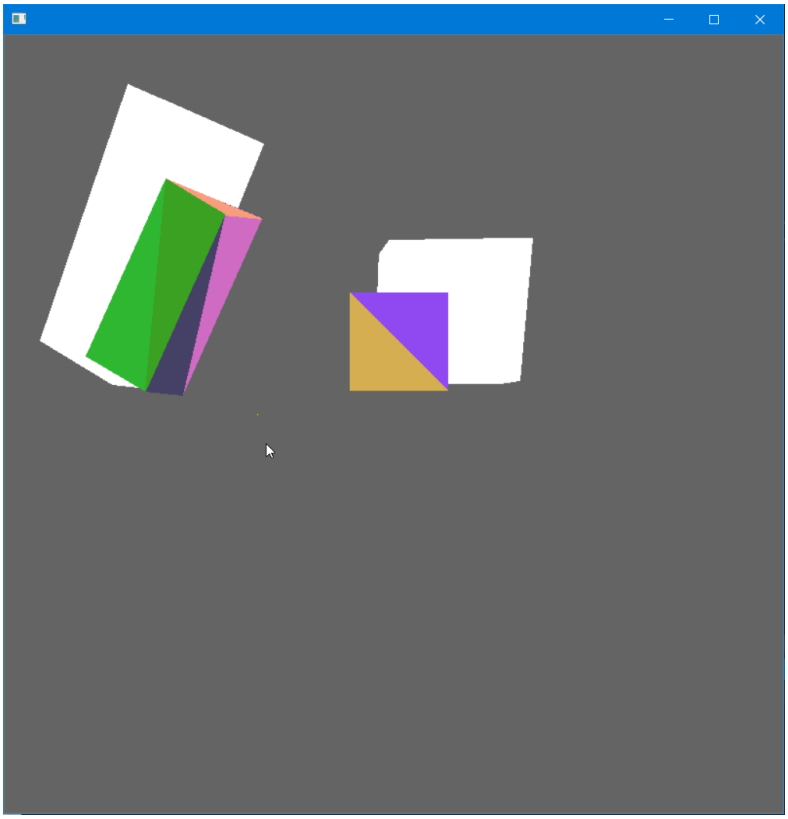

Рисунок 9 – крутим плоскость по трем осям

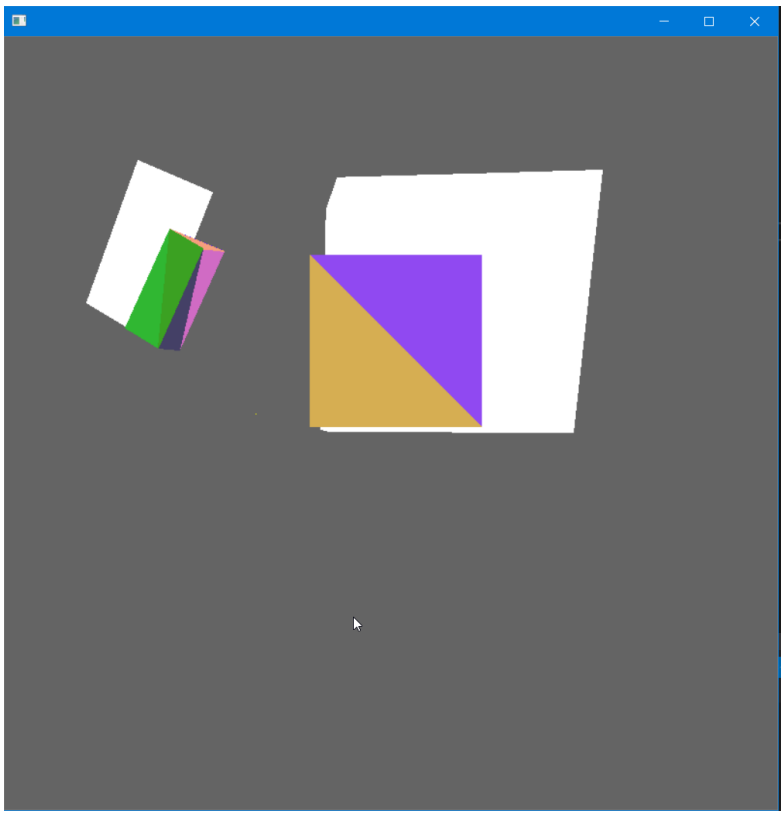

Рисунок 10 – уменьшение одной фигуры, увеличение другой

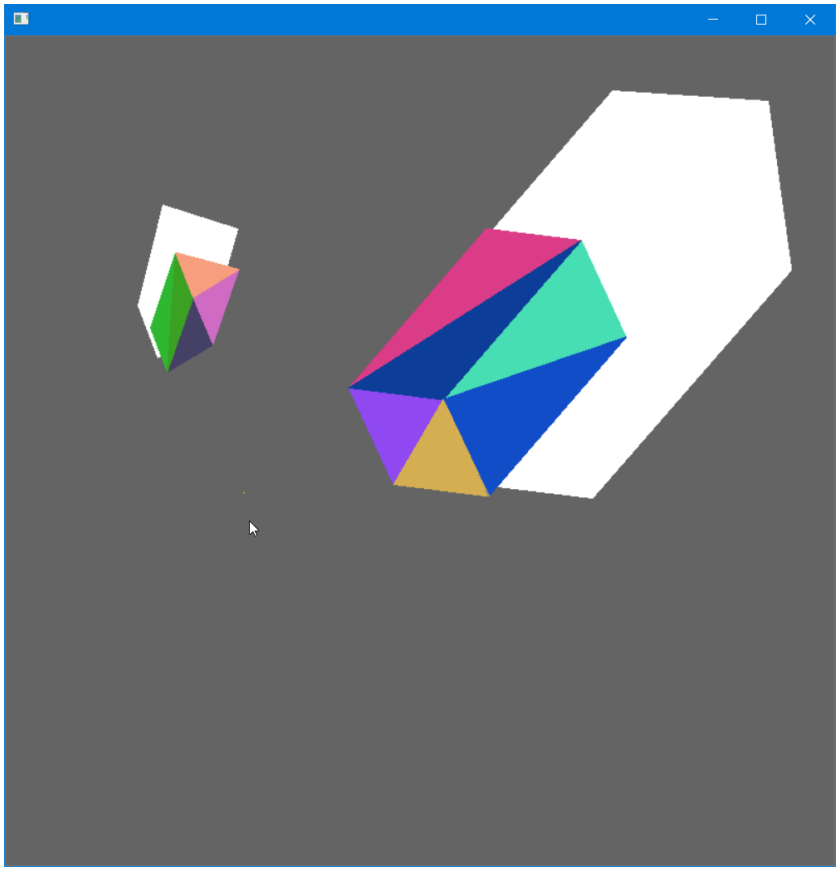

Рисунок 11 – движение и поворот всех объектов.

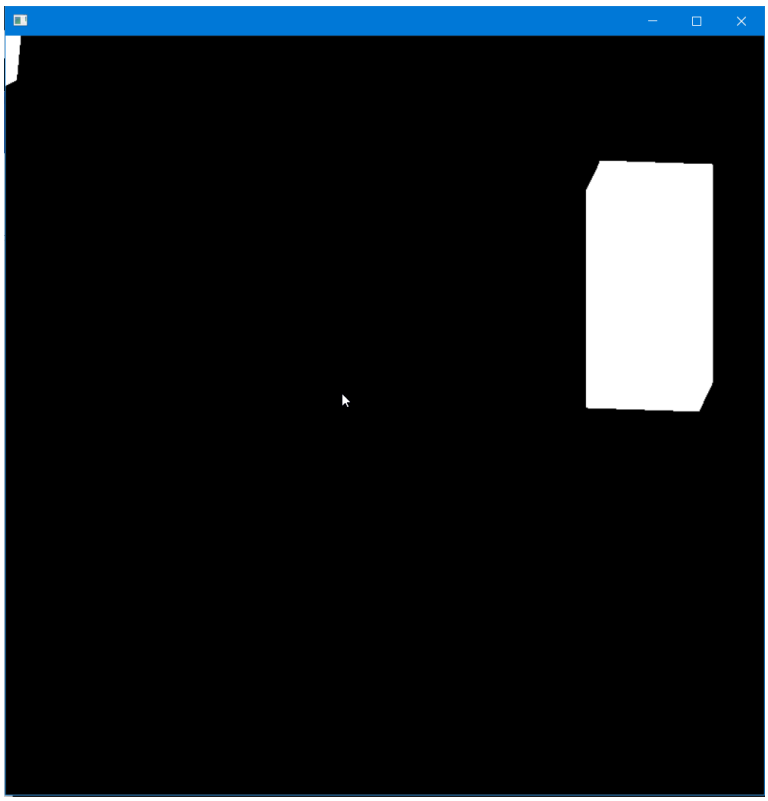

Рисунок 12 – отрисовка проекций.

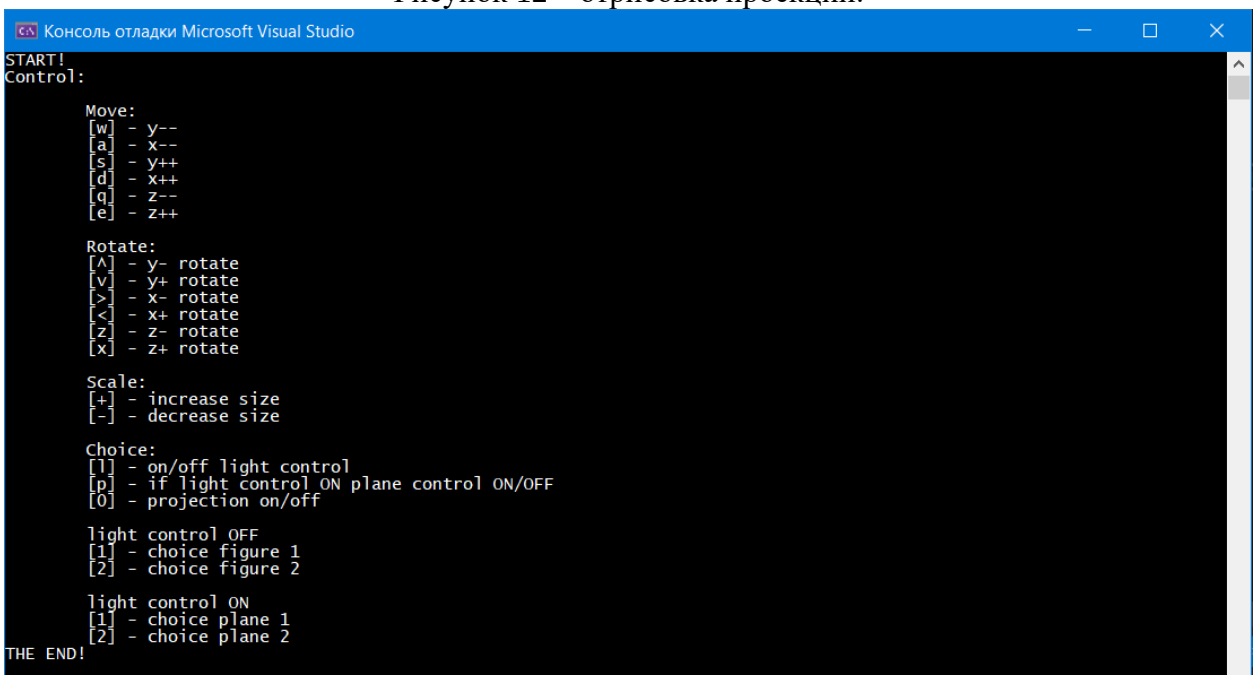

Рисунок 12 –Закрытие графического окна, весь текстовый вывод в консоли.

5. Заключение

В ходе курсовой работы были изучены принципы работы с матрицами и векторами для построения трёхмерных фигур в пространстве. Были изучены алгоритм закрашивания фигур и алгоритм «Художника» для отрисовки невидимых линий.
Были изучены методы работы с библиотекой SDL.