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** 3](#_Toc122341073)

[**2.** **Формализация** 3](#_Toc122341074)

[**3.** **Листинг программы** 5](#_Toc122341075)

[3.1. Source.cpp 5](#_Toc122341076)

[3.2. screen.h 7](#_Toc122341077)

[3.3. screen.cpp 8](#_Toc122341078)

[3.4. figure.h 17](#_Toc122341079)

[3.5. figure.cpp 18](#_Toc122341080)

[3.6. light\_system.h 26](#_Toc122341081)

[3.7. light\_system.cpp 29](#_Toc122341082)

[3.8. template\_functions.h 38](#_Toc122341083)

[3.9. template\_functions.cpp 38](#_Toc122341084)

[3.10. base\_polygone.h 39](#_Toc122341085)

[3.11. polygone.h 39](#_Toc122341086)

[3.12. polygone.cpp 40](#_Toc122341087)

[3.13. light\_source.h 43](#_Toc122341088)

[3.14. light\_source.cpp 43](#_Toc122341089)

[3.15. plane.h 44](#_Toc122341090)

[3.16. plane.cpp 45](#_Toc122341091)

[3.17. shadow\_polygone.h 46](#_Toc122341092)

[3.18. shadow\_polygone.cpp 46](#_Toc122341093)

[**4.** **Примеры работы** 49](#_Toc122341094)

[**5.** **Заключение** 53](#_Toc122341095)

## **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)

# **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*](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.

# **Листинг программы**

# 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 плоскоcти 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;

}

# 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();

};

# 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);

}

# 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);

};

# 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;

}

# 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);

};

# 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);

}

}

# 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);

# 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;

}

# 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\_сolor(Color сolor\_) = 0;

virtual Color get\_сolor() = 0;

virtual void draw(SDL\_Renderer\* ren) = 0;

protected:

Color c;

};

# 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\_сolor(Color сolor\_) override;

virtual Color get\_сolor() 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;

};

# 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\_сolor(Color сolor\_)

{

this->c = сolor\_;

}

Color Polygone::get\_сolor()

{

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])));

}

\*/

}

# 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\_сolor(Color сolor\_) override;

virtual Color get\_сolor() override;

virtual void draw(SDL\_Renderer\* ren) override;

void associate(double\*\* l\_source);

private:

double\*\* l;

};

# 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\_сolor(Color сolor\_)

{

this->c.r = 255;

this->c.g = 255;

this->c.b = 255;

this->c.a = 255;

}

Color Light\_source::get\_сolor()

{

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;

}

# 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\_сolor(Color сolor\_) override;

virtual Color get\_сolor() 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;

};

# 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 отбражения теней

}

void Plane::set\_сolor(Color сolor)

{

this->c.r = 255;

this->c.g = 255;

this->c.b = 255;

this->c.a = 255;

}

Color Plane::get\_сolor()

{

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];

}

# 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\_сolor(Color сolor\_) override;

virtual Color get\_сolor() 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

};

# 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\_сolor(Color сolor)

{

this->c.r = 255;

this->c.g = 255;

this->c.b = 255;

this->c.a = 255;

}

Color Shadow\_polygone::get\_сolor()

{

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])));

}

\*/

}

# **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

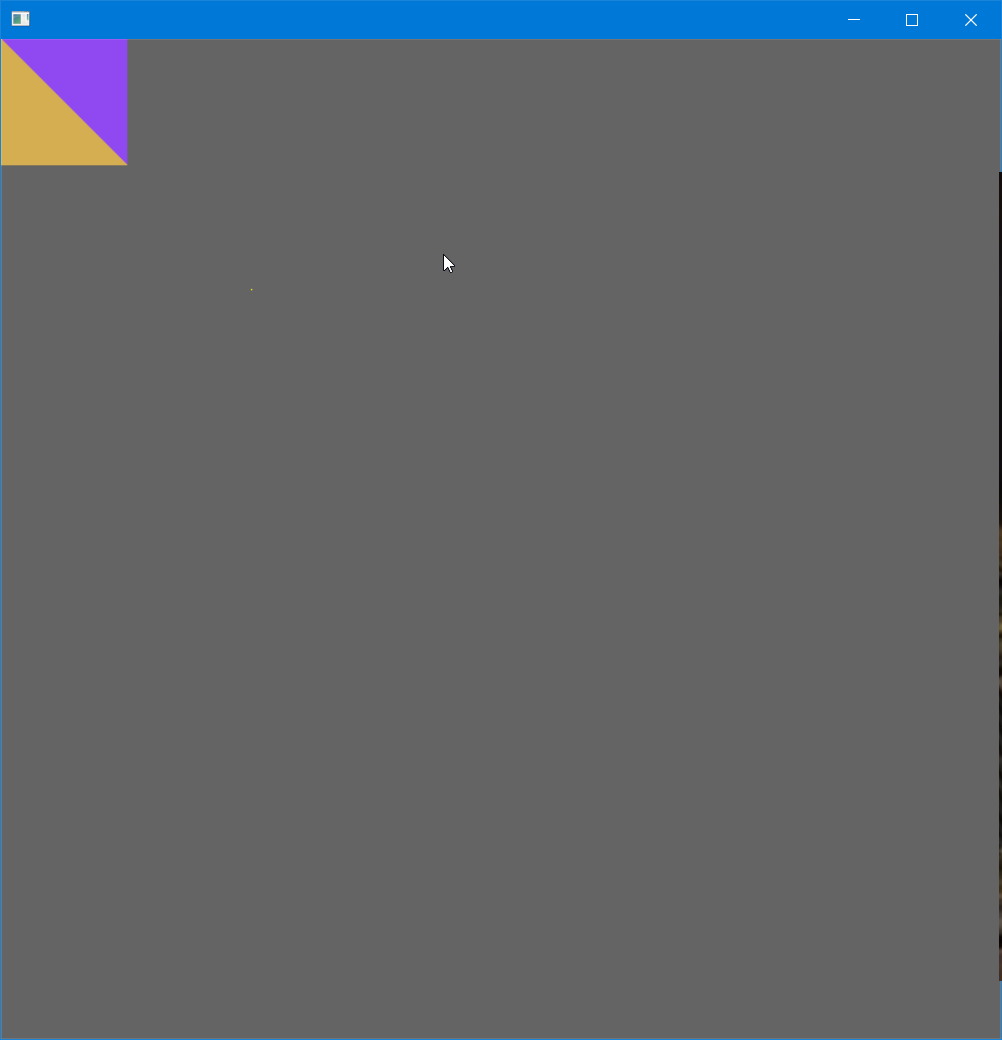
****

Рисунок 1 – Отрисовка двух фигур

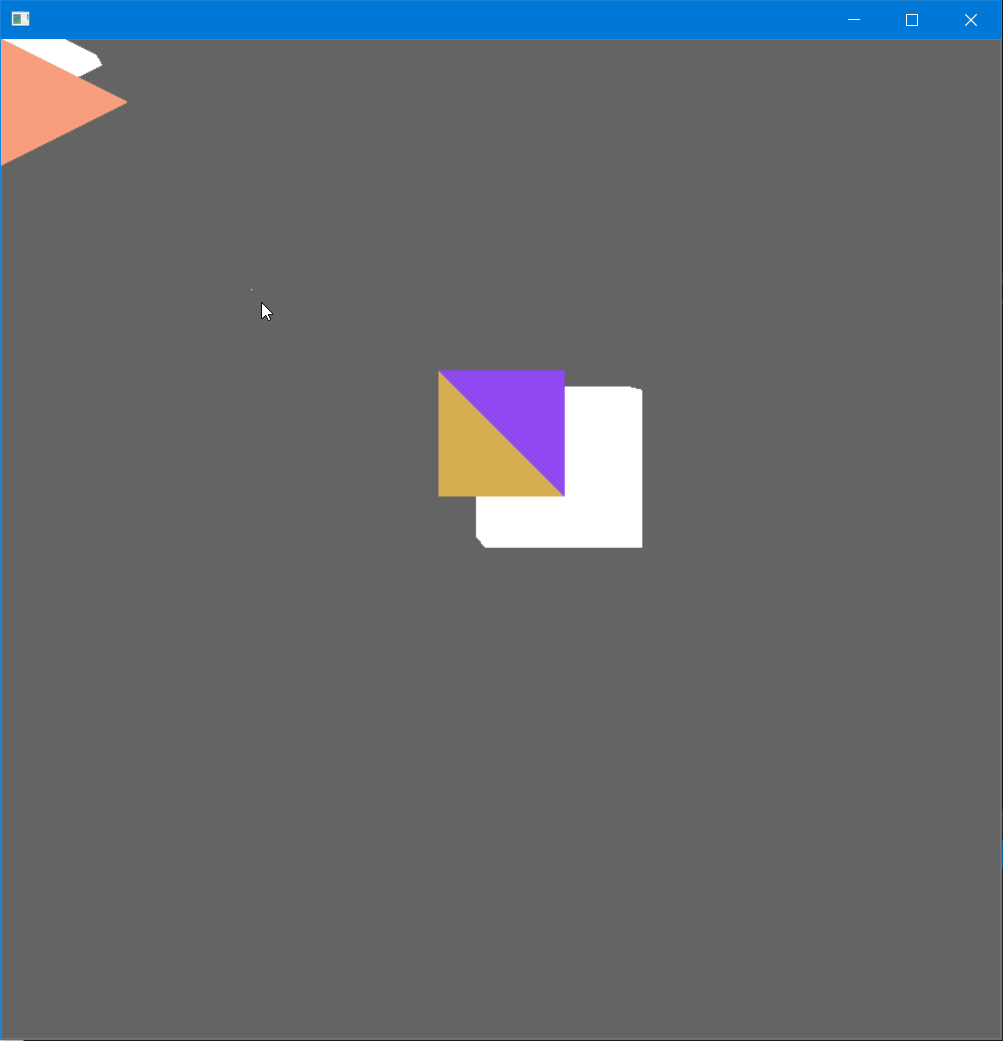


Рисунок 2 – движение одной из фигур по двум осям

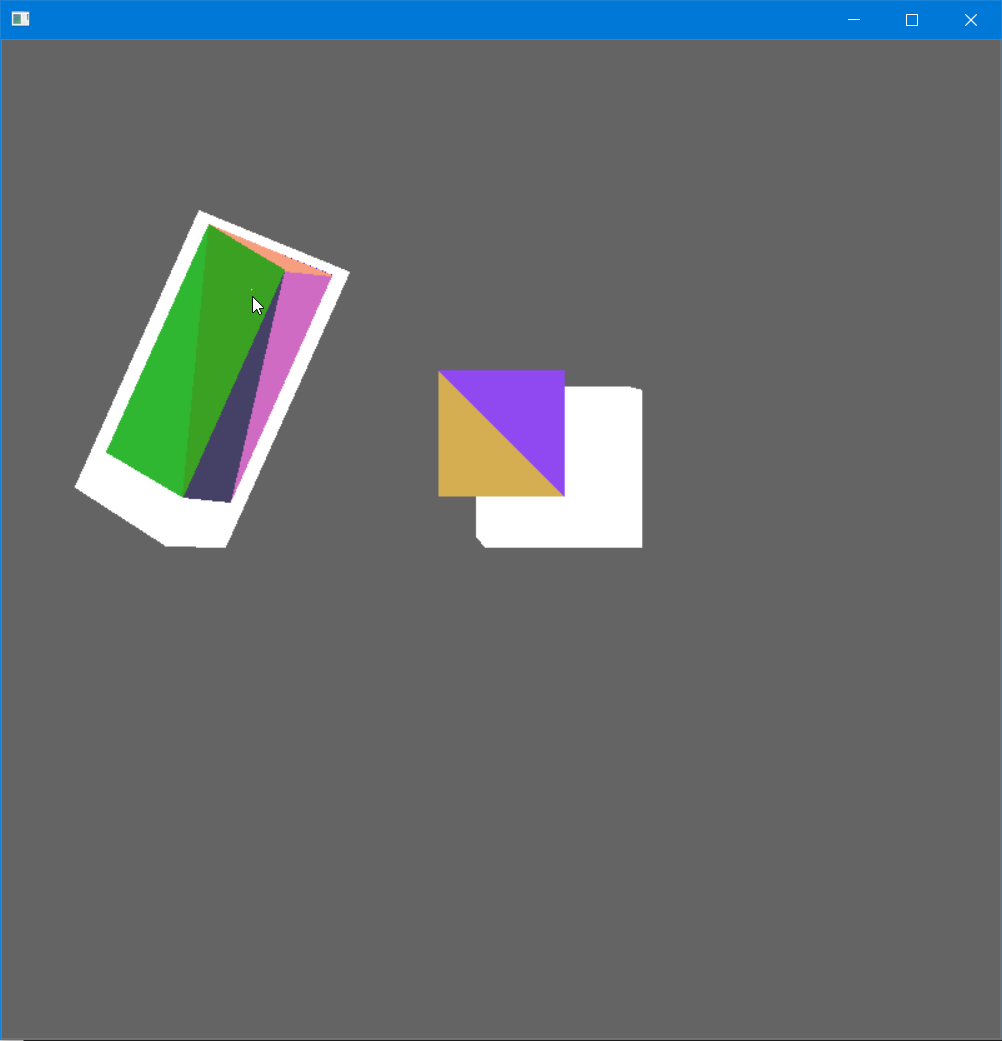


Рисунок 3 – переключение на другую фигуру, движение ее по двум осям и поворот ее по двум осям.

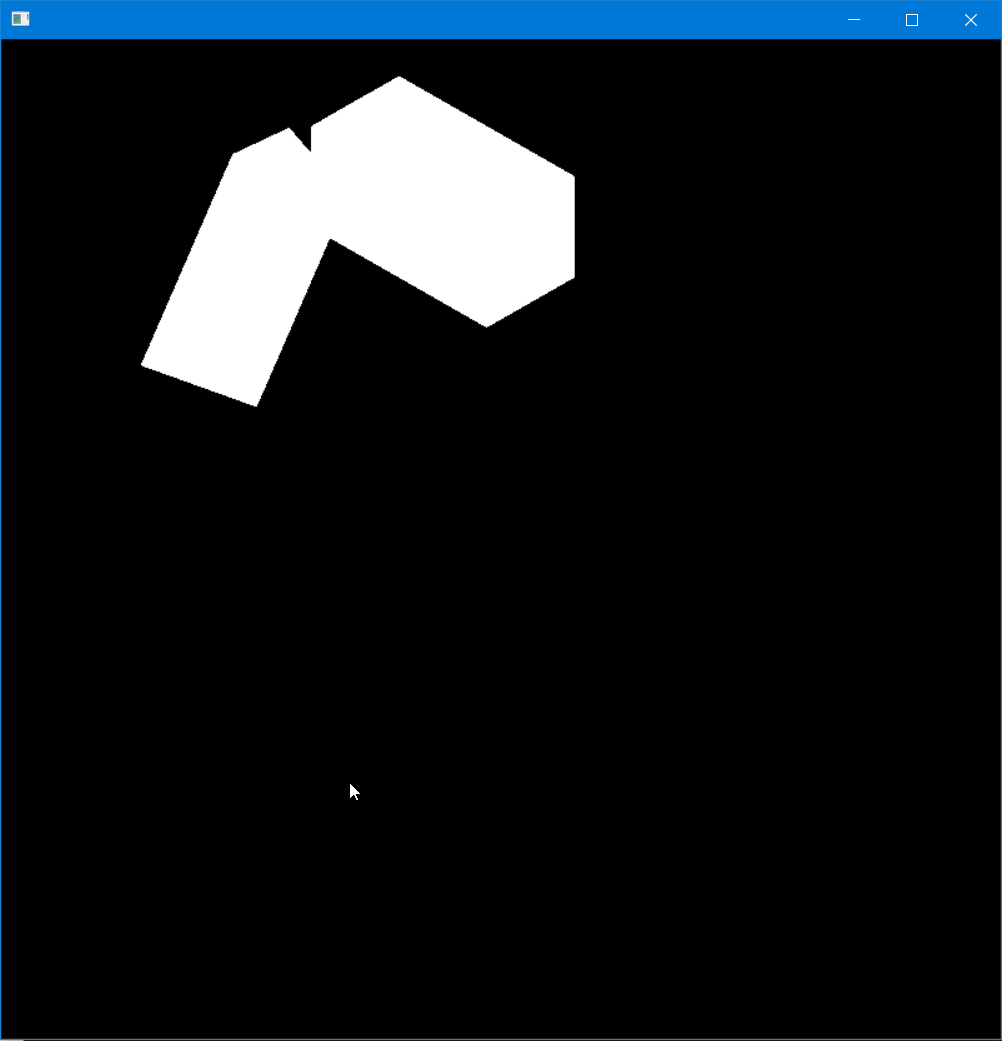


Рисунок 4 – Отображение проекции фигур

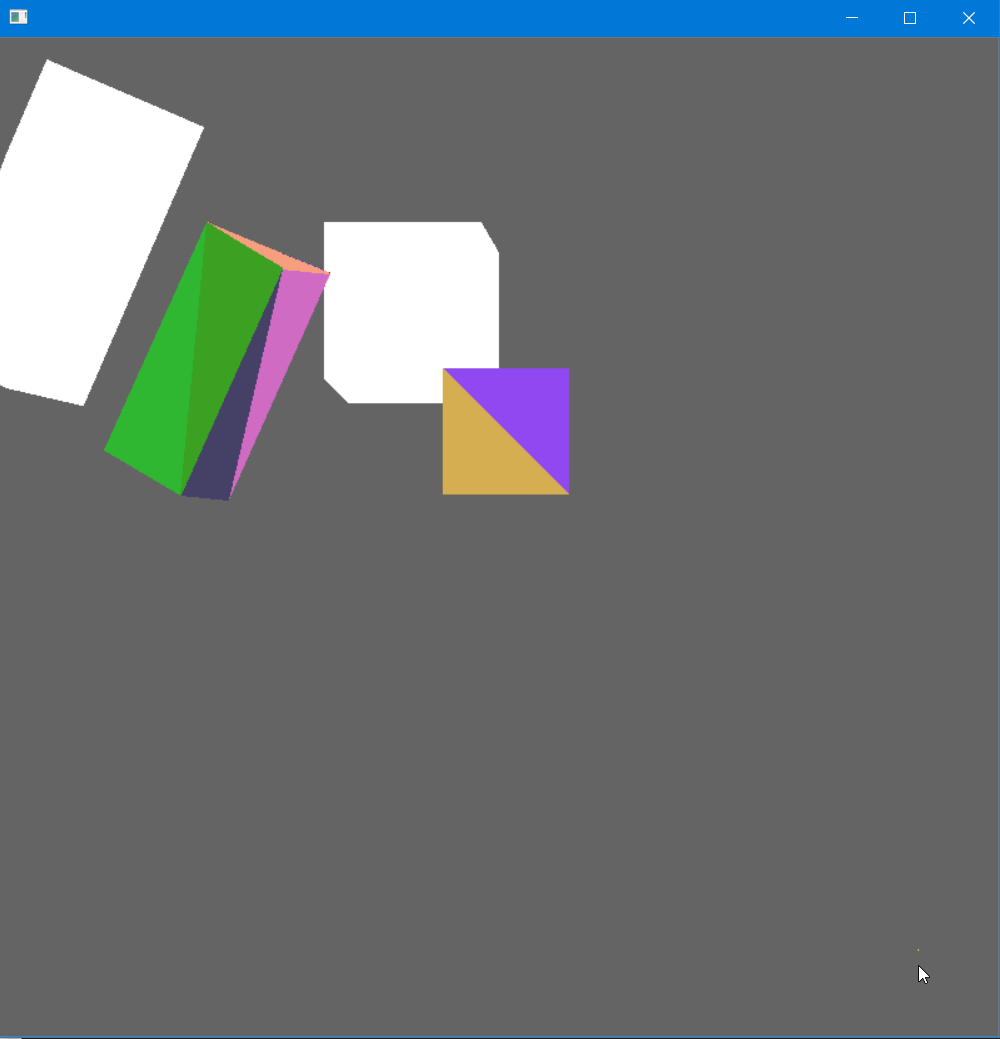


Рисунок 5 – двигаем источник света по двум осям

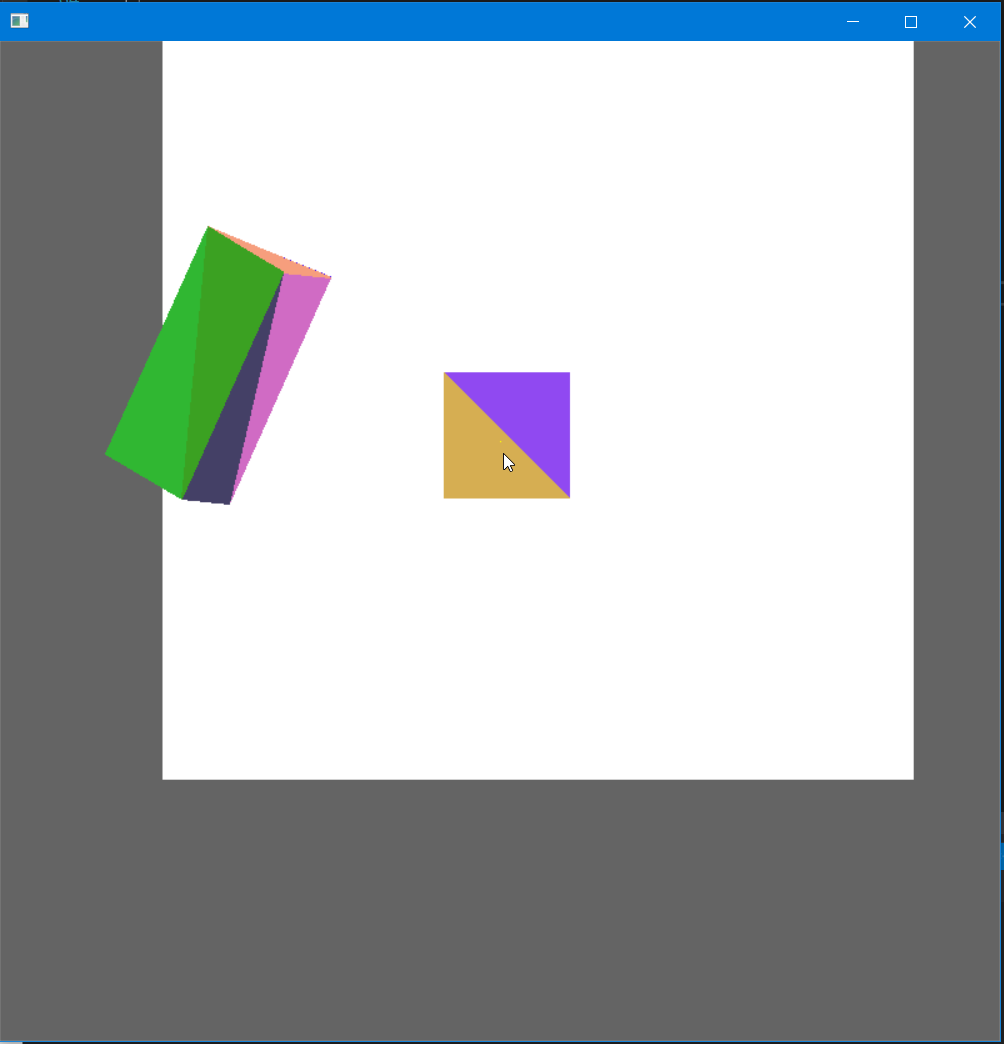


Рисунок 7 – двигаем источник света по трем осям ближе к одной из фигур

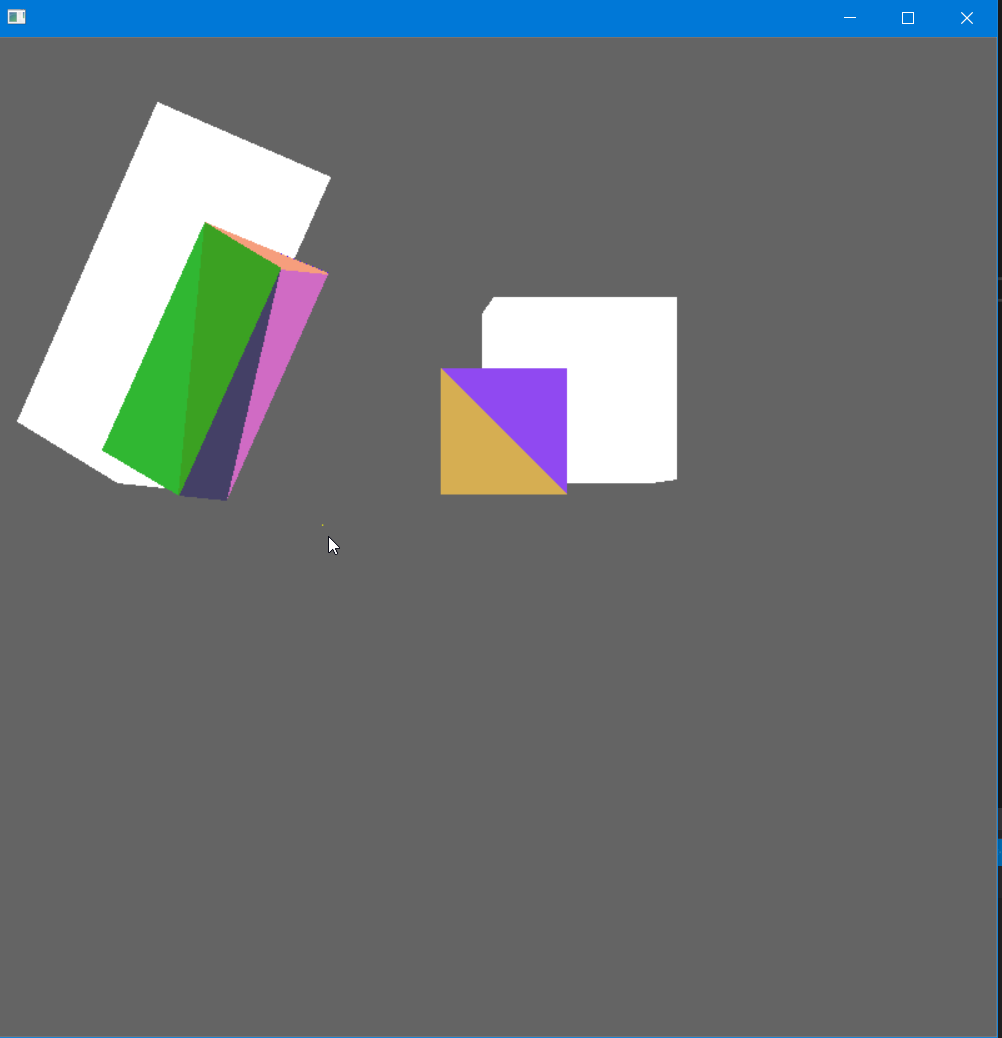


Рисунок 8 – отдаляем источник света

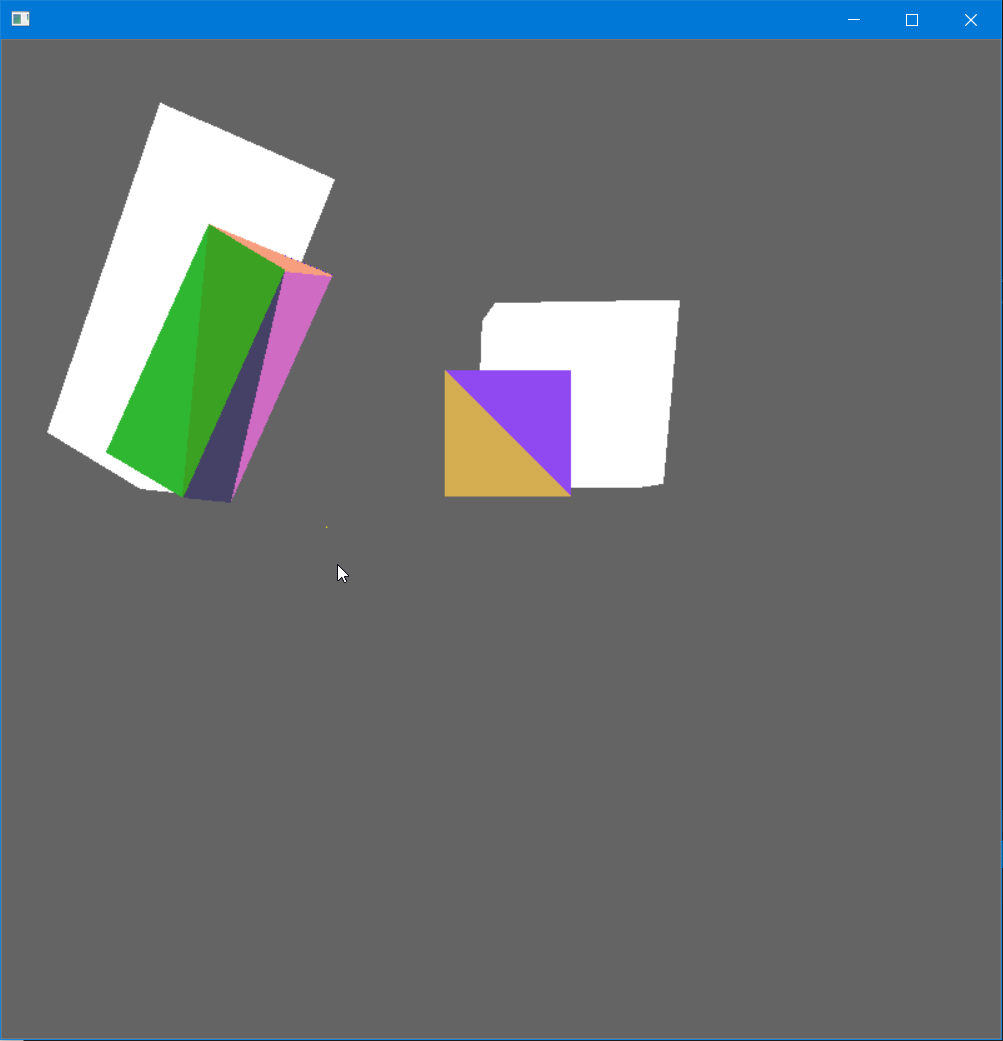


Рисунок 9 – крутим плоскость по трем осям

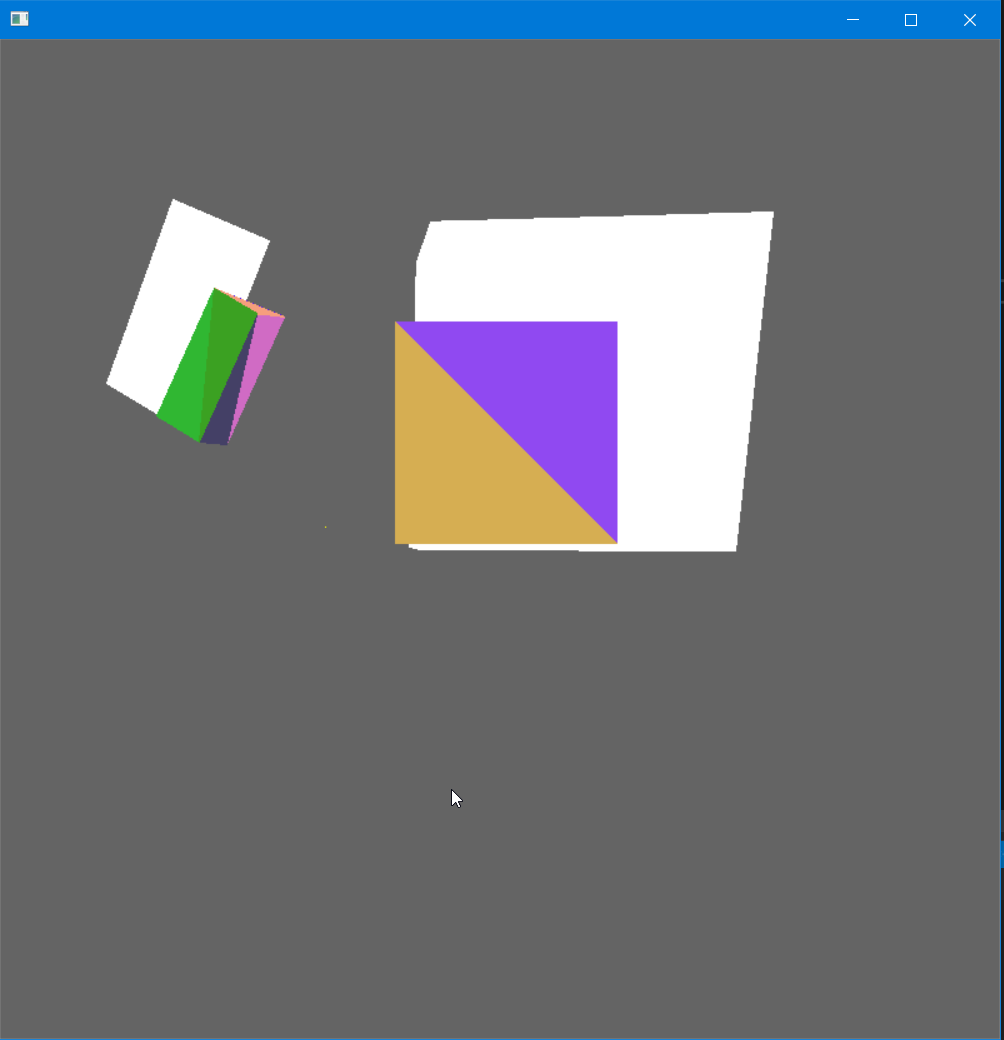


Рисунок 10 – уменьшение одной фигуры, увеличение другой

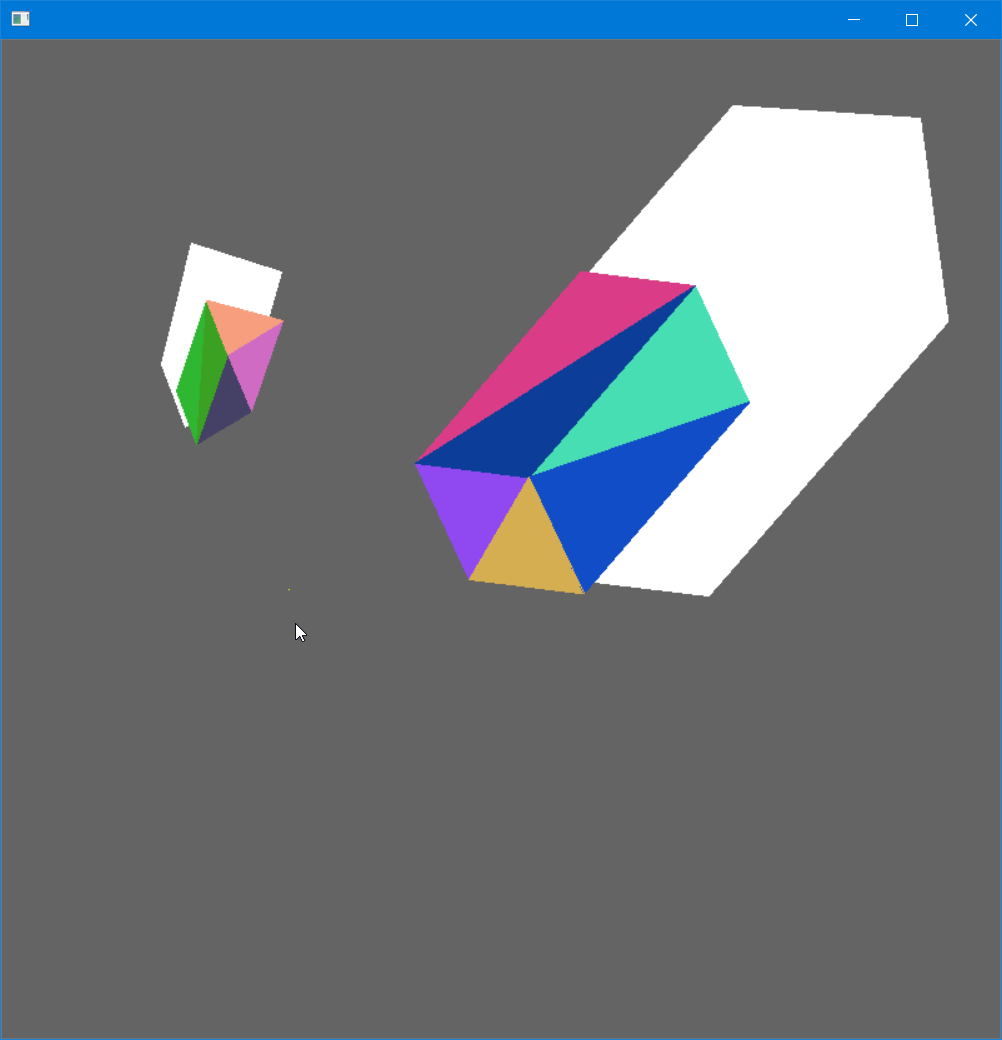


Рисунок 11 – движение и поворот всех объектов.

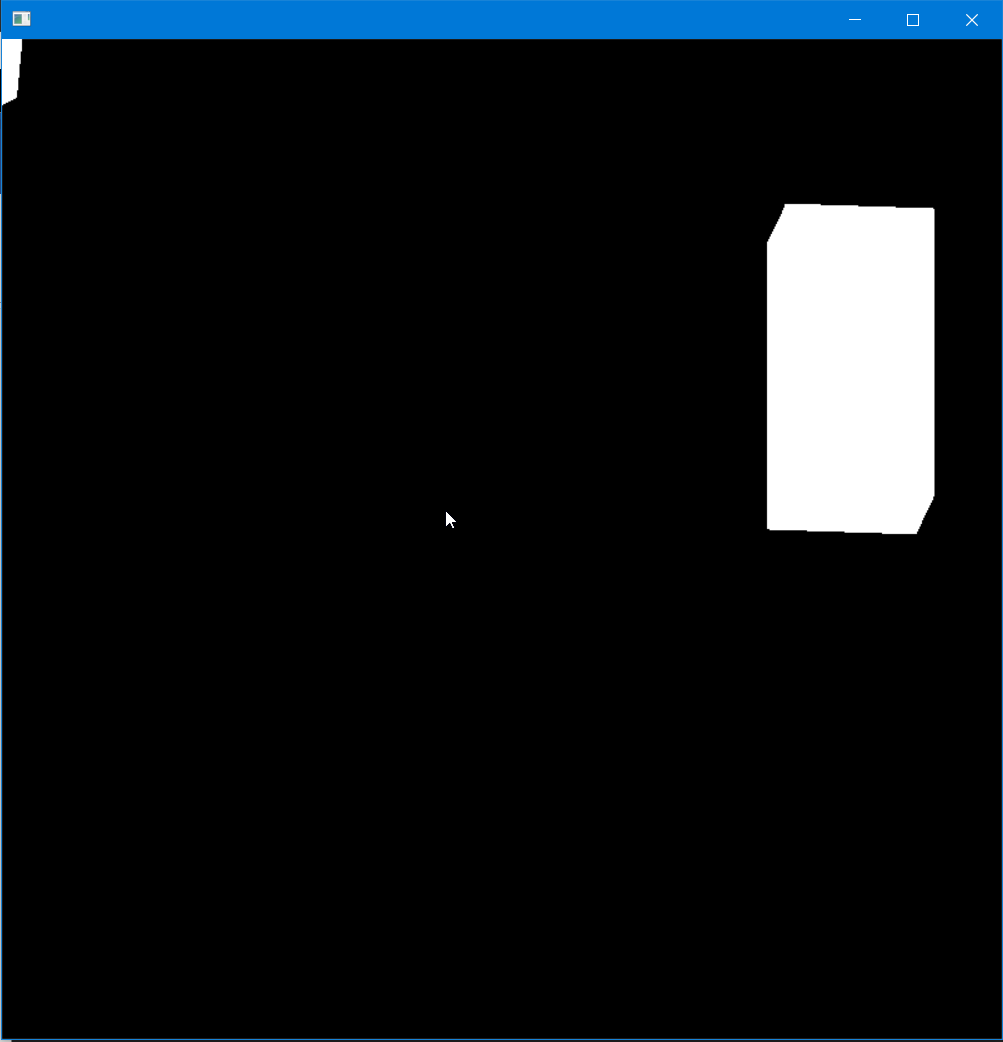


Рисунок 12 – отрисовка проекций.

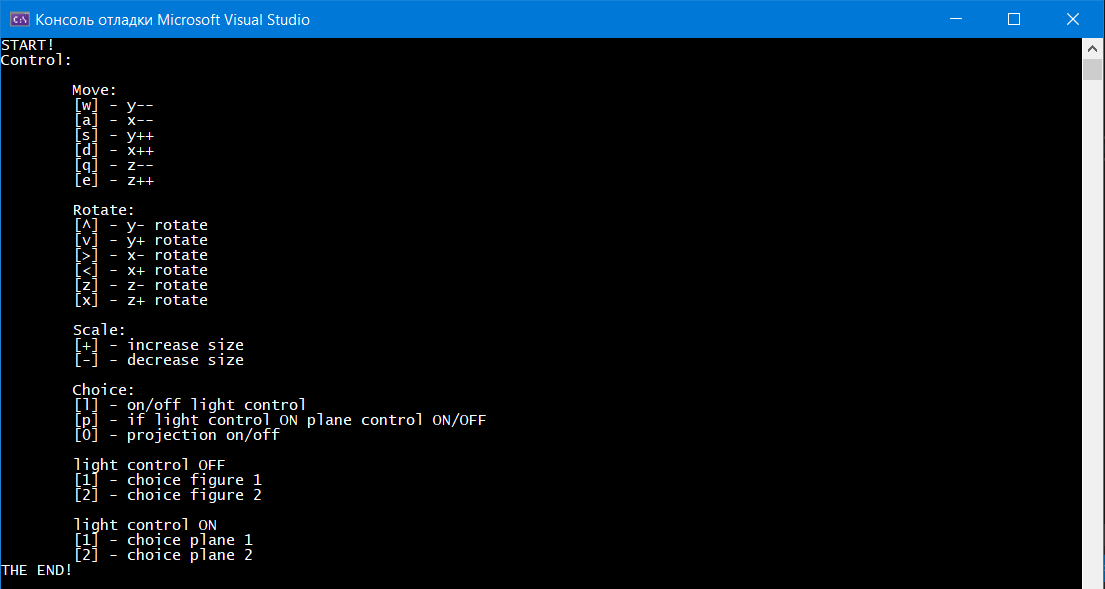


Рисунок 12 –Закрытие графического окна, весь текстовый вывод в консоли.

# **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