**Импортирование созданных моделей и ландшафта в программную среду**

Экспортировать файл модели будем в формате obj. Данный формат содержит информацию о вершинах модели и гранях.

Формат obj файла:

#вершины

v 0.123 0.234 0.345

v -0.123 0.234 0.345

v 0.123 -0.234 0.345

#грани (индексы вершин)

f 1 2 3

f 3 4 5

f 6 3 7

f 7 8 9

Алгоритм чтения из файла:

Токен - это строка с присвоенным и, таким образом, идентифицированным значением. В нашем случае токенами будут “v”, “t”, “#”, числовые значения а также некоторые другие лексемы obj формата, неиспользуемые нашей программой, но которые могут быть сгенерированы программой 3D моделирования. Их мы будем игнорировать.

Объявление токена:

enum TokenKind {  
 *TOKEN\_FLOAT* = 128,*//0-128 reserved for symbols  
 TOKEN\_INT*,  
 *TOKEN\_NAME*,  
 *TOKEN\_COMMENT*,  
 *TOKEN\_VERTEX*,  
 *TOKEN\_FACE*,  
 *TOKEN\_OBJECT\_NAME*,  
 *// ...*};

struct Token {  
 TokenKind kind;  
 const char \*start;  
 const char \*end;  
 union {  
 double float\_val;  
 uint32\_t int\_val;  
 const char\* name;  
 };  
};

Функция *next\_token()* обрабатывает информацию о следующем токене. Так же в ней, считываются числовые значения координат и индексов вершин.

Загрузка файла в память:

const char\* stream\_start;  
const char\* stream;

void init\_stream(const char\* str) {  
 stream\_start = stream = str;  
 char\_to\_digit['0'] = 0;  
 char\_to\_digit['1'] = 1;  
 char\_to\_digit['2'] = 2;  
 char\_to\_digit['3'] = 3;  
 char\_to\_digit['4'] = 4;  
 char\_to\_digit['5'] = 5;  
 char\_to\_digit['6'] = 6;  
 char\_to\_digit['7'] = 7;  
 char\_to\_digit['8'] = 8;  
 char\_to\_digit['9'] = 9;  
  
 next\_token();  
}  
  
void init\_stream\_from\_file(FILE\* from){  
 size\_t grow\_size = 1024;  
 char\* buffer = (char\*)malloc(grow\_size);  
 int len = 0,grow\_len = 0;  
  
 int end = 0;  
 for(;end!=**EOF**;){  
 end = fscanf(from,"%c",buffer+len);  
 len++;  
 grow\_len++;  
 if (grow\_len == grow\_size){  
 buffer = (char\*)realloc(buffer,len+grow\_size);  
 grow\_len = 1;  
 }  
 }  
 buffer[len-1] = '\0';  
 init\_stream(buffer);  
}

Функция *parse\_vertex*, будет вызываться каждый раз, когда находится токен вершины. Функция последовательно считывает координаты вершин и записывает из в вектор.

POINT3 vertex;  
Edge edge;  
std::vector<uint32\_t> indexes;  
  
bool parse\_vertex(){  
 next\_token();  
 expect\_token(TokenKind(' '));  
 if(is\_token(*TOKEN\_FLOAT*)){  
 vertex.x = token.float\_val;  
 }else if(is\_token(*TOKEN\_INT*)){  
 vertex.x = token.int\_val;  
 }else{  
 fatal("unexpected token");  
 }  
  
 next\_token();  
 expect\_token(TokenKind(' '));  
 if(is\_token(*TOKEN\_FLOAT*)){  
 vertex.y = token.float\_val;  
 }else if(is\_token(*TOKEN\_INT*)){  
 vertex.y = token.int\_val;  
 }else{  
 fatal("unexpected token");  
 }  
  
 next\_token();  
 expect\_token(TokenKind(' '));  
 if(is\_token(*TOKEN\_FLOAT*)){  
 vertex.z = token.float\_val;  
 }else if(is\_token(*TOKEN\_INT*)){  
 vertex.z = token.int\_val;  
 }else{  
 fatal("unexpected token");  
 }  
  
 next\_token();  
 expect\_token(TokenKind('\n'));  
 if (is\_token(*TOKEN\_VERTEX*)){  
 return false;  
 }else{  
 return true;  
 }  
}

Функция *parse\_face*, будет вызываться каждый раз, когда находится токен грани. Функция последовательно считывает индексы вершин и записывает новую грань в вектор.

static bool parse\_face(){  
 next\_token();  
 int count = 0;  
 while(token.kind != '\n' && \*(stream+1) !='\0'){  
 expect\_token(TokenKind(' '));  
 if (is\_token(*TOKEN\_INT*)){  
 indexes.push\_back(token.int\_val-1);  
 count++;  
 }else{  
 fatal("unexpected token");  
 }  
 next\_token();  
 if(is\_token(TokenKind('/'))){  
 while(token.kind != TokenKind(' ') && token.kind != TokenKind('\n') && \*(stream+1) !='\0'){  
 next\_token();  
 }  
 }  
 }  
 face.vertices\_count = count;  
  
 next\_token();  
 if (is\_token(*TOKEN\_FACE*)){  
 return false;  
 }else{  
 return true;  
 }  
}

В функции *parse\_object* реализован цикл обработки файла. Функция возвращает объект типа *Object*. Выделение памяти под данные модели, происходит из приватной кучи для объектов. Это структурирует данные, и увеличивает локальность кэша. (см. Приложение arena.h, arena.cpp).

Object\* parse\_obj(arena\_t\* arena,const char\* fname);

std::vector<POINT3> vertexes;  
std::vector<Face> faces;  
  
bool last = false;  
while(!is\_token(TokenKind('\0'))) {  
 switch (token.kind) {  
 case *TOKEN\_VERTEX*:  
 while(!last){  
 last = parse\_vertex();  
 vertexes.push\_back(vertex);  
 }  
 last = false;  
 break;  
 case *TOKEN\_FACE*:  
 while(!last){  
 last = parse\_face();  
 faces.push\_back(face);  
 }  
 last = false;  
 break;  
 default:  
 next\_token();  
 break;  
 }  
}

**Разработка программы визуализации объекта**

Для визуализации объекта были объявлены следующие классы:

struct POINT3{  
 double x;  
 double y;  
 double z;  
};

struct Face{  
 uint32\_t\* order;  
 uint32\_t vertices\_count;  
};  
  
class Object{  
private:  
 POINT3\* world\_points;  
 POINT\* screen\_points;  
 Face\* faces;  
 uint32\_t edges\_count;  
 uint32\_t vertices\_count;  
public:  
 Object(){  
 world\_points = nullptr;  
 screen\_points = nullptr;  
 faces = nullptr;  
 edges\_count = 0;  
 vertices\_count = 0;  
 }  
 Object(POINT3\* world\_points, POINT\* screen\_points, Face\* edges, uint32\_t vertices\_count, uint32\_t edges\_count){  
 this->world\_points = world\_points;  
 this->screen\_points = screen\_points;  
 this->faces = edges;  
 this->edges\_count = edges\_count;  
 this->vertices\_count = vertices\_count;  
 }  
 void update(Camera& cam, int w\_height, int w\_width);  
 void draw(HDC hdc,COLORREF color);  
};

Для динамики камеры были объявлены следующие классы:

#define **ANGLE\_UPPER\_LIMIT**(deg) ((deg) < 360 ? (deg) : (deg)-360)  
#define **ANGLE\_LOWER\_LIMIT**(deg) ((deg) > 0 ? (deg) : (deg)+360)  
  
struct Radius{  
 double radius;  
  
 inline double operator =(double r){  
 radius = (r > 0 ? r : 0);  
 return radius;  
 }  
  
 inline void operator +=(double r){  
 radius += r;  
 }  
 inline void operator -=(double r){  
 radius = (radius-r > 0 ? radius-r : 0);  
 }  
 inline void operator \*=(double r){  
 radius \*= r;  
 }  
 inline void operator /=(double r){  
 radius /= r;  
 }  
 inline bool operator ==(double r){  
 return radius == r;  
 }  
 inline bool operator !=(double r){  
 return radius != r;  
 }  
};  
  
struct Angle{  
 double angle;  
  
 void operator +=(double deg){  
 angle = **ANGLE\_UPPER\_LIMIT**(angle+deg);  
 }  
 void operator -=(double deg){  
 angle = **ANGLE\_LOWER\_LIMIT**(angle-deg);  
 }  
 void operator \*=(double deg){  
 angle = **ANGLE\_UPPER\_LIMIT**(angle\*deg);  
 }  
 void operator /=(double deg){  
 angle /= deg;  
 }  
 inline bool operator ==(double deg){  
 return angle == deg;  
 }  
 inline bool operator !=(double deg){  
 return angle != deg;  
 }  
};  
  
struct SphericalCoords{  
 Radius r;  
 Angle phi;  
 Angle theta;  
  
 POINT3 to\_cartesian();  
};  
  
struct Camera{  
 SphericalCoords center\_position;   
 POINT offsets;  
};

Структуры *Angle* и *Radius*, были созданы для предотвращения переполнения значений сферических координат.

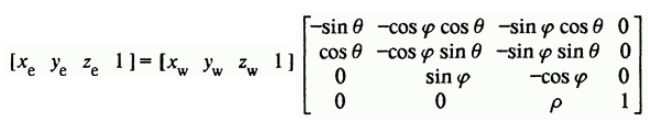
Основной класс Object содержит информацию необходимую для рендеринга модели. Метод *update* реализует обновление экранных координат объекта *screen\_points*, путем преобразования мировых координат в экранные, в зависимости от расположения камеры и экранной плоскости.

void Object::update(Camera& cam, int w\_height, int w\_width) {  
 for (int i = 0; i < vertices\_count; ++i) {  
 screen\_points[i]=RenderPoint(world\_points[i],cam,w\_height,w\_width);  
 }  
}

Функция *RenderPoint* последовательно производит следующие преобразования над каждой мировой координатой модели:

Видовые преобразования:

Значений координат выведены из формулы видовых преобразований:



double A = DegreeToRad(cam.center\_position.phi.value);  
double B = DegreeToRad(cam.center\_position.theta.value);  
  
double sin\_A = sin(A);  
double sin\_B = sin(B);  
double cos\_A = cos(A);  
double cos\_B = cos(B);  
  
double X = world\_point.x, Y= world\_point.y, Z= world\_point.z;  
  
double M3[3] = {Y\*cos\_A - X\*sin\_A,  
 Z\*sin\_B - X\*cos\_A\*cos\_B - Y\*sin\_A\*cos\_B,  
 -X\*cos\_A\*cos\_B - Y\*sin\_A\*sin\_B - Z\*cos\_B + cam.center\_position.r.value};  
  
POINT3 screen\_dot = PointEx(M3[0], M3[1], M3[2]);

Перспективные преобразования:  
X = screen\_dot.x,Y = screen\_dot.y, Z = screen\_dot.z;  
  
POINT projected = Point(Round(X/Z\*cam.dc),  
 Round(Y/Z\*cam.dc));

Экранные:

return Point(projected.x + w\_width/2 + cam.offsets.x,-projected.y + w\_height/1.5 + cam.offsets.y);

Метод *draw* рисует ребра модели по координатам *screen\_points* и их индексам в *faces*.

void Object::draw(HDC hdc,COLORREF color) {  
 SelectObject(hdc,CreatePen(**PS\_SOLID**,0,color));  
  
 POINT p1,p2;  
 for (int i = 0; i < edges\_count; ++i) {  
 p1 = screen\_points[faces[i].order[0]];  
 MoveToEx(hdc,p1.x,p1.y, nullptr);  
 for (int j = 1; j < faces[i].vertices\_count; ++j) {  
 p2 = screen\_points[faces[i].order[j]];  
 LineTo(hdc,p2.x,p2.y);  
 }  
 LineTo(hdc,p1.x,p1.y);  
 }  
}

Функции рисования земли и осей абсцисс, ординат и аппликат были реализованы отдельно, т.к. координаты необходимые для их рендеринга легко рассчитываются.

**Реализация интерфейса в отношении управления камерой**

Управления камерой производится с помощью клавиатуры

W,A,S,D - поворот камеры

R,T - приблизиться к объекту, отдалиться от объекта.

F,G - увеличить, уменьшить влияние перспективы (поле зрения)

Стрелки - перемещение экранной плоскости

Дополнительный функционал:

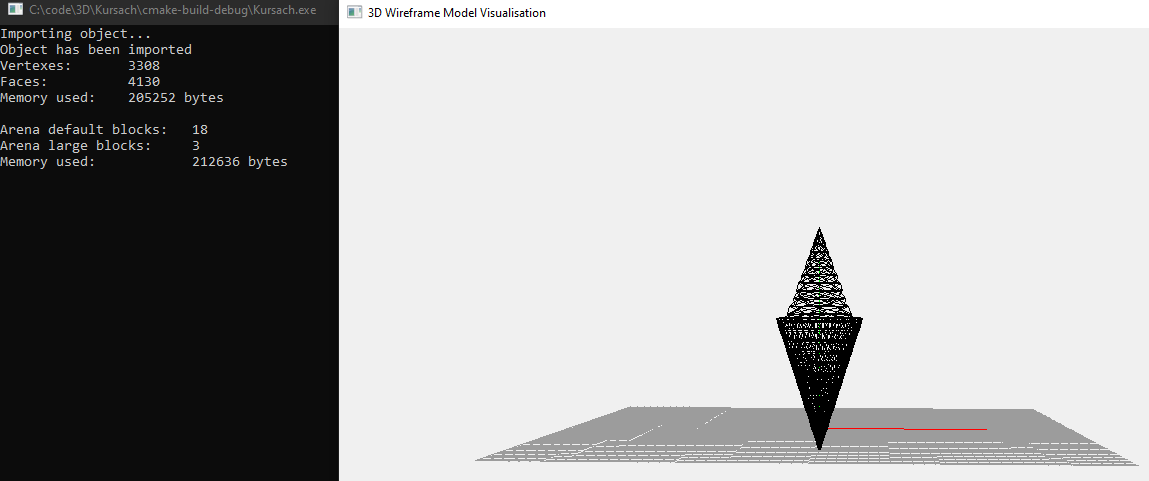
F5 – вывод положения камеры в консоль

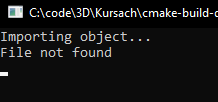
**Отладка и тестирование программы**

Таблица 1 – Описание поведения программы при тестировании

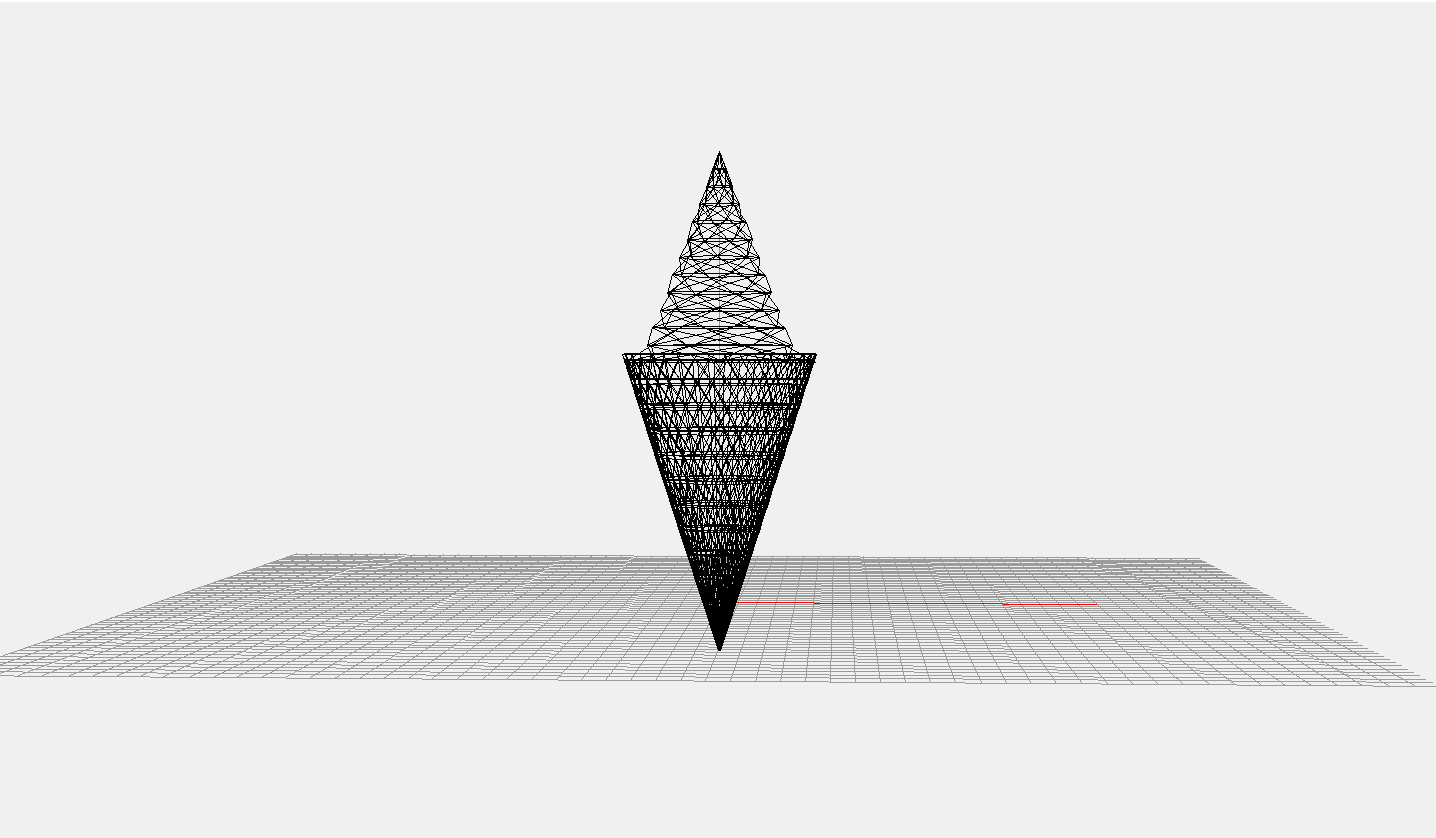
|  |  |  |
| --- | --- | --- |
| Выполнение действия | Ожидаемый результат | Полученный результат |
| Загрузка существующего файла | Импорт файла, рисование модели | Импорт файла, рисование модели |
| Загрузка несуществующего файла | Аварийное завершение программы | Аварийное завершение программы |
| Нажатие клавиш поворота камера | Поворот проекции | Поворот проекции |
| Нажатие клавиш приближения, удаления камеры | Приближение, удаление проекции | Приближение, удаление проекции |
| Нажатия клавиш перемещения экранной плоскости | Перемещение экранной плоскости | Перемещение экранной плоскости |
| Нажатие клавиш изменения поля зрения | Увеличение, уменьшение поля зрения | Увеличение, уменьшение поля зрения только до некоторого значения, |
| Нажатие клавиши, для вывода информации о положении камеры | Вывод информации о положении камеры | Вывод информации о положении камеры |

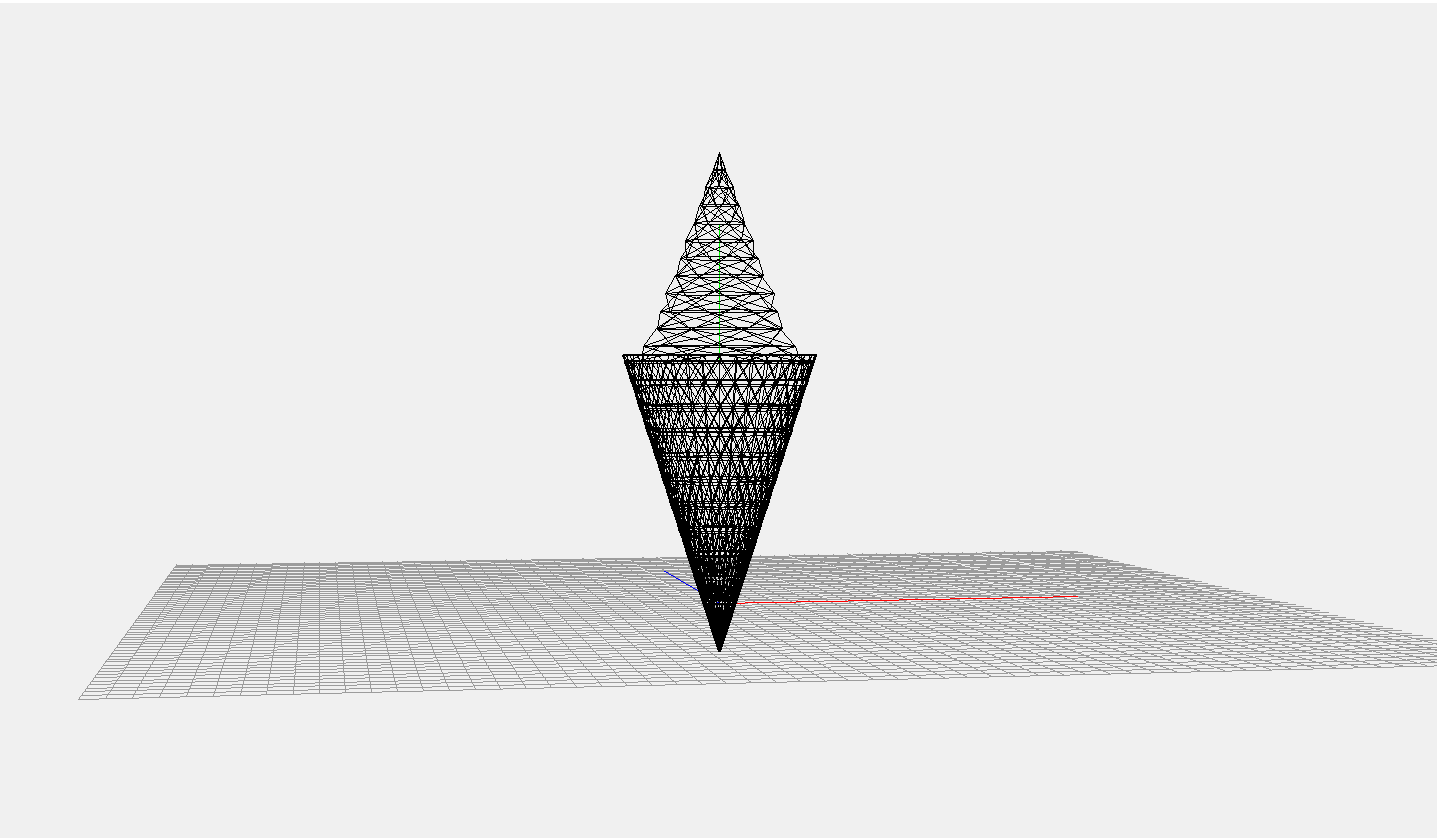
Загрузка файла:

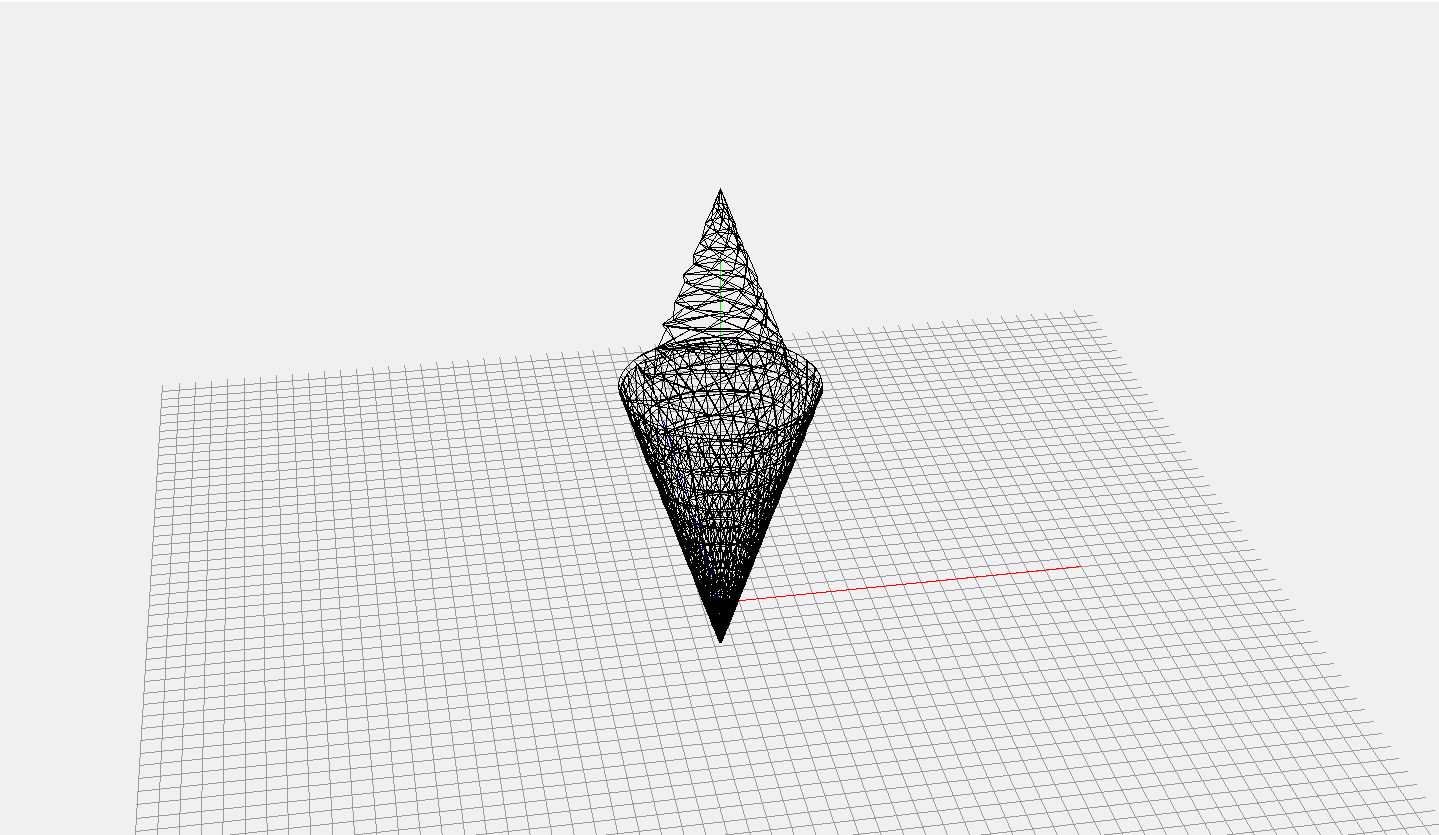


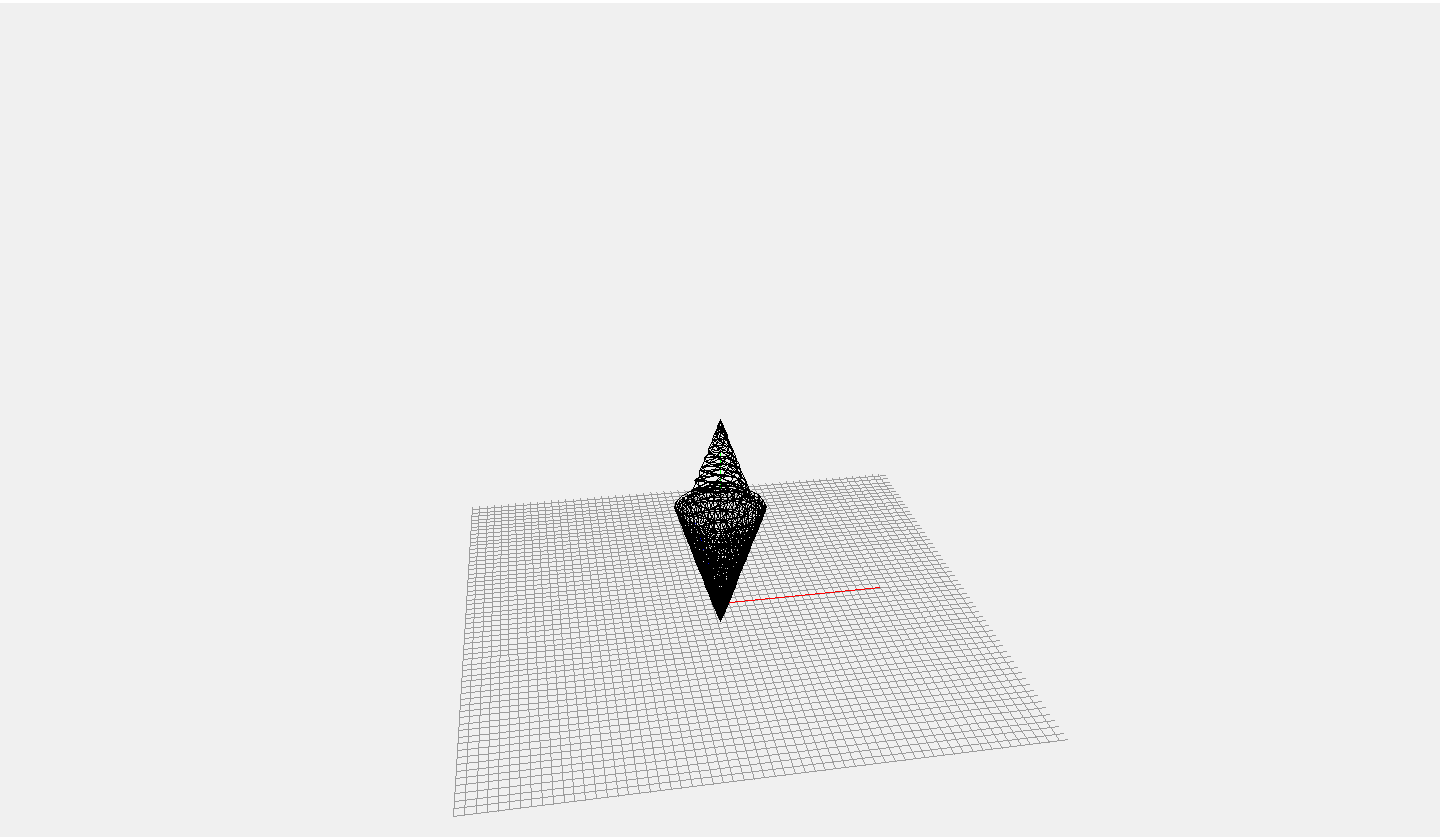


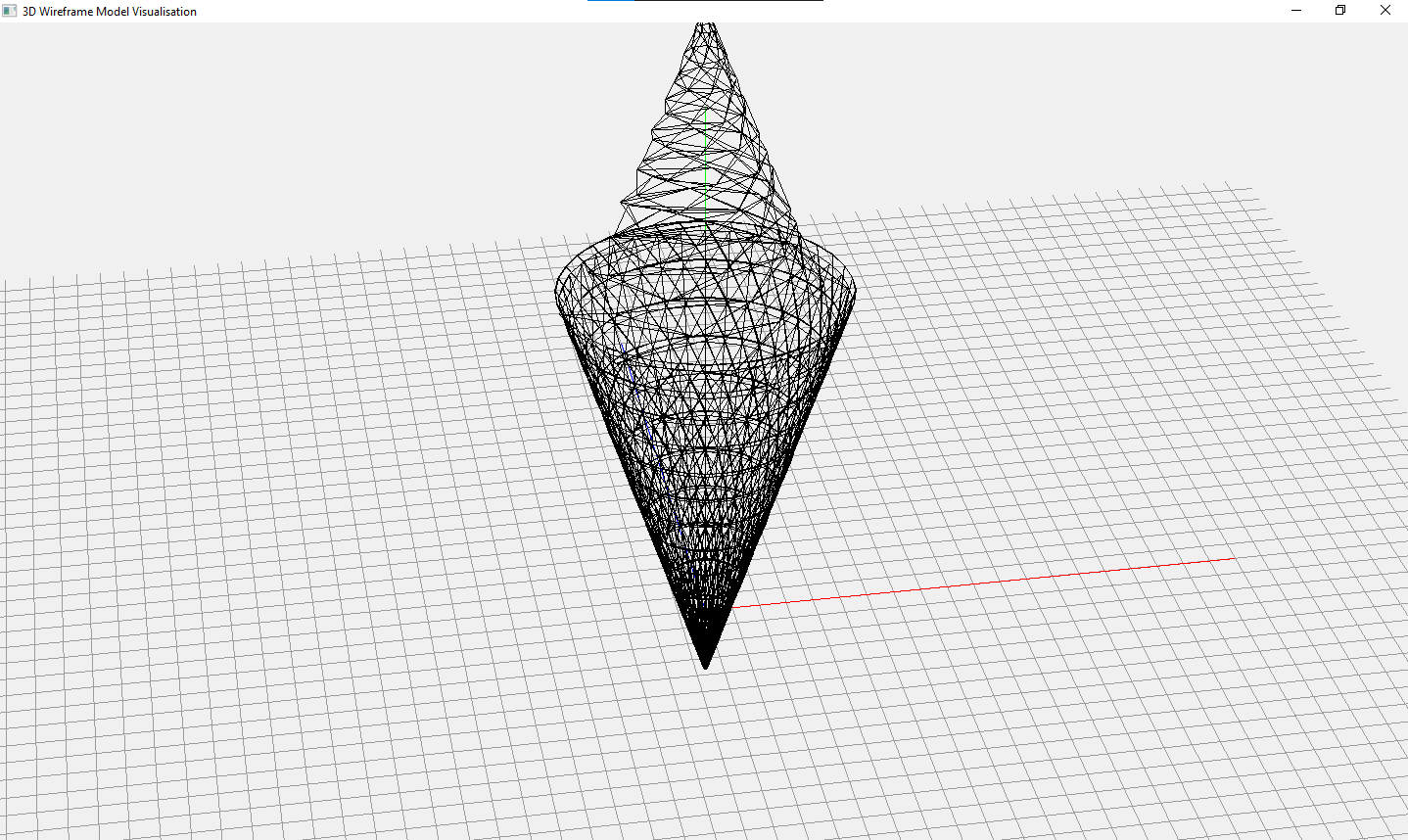
Перемещение камеры:



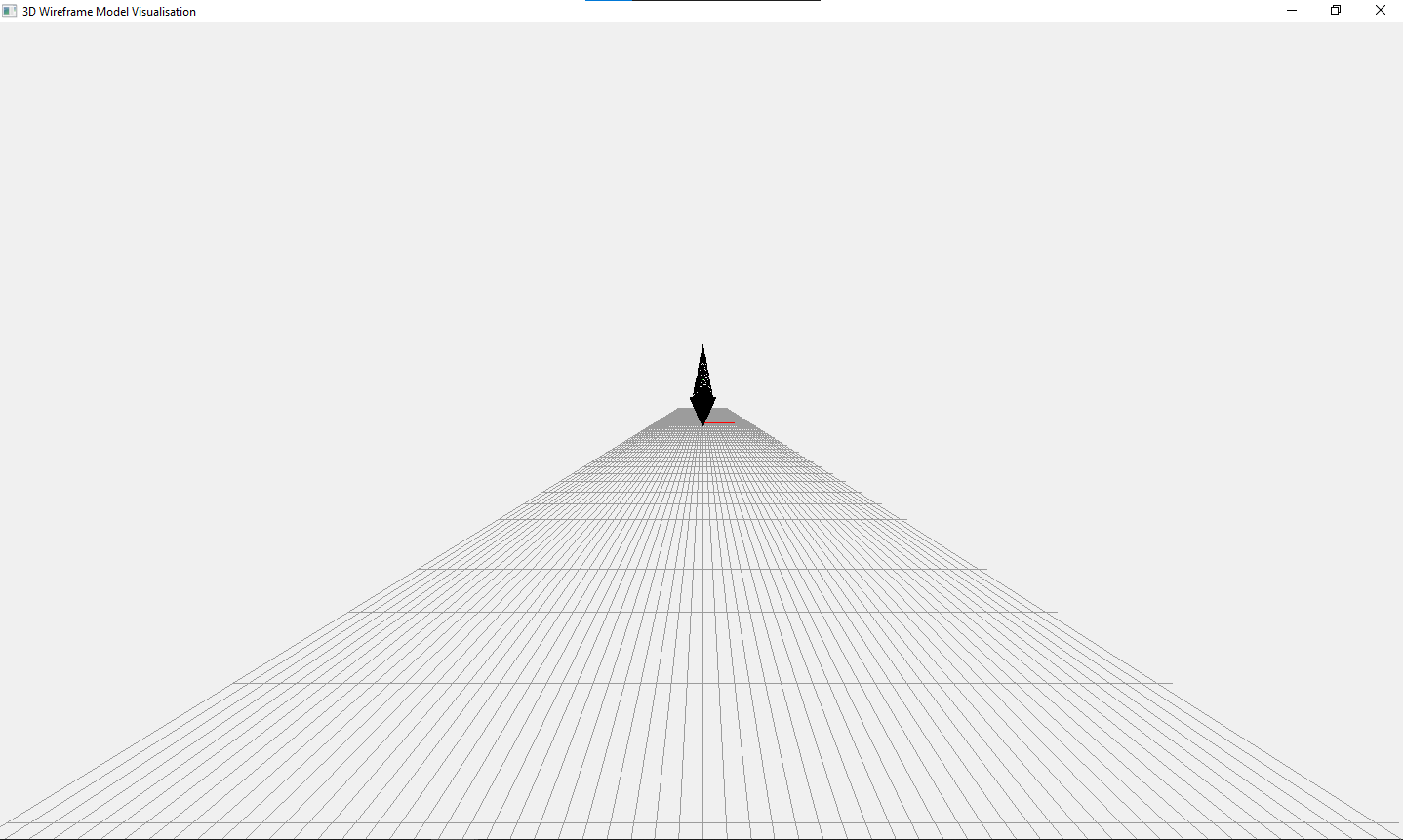


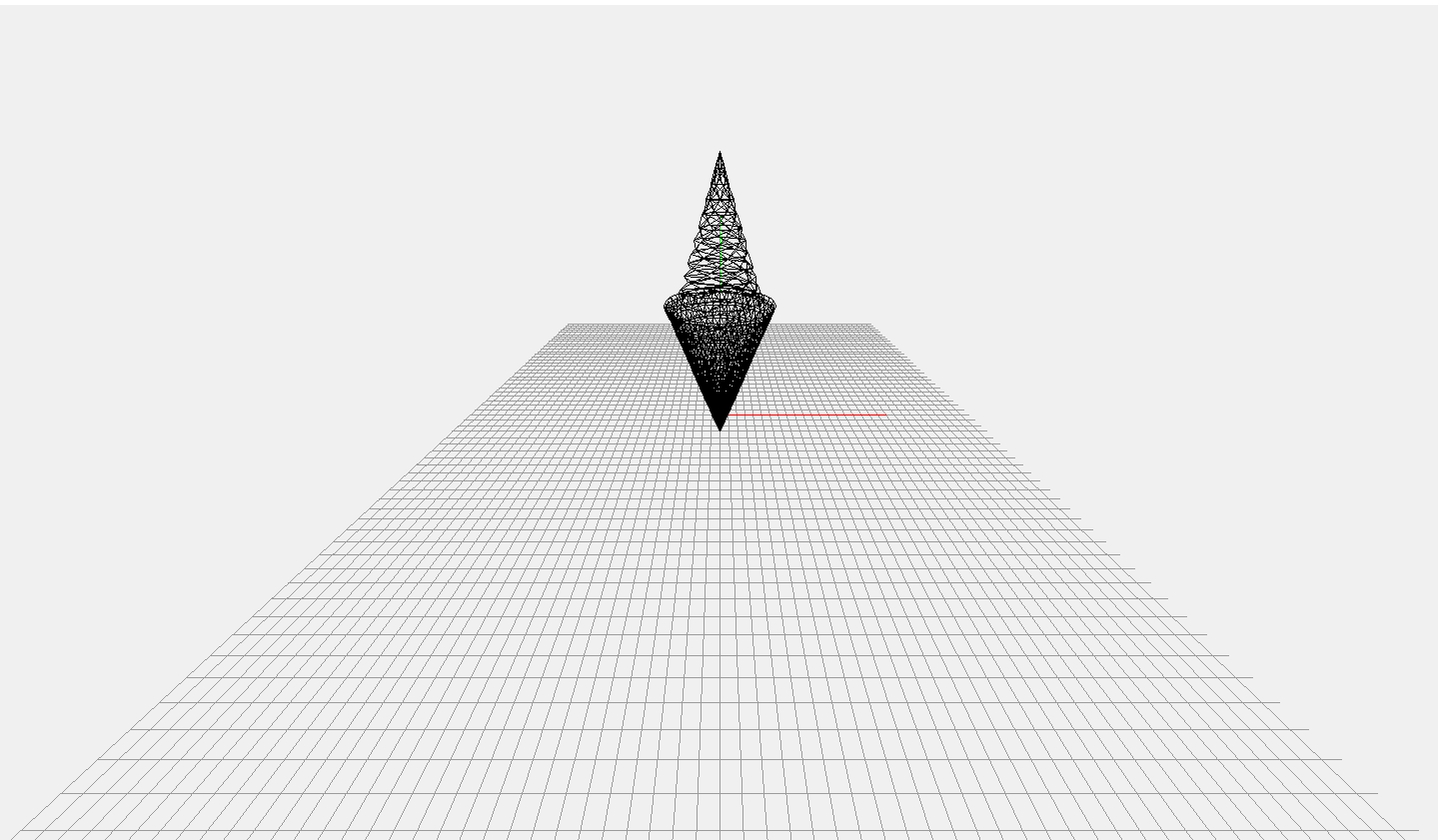




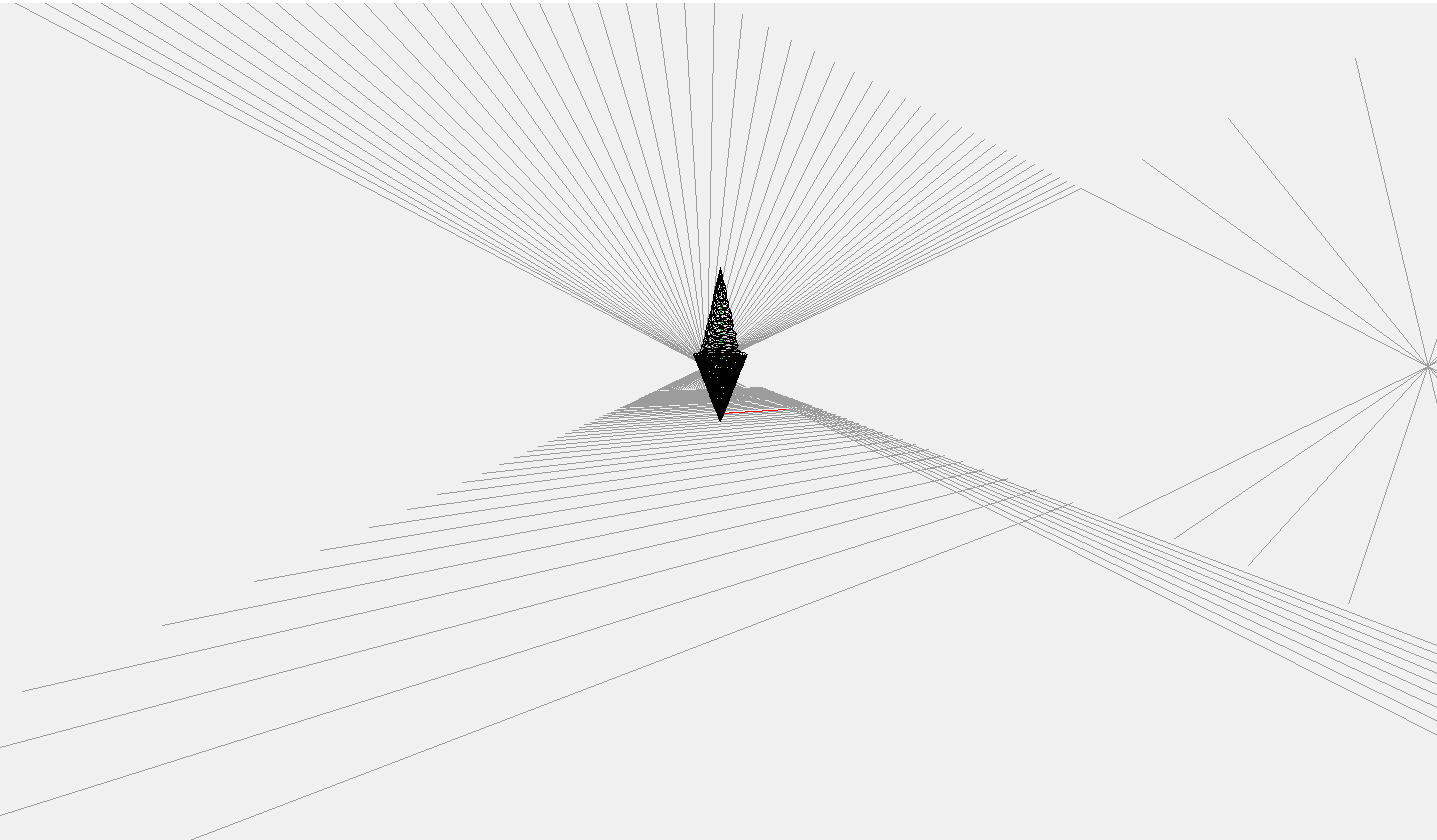


Изменение поля зрения

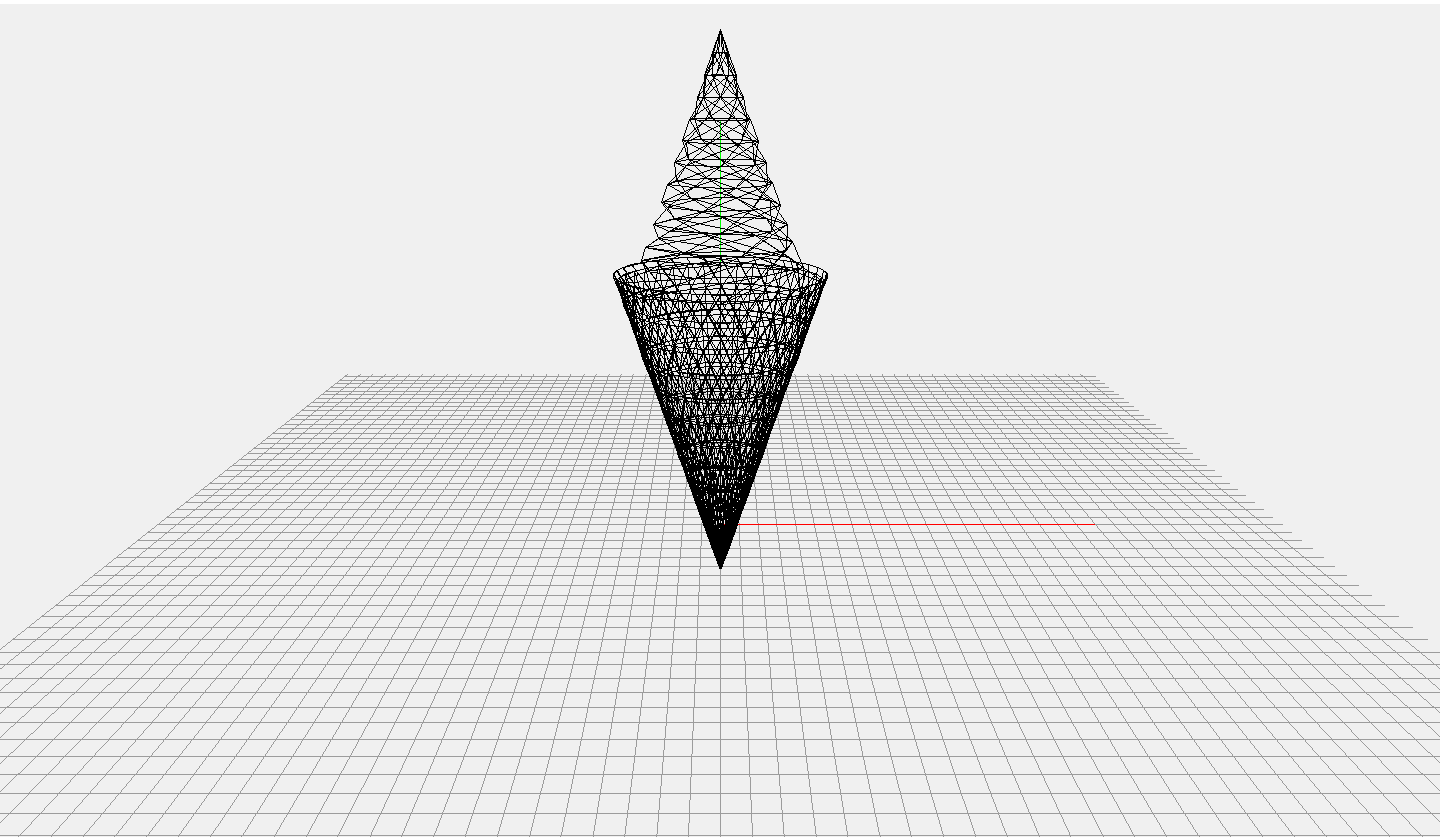


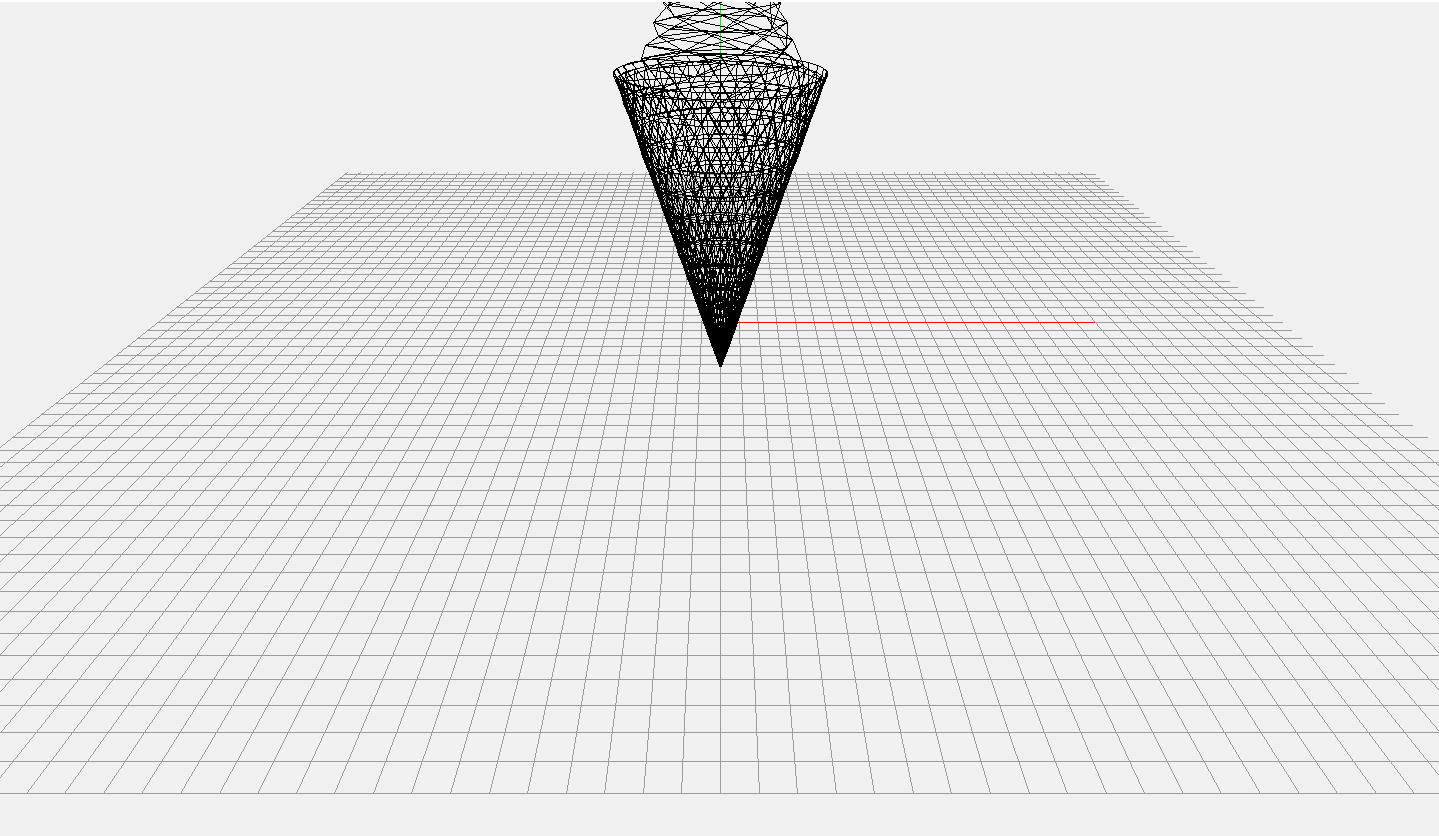


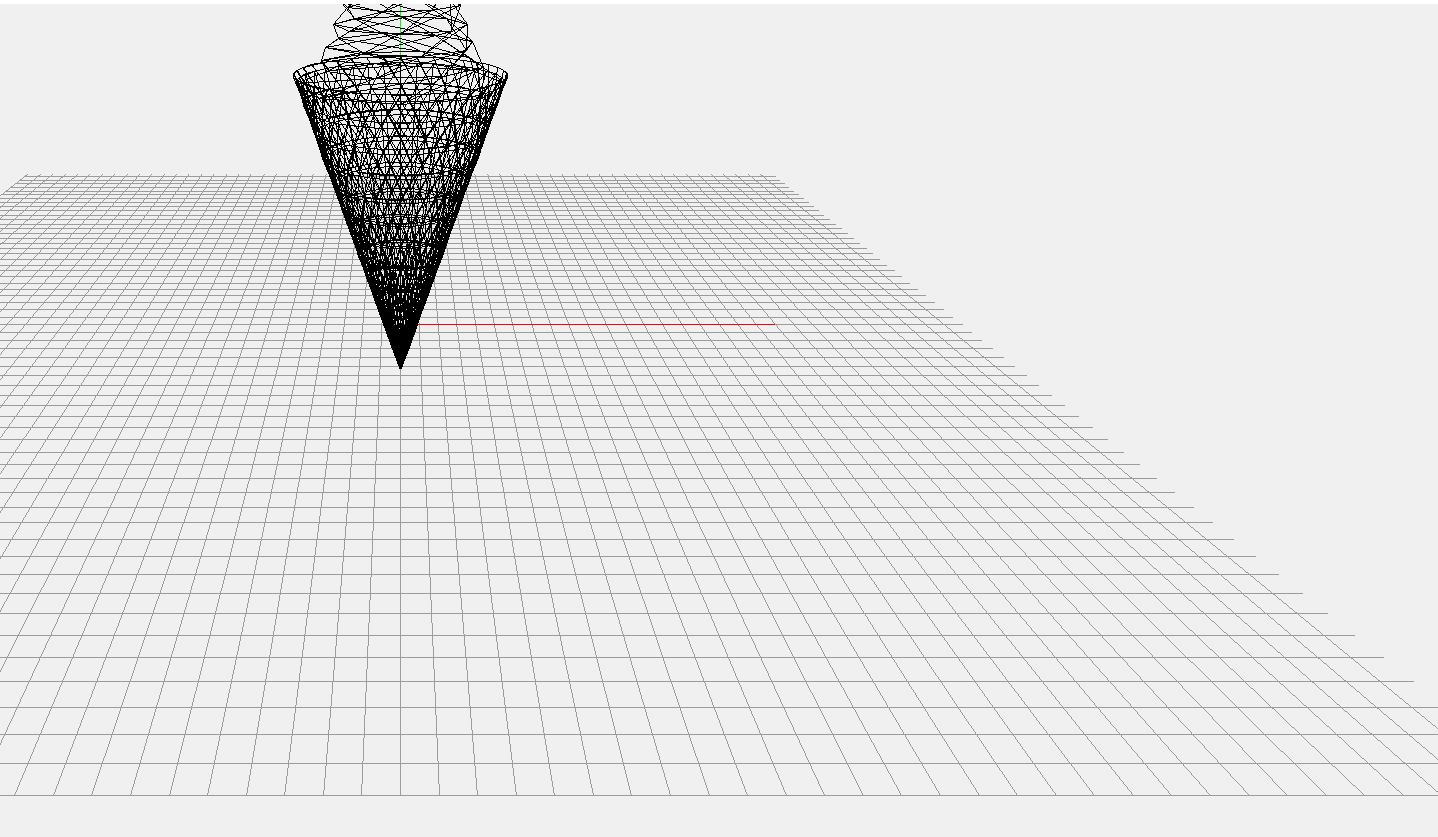
Дефекты, вызванные большим увеличением поля зрения. Возможная причина – переполнение значений координат.



Перемещение экранной плоскости:







Информация о положении камеры:

