МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Череповецкий государственный университет»

**Лабораторная работа № 2**

**«Удаление невидимых граней»**

**Выполнил:**

студент гр. 1ИВТпб-01-31оп

Климов А.Г.

**Проверил:** преподаватель

Юдина О.В., доцент  
Отметка о зачете:

Череповец

2017 год

**Цель работы**

Изучить метод не лицевых граней, освоить на практике удаление не лицевых граней с помощью пирамиды отсечения в любой сцене.

**Задание**

Написать программу, реализующую построение части сцены, попадающей в пирамиду отсечения, связанной с камерой. Отображаться должны только лицевые грани. Сцена должна представлять собой изображение трех фигур; куб, шар, пирамида, касающихся друг друга.

Интерфейс должен предусматривать изменение положения и направления камеры просмотра. То есть, должна быть реализована возможность «обхода» комбинации фигур.

**Текст программы**

#include <GL/glut.h>

#include <iostream>

double rotate\_y = 0;

double rotate\_x = 0;

float x = 0.0f, z = 5.0f;

float frustum[6][4];

void ExtractFrustum()

{

float proj[16];

float modl[16];

float clip[16];

float t;

// Узнаем текущую матрицу PROJECTION

glGetFloatv(GL\_PROJECTION\_MATRIX, proj);

// Узнаем текущую матрицу MODELVIEW

glGetFloatv(GL\_MODELVIEW\_MATRIX, modl);

// Комбинируем матрицы(перемножаем)

clip[0] = modl[0] \* proj[0] + modl[1] \* proj[4] + modl[2] \* proj[8] + modl[3] \* proj[12];

clip[1] = modl[0] \* proj[1] + modl[1] \* proj[5] + modl[2] \* proj[9] + modl[3] \* proj[13];

clip[2] = modl[0] \* proj[2] + modl[1] \* proj[6] + modl[2] \* proj[10] + modl[3] \* proj[14];

clip[3] = modl[0] \* proj[3] + modl[1] \* proj[7] + modl[2] \* proj[11] + modl[3] \* proj[15];

clip[4] = modl[4] \* proj[0] + modl[5] \* proj[4] + modl[6] \* proj[8] + modl[7] \* proj[12];

clip[5] = modl[4] \* proj[1] + modl[5] \* proj[5] + modl[6] \* proj[9] + modl[7] \* proj[13];

clip[6] = modl[4] \* proj[2] + modl[5] \* proj[6] + modl[6] \* proj[10] + modl[7] \* proj[14];

clip[7] = modl[4] \* proj[3] + modl[5] \* proj[7] + modl[6] \* proj[11] + modl[7] \* proj[15];

clip[8] = modl[8] \* proj[0] + modl[9] \* proj[4] + modl[10] \* proj[8] + modl[11] \* proj[12];

clip[9] = modl[8] \* proj[1] + modl[9] \* proj[5] + modl[10] \* proj[9] + modl[11] \* proj[13];

clip[10] = modl[8] \* proj[2] + modl[9] \* proj[6] + modl[10] \* proj[10] + modl[11] \* proj[14];

clip[11] = modl[8] \* proj[3] + modl[9] \* proj[7] + modl[10] \* proj[11] + modl[11] \* proj[15];

clip[12] = modl[12] \* proj[0] + modl[13] \* proj[4] + modl[14] \* proj[8] + modl[15] \* proj[12];

clip[13] = modl[12] \* proj[1] + modl[13] \* proj[5] + modl[14] \* proj[9] + modl[15] \* proj[13];

clip[14] = modl[12] \* proj[2] + modl[13] \* proj[6] + modl[14] \* proj[10] + modl[15] \* proj[14];

clip[15] = modl[12] \* proj[3] + modl[13] \* proj[7] + modl[14] \* proj[11] + modl[15] \* proj[15];

// Находим A, B, C, D для ПРАВОЙ плоскости

frustum[0][0] = clip[3] - clip[0];

frustum[0][1] = clip[7] - clip[4];

frustum[0][2] = clip[11] - clip[8];

frustum[0][3] = clip[15] - clip[12];

// Приводим уравнение плоскости к нормальному виду

t = sqrt(frustum[0][0] \* frustum[0][0] + frustum[0][1] \* frustum[0][1] + frustum[0][2] \* frustum[0][2]);

frustum[0][0] /= t;

frustum[0][1] /= t;

frustum[0][2] /= t;

frustum[0][3] /= t;

// Находим A, B, C, D для ЛЕВОЙ плоскости

frustum[1][0] = clip[3] + clip[0];

frustum[1][1] = clip[7] + clip[4];

frustum[1][2] = clip[11] + clip[8];

frustum[1][3] = clip[15] + clip[12];

// Приводим уравнение плоскости к нормальному виду

t = sqrt(frustum[1][0] \* frustum[1][0] + frustum[1][1] \* frustum[1][1] + frustum[1][2] \* frustum[1][2]);

frustum[1][0] /= t;

frustum[1][1] /= t;

frustum[1][2] /= t;

frustum[1][3] /= t;

// Находим A, B, C, D для НИЖНЕЙ плоскости

frustum[2][0] = clip[3] + clip[1];

frustum[2][1] = clip[7] + clip[5];

frustum[2][2] = clip[11] + clip[9];

frustum[2][3] = clip[15] + clip[13];

// Приводим уравнение плоскости к нормальному

t = sqrt(frustum[2][0] \* frustum[2][0] + frustum[2][1] \* frustum[2][1] + frustum[2][2] \* frustum[2][2]);

frustum[2][0] /= t;

frustum[2][1] /= t;

frustum[2][2] /= t;

frustum[2][3] /= t;

//ВЕРХНЯЯ плоскость

frustum[3][0] = clip[3] - clip[1];

frustum[3][1] = clip[7] - clip[5];

frustum[3][2] = clip[11] - clip[9];

frustum[3][3] = clip[15] - clip[13];

// Нормальный вид

t = sqrt(frustum[3][0] \* frustum[3][0] + frustum[3][1] \* frustum[3][1] + frustum[3][2] \* frustum[3][2]);

frustum[3][0] /= t;

frustum[3][1] /= t;

frustum[3][2] /= t;

frustum[3][3] /= t;

// ЗАДНЯЯ плоскость

frustum[4][0] = clip[3] - clip[2];

frustum[4][1] = clip[7] - clip[6];

frustum[4][2] = clip[11] - clip[10];

frustum[4][3] = clip[15] - clip[14];

// Нормальный вид

t = sqrt(frustum[4][0] \* frustum[4][0] + frustum[4][1] \* frustum[4][1] + frustum[4][2] \* frustum[4][2]);

frustum[4][0] /= t;

frustum[4][1] /= t;

frustum[4][2] /= t;

frustum[4][3] /= t;

// ПЕРЕДНЯЯ плоскость

frustum[5][0] = clip[3] + clip[2];

frustum[5][1] = clip[7] + clip[6];

frustum[5][2] = clip[11] + clip[10];

frustum[5][3] = clip[15] + clip[14];

// Нормальный вид

t = sqrt(frustum[5][0] \* frustum[5][0] + frustum[5][1] \* frustum[5][1] + frustum[5][2] \* frustum[5][2]);

frustum[5][0] /= t;

frustum[5][1] /= t;

frustum[5][2] /= t;

frustum[5][3] /= t;

}

bool SphereInFrustum(float x, float y, float z, float radius)

{

int p;

for (p = 0; p < 6; p++)

if (frustum[p][0] \* x + frustum[p][1] \* y + frustum[p][2] \* z + frustum[p][3] <= -radius)

return false;

return true;

}

bool CubeInFrustum(float x, float y, float z, float size)

{

int p;

for (p = 0; p < 6; p++)

{

if (frustum[p][0] \* (x - size) + frustum[p][1] \* (y - size) + frustum[p][2] \* (z - size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x + size) + frustum[p][1] \* (y - size) + frustum[p][2] \* (z - size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x - size) + frustum[p][1] \* (y + size) + frustum[p][2] \* (z - size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x + size) + frustum[p][1] \* (y + size) + frustum[p][2] \* (z - size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x - size) + frustum[p][1] \* (y - size) + frustum[p][2] \* (z + size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x + size) + frustum[p][1] \* (y - size) + frustum[p][2] \* (z + size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x - size) + frustum[p][1] \* (y + size) + frustum[p][2] \* (z + size) + frustum[p][3] > 0)

continue;

if (frustum[p][0] \* (x + size) + frustum[p][1] \* (y + size) + frustum[p][2] \* (z + size) + frustum[p][3] > 0)

continue;

return false;

}

return true;

}

void changeSize(int w, int h) {

// предотвращение деления на ноль

if (h == 0)

h = 1;

float ratio = w \* 1.0 / h;

// используем матрицу проекции

glMatrixMode(GL\_PROJECTION);

// обнуляем матрицу

glLoadIdentity();

// установить параметры

glViewport(0, 0, w, h);

// установить корректную перспективу

gluPerspective(45.0f, ratio, 0.1f, 100.0f);

// вернуться к матрице проекции

glMatrixMode(GL\_MODELVIEW);

}

void Draw()

{

//КУБ

//Лицевая грань

glBegin(GL\_QUADS);

glColor3f(0.0f, 1.0f, 0.0f); //Зелёный

glVertex3f(-0.5, -0.5, 0.5);

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(0.5, 0.5, 0.5);

glVertex3f(-0.5, 0.5, 0.5);

glEnd();

//Задняя грань

glBegin(GL\_QUADS);

glColor3f(1.0f, 0.0f, 0.0f); //Красный

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(0.5, -0.5, -0.5);

glVertex3f(0.5, 0.5, -0.5);

glVertex3f(-0.5, 0.5, -0.5);

glEnd();

// Верхняя грань

glBegin(GL\_QUADS);

glColor3f(0.0f, 0.0f, 1.0f); //Синий

glVertex3f(-0.5, 0.5, -0.5);

glVertex3f(-0.5, 0.5, 0.5);

glVertex3f(0.5, 0.5, 0.5);

glVertex3f(0.5, 0.5, -0.5);

glEnd();

//Нижняя грань

glBegin(GL\_QUADS);

glColor3f(1.0f, 1.0f, 0.0f); //Жёлтый

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(-0.5, -0.5, 0.5);

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(0.5, -0.5, -0.5);

glEnd();

//Левая грань

glBegin(GL\_QUADS);

glColor3f(1.0f, 0.5f, 0.0f); //Оранжевый

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(-0.5, 0.5, -0.5);

glVertex3f(-0.5, 0.5, 0.5);

glVertex3f(-0.5, -0.5, 0.5);

glEnd();

//Правая грань

glBegin(GL\_QUADS);

glColor3f(1.0f, 0.0f, 1.0f); //Фиолетовый

glVertex3f(0.5, -0.5, -0.5);

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(0.5, 0.5, 0.5);

glVertex3f(0.5, 0.5, -0.5);

glEnd();

glTranslatef(0.0, 1.0, 0.0);

//ПИРАМИДА

glBegin(GL\_TRIANGLES);

glColor3f(0.0, 1.0, 0.0); //Зелёный

glVertex3f(0.5, -0.5, -0.5);

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(0.0, 0.5, 0.0);

glEnd();

glBegin(GL\_TRIANGLES);

glColor3f(1.0, 0.84, 0.0); //Жёлтый

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(-0.5, -0.5, 0.5);

glVertex3f(0.0, 0.5, 0.0);

glEnd();

glBegin(GL\_TRIANGLES);

glColor3f(1.0, 0.0, 0.0); //Красный

glVertex3f(-0.5, -0.5, 0.5);

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(0.0, 0.5, 0.0);

glEnd();

glBegin(GL\_TRIANGLES);

glColor3f(1.0, 1.0, 1.0); //Белый

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(0.5, -0.5, -0.5);

glVertex3f(0.0, 0.5, 0.0);

glEnd();

//Основание пирамиды

glBegin(GL\_QUADS);

glColor3f(0.0, 0.0, 1.0); //Синий

glVertex3f(0.5, -0.5, 0.5);

glVertex3f(-0.5, -0.5, 0.5);

glVertex3f(-0.5, -0.5, -0.5);

glVertex3f(0.5, -0.5, -0.5);

glEnd();

}

void DrawSphere()

{

glTranslatef(2.0f, -1.0f, 2.0f);

glPushMatrix();

glTranslatef(-2.0f, 0.0f, -1.0f);

glColor3f(0.8, 0.2, 0.1);

glScalef(0.7, 0.7, 0.7);

glutSolidSphere(0.75f, 20, 20);

glPopMatrix();

}

void Floor()

{

glTranslatef(-2.0f, -0.2f, -2.0f);

glScalef(0.5, 0.5, 0.5);

glBegin(GL\_QUADS);

glColor3f(1.0, 1.0, 1.0);

glVertex3f(-5.0, -0.7, 5.0);

glVertex3f(5.0, -0.7, 5.0);

glVertex3f(5.0, -0.7, -5.0);

glVertex3f(-5.0, -0.7, -5.0);

glEnd();

}

void renderScene(void) {

//очистить буфер цвета и глубины

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// обнулить трансформацию

glLoadIdentity();

// установить камеру

gluLookAt(x + 3, 1.0f, z + 4,

2, 1.0f, 4,

0.0f, 1.0f, 0.0f);

ExtractFrustum();

glPushMatrix();

glTranslatef(0, 0, -5);

glRotatef(rotate\_x, 0.0, 1.0, 0.0);

Draw();

DrawSphere();

Floor();

glPopMatrix();

glutSwapBuffers();

}

void specialKeys(int key, int x, int y) {

switch (key) {

case GLUT\_KEY\_RIGHT:

rotate\_x += 5;

break;

case GLUT\_KEY\_LEFT:

rotate\_x -= 5;

break;

}

}

int main(int argc, char\* argv[]) {

glutInit(&argc, argv); // Инициализация GLUT и обработка пользовательских параметров

// Запрашиваем окно с поддержкой двойной буферизации, z-буферизации и цветовой схемы True Color

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

// Задаём позицию окна

glutInitWindowPosition(500, 200);

// Задаём размер окна

glutInitWindowSize(400, 400);

// Создаем окно

glutCreateWindow("Лабораторная работа 2");

glClearColor(0.0, 0.0, 0.0, 1.0);

// регистрация

glutDisplayFunc(renderScene);

glutReshapeFunc(changeSize);

glutIdleFunc(renderScene);

glutSpecialFunc(specialKeys);

// OpenGL - инициализация функции теста

glEnable(GL\_DEPTH\_TEST);

// Передаем контроль над событиями в GLUT

glutMainLoop();

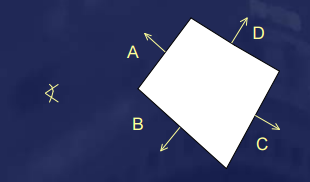
return 0;

}

**Контрольные вопросы**

*1. Что такое не лицевая грань?*

Если грани являются границей тела (или нескольких тел), то для каждой из них можно определить вектор внешней нормали A, B, C, D.



Нормали к граням А и В смотрят в сторону наблюдателя (наблюдатель находится в положительном полупространстве по отношению к плоскости, проходящей через соответствующую грань). Такие грани называются лицевыми. Для граней C и D нормали направлены от наблюдателя, их называют не лицевыми.

В случае, когда грани являются границей тела (или нескольких тел), то ни одна из не лицевых граней не может быть видна даже частично – любая из них всегда будет закрываться от наблюдателя лицевыми гранями. При определении видимости все не лицевые грани можно всегда отбрасывать, что сокращает число рассматриваемых граней примерно вдвое (в общем случае количество лицевых граней примерно равно количеству лицевых, т.е. составляет половину от общего числа граней). Когда вся сцена состоит из одного выпуклого объекта, то все лицевые грани и только они будут видны, причем полностью.

*2. Что необходимо для вычисления вектора внешней нормали?*

Видовая матрица отвечает за систему координат создаваемой трехмерной

Для вычисления вектора внешней нормали необходимо знать три точки на плоскости. Пусть эти точки описаны радиусами векторов A, B, C и центром тела - точка P. Тогда вектор нормали N будет равен

N = (B-A)\*(C-B). Если N\*(A-P)>=0, то N - вектор внешней нормали; если N\*(A-P)<0, то вектор внешней нормали N = -N.

Для выпуклого тела центр можно вычислить как

, где Vi – вершины тела.

Оценку угла можно провести с помощью косинуса. Косинус острого угла всегда положителен, а тупого - отрицателен. Так как скалярное произведение двух векторов равно

V1\*V2=|V1| \* | V2| \*cos a,

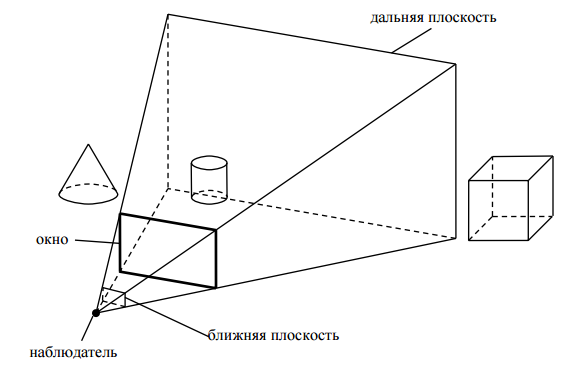
то знак скалярного произведения будет зависеть от косинуса угла.

*3. Опишите метод построения тела с помощью пирамиды отсечения.*

Отсечение — метод оптимизации в компьютерной графике, когда компьютер прорисовывает только ту часть сцены, которая может находиться в поле зрения пользователя.

Пирамида зрения представляет собой объём пространства, который виден наблюдателю через окно. При перспективном проецировании она ограничена плоскостями, построенными на точке наблюдателя и сторонах окна. Кроме того, она часто ограничивается ещё двумя плоскостями, параллельными окну – ближней и дальней.

Ближняя плоскость необходима для того, чтобы отсечь объекты находящиеся очень близко к наблюдателю и сзади него. Даже если основное отсечение происходит в картинной плоскости, без участия пирамиды зрения, отсечение ближней плоскостью необходимо, чтобы правильно спроецировать вершины объектов на картинную плоскость. Перспективное проецирование требует деления на координату глубины (z), и в случае, когда она отрицательная, проекция получается неверной, а когда нулевая - происходит ошибка деления на ноль. Дальняя плоскость используется для сокращения числа отображаемых объектов, ей отсекаются объекты, расположенные дальше некоторого расстояния от наблюдателя.



*4. Опишите метод построения тела с помощью плоскости отсечения.*

Если нужно несколько раз обращаться к одной и той же группе команд, эти

Отсечение — метод оптимизации в компьютерной графике, когда компьютер прорисовывает только ту часть сцены, которая может находиться в поле зрения пользователя.

Алгоритм Робертса позволяет определить, какие рёбра или части рёбер объектов сцены видимы, а какие заслонены гранями других объектов. Объекты проецируются на картинную плоскость, и анализ видимости происходит на плоскости. Каждое ребро последовательно сравнивается со всеми гранями, обычно, выпуклыми многоугольниками. Могут быть следующие варианты их взаимного расположения:

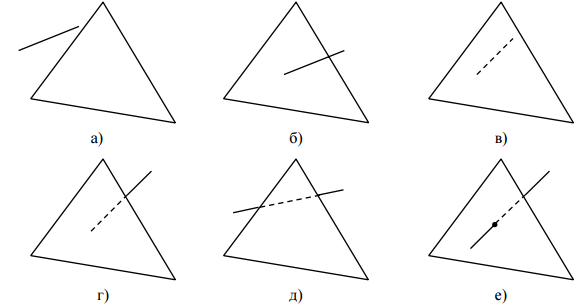
- проекции ребра и грани не пересекаются, грань не заслоняет ребро(а);

- проекции ребра и грани пересекаются, но ребро лежит ближе грани, грань не заслоняет ребро (б);

- проекции ребра и грани пересекаются, и ребро лежит дальше грани, грань заслоняет всё ребро (в) или часть ребра (г, д);

- ребро пересекает грань (е).

Если ребро не заслоняется гранью, то оно сравнивается со следующей гранью. Если ребро заслоняется полностью, то оно невидимо и сравнивать его со следующими гранями не нужно. Если грань заслоняет часть ребра, то нужно разделить отрезок на части, невидимую часть отбросить, а видимые сравнивать дальше с остальными гранями. Выпуклая грань может заслонить лишь одну часть отрезка, поэтому не заслонёнными останутся не более двух частей (г, д, е).



Плоскость отсечения. Пусть дан массив точек V[k], параметры плоскости отсечения N и d. Если j-я и j+1-я точки расположены по разные стороны от плоскости, то вычисляется точка пересечения P отрезка (Vi,Vi+1) с плоскостью. В выходной массив заноситься точка Р. Если j-я точка лежит во внутренней полупространстве, то добавляется в выходной массив точка Vi.