МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА» (БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №5

по дисциплине: Компьютерная графика

тема: «Алгоритмы удаления невидимых поверхностей»

Выполнил: ст. группы ПВ-223

Дмитриев Андрей Александрович

Проверил:

Осипов Олег Васильевич

Белгород 2024 г.

**Цель работы:** изучить алгоритмы удаления невидимых поверхностей и создать программу для визуализации объёмной трёхмерной модели с закрашенными гранями.

**Порядок выполнения работы**

1. Разработать алгоритм и составить программу для построения на

экране трѐхмерных изображений в соответствии с номером варианта.

В качестве исходных данных взять указанные в таблице №1.

**Требования к программе**

1. В программе по центру окна должна отображаться только центральная или ортографическая проекция фигуры по выбору пользователя. Грани фигуры должны быть закрашены и отсортированы по глубине, т.е. пользователь должен видеть передние грани. Перед растеризацией грани должны быть отсортированы по алгоритму художника (на пятёрку) или по глубине (на четвёрку). Сортировку методом z-буфера, которая уже реализована в предыдущей лабораторной работе, временно отключить.
2. Пользователь должен иметь возможность поворачивать и перемещать фигуру вдоль оси 𝑧 с использованием мыши. Поворот фигуры лучше выполнять при обработке события WM\_MOUSEMOVE при зажатой кнопке мыши, перенос – при обработке события WM\_MOUSEWHEEL. Фигура должна поворачиваться «вслед» за мышью. Это значит, что, если мышь перемещается влево-вправо, то фигура должна поворачиваться вокруг вертикальной оси; если вверх-вниз, то вокруг горизонтальной оси. 3. В программе должна быть предусмотрена возможность изменять прозрачность граней с помощью клавиш или мыши.
3. Наложить на одну или несколько граней объекта шейдер с какой-либо текстурой (кирпичная кладка, сетка, горошек, пчелиные соты или любая другая). Для проверки правильности работы растеризатора наложить на одну отдельную квадратную грань шахматную текстуру. Текстура не должна искажаться перспективой.

**Содержание отчѐта**

1. Название темы.

2. Цель работы.

3. Постановка задачи.

4. Вывод необходимых формул для построения граней фигуры.

5. Вывод необходимых формул для построения всех проекций. Указать

какие матрицы используются для построения всех четырѐх проекций

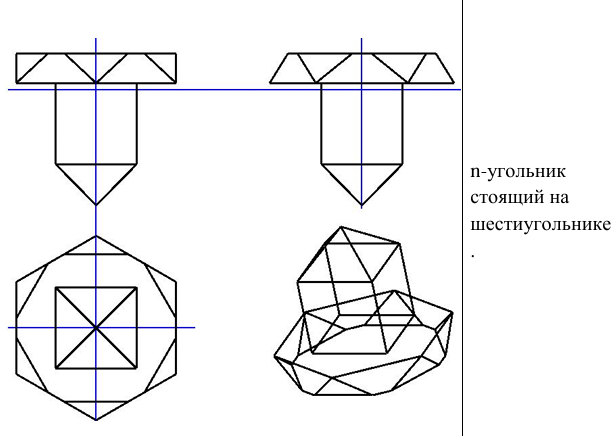
изображений и в какой последовательности они умножаются.

6. Текст программы.

7. Результат работы программы (снимки экрана).

8. Вывод о проделанной работе

Вариант 2:



Ход работы:

Пространство, в котором будет построена фигура – куб 1на1на1.

Начнём построение со «шляпки». Поворотом по оси Z отметим точки 6-угольника сверху, затем по среднему между полученными точками определим нижнюю часть 6-угольника и парами точек определим стенки n-угольника:

const std::vector<Vector> get\_points() {  
 std::vector<Vector> result;  
  
 result.emplace\_back(0, 0, -0.9); // 0 - нижняя точка ножки  
 result.emplace\_back(0, 0, 0.5); // 1 - верхняя точка ножки  
 result.emplace\_back(0, 0, 0.7); // 2 - верхняя точка шлипки  
  
 result.emplace\_back(0.4, 0, -0.5); // 3..7 - нижний многоугольник  
 Matrix rot\_z\_matrix = Matrix::RotationZ(360 / 5 \* M\_PI / 180);  
 for (int i = 1; i < 5; i++) {  
 Vector temp = result.back() \* rot\_z\_matrix;  
 result.emplace\_back(temp.x, temp.y, temp.z);  
 }  
  
 result.emplace\_back(0.4, 0, 0.5); // 8..12 - верхний многоугольник  
 for (int i = 1; i < 5; i++) {  
 Vector temp = result.back() \* rot\_z\_matrix;  
 result.emplace\_back(temp.x, temp.y, temp.z);  
 }  
  
 rot\_z\_matrix = Matrix::RotationZ(60 \* M\_PI / 180);  
 std::vector<Vector> sub; // 13..18 - верхний многоугольник  
 sub.emplace\_back(0.7, 0, 0.7);  
 for (int i = 1; i < 6; i++) {  
 Vector temp = sub[i - 1] \* rot\_z\_matrix;  
 sub.emplace\_back(temp.x, temp.y, temp.z);  
 }  
  
 result.insert(result.end(), sub.begin(), sub.end());  
  
 for (int i = 0; i < 6; i++) { // 19..24 - нижний многоугольник  
 result.emplace\_back((sub[i].x + sub[(i + 1) % 6].x) / 2,  
 (sub[i].y + sub[(i + 1) % 6].y) / 2,  
 (sub[i].z + sub[(i + 1) % 6].z) / 2 - 0.2);  
 }  
  
 return result;  
}

Составляем тройки из координат, они будут являться полигонами:

std::vector<std::tuple<int, int, int>> get\_polygons() {  
 std::vector<std::tuple<int, int, int>> result;  
  
 // наконечник снизу  
 result.emplace\_back(0, 3, 4);  
 result.emplace\_back(0, 4, 5);  
 result.emplace\_back(0, 5, 6);  
 result.emplace\_back(0, 6, 7);  
 result.emplace\_back(0, 7, 3);  
  
 // бок с низу - стенка  
 result.emplace\_back(9, 3, 4);  
 result.emplace\_back(10, 4, 5);  
 result.emplace\_back(11, 5, 6);  
 result.emplace\_back(12, 6, 7);  
 result.emplace\_back(8, 7, 3);  
  
 // бок с верху - тенка  
 result.emplace\_back(3, 8, 9);  
 result.emplace\_back(4, 9, 10);  
 result.emplace\_back(5, 10, 11);  
 result.emplace\_back(6, 11, 12);  
 result.emplace\_back(7, 12, 8);  
  
 // основание нижнего многоугольника  
 result.emplace\_back(1, 19, 20);  
 result.emplace\_back(1, 20, 21);  
 result.emplace\_back(1, 21, 22);  
 result.emplace\_back(1, 22, 23);  
 result.emplace\_back(1, 23, 24);  
 result.emplace\_back(1, 24, 19);  
  
 // бока нижнего многоугольника  
 result.emplace\_back(14, 19, 20);  
 result.emplace\_back(15, 20, 21);  
 result.emplace\_back(16, 21, 22);  
 result.emplace\_back(17, 22, 23);  
 result.emplace\_back(18, 23, 24);  
 result.emplace\_back(13, 24, 19);  
  
 // бока верхнего многоугольника  
 result.emplace\_back(14, 15, 20);  
 result.emplace\_back(15, 16, 21);  
 result.emplace\_back(16, 17, 22);  
 result.emplace\_back(17, 18, 23);  
 result.emplace\_back(18, 13, 24);  
 result.emplace\_back(13, 14, 19);  
  
 // верхний многоугольник  
 result.emplace\_back(13, 14, 2);  
 result.emplace\_back(14, 15, 2);  
 result.emplace\_back(15, 16, 2);  
 result.emplace\_back(16, 17, 2);  
 result.emplace\_back(17, 18, 2);  
 result.emplace\_back(18, 13, 2);  
  
 return result;  
}

Формируем материалы, которые будут использоваться для окрашивания полигонов, они соответствуют количеству полигонов:

std::vector<BaseShader \*> get\_materials(int alpha) {  
 std::vector<BaseShader \*> result;  
  
 for (int i = 0; i < 33; i++) {  
 result.emplace\_back(new ColorShader({100 + i \* 10, 0, 60, alpha}));  
 }  
  
 Matrix rot\_z\_matrix = Matrix::RotationZ(60 \* M\_PI / 180);  
 std::vector<Vector> sub; // 13..18 - верхний многоугольник  
 sub.emplace\_back(0.7, 0, 0.7);  
 for (int i = 1; i < 6; i++) {  
 Vector temp = sub[i - 1] \* rot\_z\_matrix;  
 sub.emplace\_back(temp.x, temp.y, temp.z);  
 }  
  
 std::vector<Vector> v3;  
 v3.emplace\_back(sub.at(0));  
 v3.emplace\_back(sub.at(1));  
 v3.emplace\_back(0, 0, 0.7);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
 v3[0] = sub.at(1);  
 v3[1] = sub.at(2);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
 v3[0] = sub.at(2);  
 v3[1] = sub.at(3);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
 v3[0] = sub.at(3);  
 v3[1] = sub.at(4);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
 v3[0] = sub.at(4);  
 v3[1] = sub.at(5);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
 v3[0] = sub.at(5);  
 v3[1] = sub.at(0);  
 result.emplace\_back(new DottedShader(v3.at(0), v3.at(1), v3.at(2), alpha));  
  
 return result;  
}

Алгоритм художника

Для начала создадим функцию сравнения удалённости отрезков AB и CD. Основана она на решении уравнения:



Если 𝑧𝐴𝐵 < 𝑧𝐶𝐷, то отрезок AB перекрывает CD с точки зрения наблюдателя, т.е. AB расположен ближе, чем CD. Функция cmp\_sides():

// Возвращает 1, если AB ближе CD.  
// Возвращает -1, еслир AB дальше CD.  
// Возвращает 0, если отрезки не пересекаются.  
int cmp\_sides(Vector A, Vector B, Vector C, Vector D) {  
 auto inters\_res = intersect\_points\_2d(  
 {A.x, A.y, 0, 0},  
 {B.x, B.y, 0, 0},  
 {C.x, C.y, 0, 0},  
 {D.x, D.y, 0, 0}  
 );  
  
 // Если точки совпадают, нужно вернуть 0, иначе всё ломается (очень плохо ломается)  
 if (std::abs(A.x - C.x) < EPS && std::abs(A.y - C.y) < EPS ||  
 std::abs(A.x - D.x) < EPS && std::abs(A.y - D.y) < EPS ||  
 std::abs(B.x - C.x) < EPS && std::abs(B.y - C.y) < EPS ||  
 std::abs(B.x - D.x) < EPS && std::abs(B.y - D.y) < EPS)  
 return 0;  
  
 if (!std::get<0>(inters\_res)) return 0;  
  
 auto O = std::get<1>(inters\_res);  
 float zAB, zCD;  
  
 if (std::abs(B.x - A.x) > std::abs(B.y - A.y)) {  
 zAB = (O.x - A.x) \* (B.z - A.z) / (B.x - A.x) + A.z;  
 } else {  
 zAB = (O.y - A.y) \* (B.z - A.z) / (B.y - A.y) + A.z;  
 }  
  
 if (std::abs(D.x - C.x) > std::abs(D.y - C.y)) {  
 zCD = (O.x - C.x) \* (D.z - C.z) / (D.x - C.x) + C.z;  
 } else {  
 zCD = (O.y - C.y) \* (D.z - C.z) / (D.y - C.y) + C.z;  
 }  
  
 if (zAB < zCD) {  
 return 1;  
 } else {  
 return -1;  
 }  
}

Здесь используется вспомогательная функция, которая определяет пересечение проекций отрезков. Функция intersect\_points\_2d():

std::tuple<bool, Vector> intersect\_points\_2d(Vector A, Vector B, Vector C, Vector D) {  
 float dABx = std::abs(B.x - A.x);  
 float dCDx = std::abs(C.x - D.x);  
 float dABy = std::abs(B.y - A.y);  
 float dCDy = std::abs(C.y - D.y);  
 Vector O = {};  
  
 if (dABx < EPS && dCDy < EPS) {  
 O.x = A.x;  
 O.y = C.y;  
  
 return {true, O};  
 } else if (dABy < EPS && dCDx < EPS) {  
 O.x = C.x;  
 O.y = A.y;  
  
 return {true, O};  
 }  
 if (dABx < EPS && dCDx < EPS || dABy < EPS && dCDy < EPS) {  
 // Прямые параллельны  
 return {false, O};  
 } else if (dABx < EPS && dABy < EPS || dCDx < EPS && dCDy < EPS) {  
 // Передана точка  
 return {false, O};  
 } else if (dABx < EPS) {  
 // AB параллельна абсциссе  
 O.x = A.x;  
 O.y = (O.x - C.x) \* (D.y - C.y) / (D.x - C.x) + C.y;  
  
 return {  
 O.x < fmax(C.x, D.x) && O.x > fmin(C.x, D.x) &&  
 O.y < fmax(A.y, B.y) && O.y > fmin(A.y, B.y) &&  
 O.y < fmax(C.y, D.y) && O.y > fmin(C.y, D.y), O};  
 } else if (dABy < EPS) {  
 // AB параллельна ординате  
 O.y = A.y;  
 O.x = (O.y - C.y) \* (D.x - C.x) / (D.y - C.y) + C.x;  
  
 return {O.x < fmax(A.x, B.x) && O.x > fmin(A.x, B.x) &&  
 O.x < fmax(C.x, D.x) && O.x > fmin(C.x, D.x) &&  
 O.y < fmax(C.y, D.y) && O.y > fmin(C.y, D.y), O};  
 } else if (dCDy < EPS) {  
 O.y = C.y;  
 O.x = (O.y - A.y) \* (B.x - A.x) / (B.y - A.y) + A.x;  
  
 return {O.x < fmax(A.x, B.x) && O.x > fmin(A.x, B.x) &&  
 O.x < fmax(C.x, D.x) && O.x > fmin(C.x, D.x) &&  
 O.y < fmax(A.y, B.y) && O.y > fmin(A.y, B.y), O};  
 } else if (dCDx < EPS) {  
 O.x = C.x;  
 O.y = (O.x - A.x) \* (B.y - A.y) / (B.x - A.x) + A.y;  
  
 return {O.x < fmax(A.x, B.x) && O.x > fmin(A.x, B.x) &&  
 O.y < fmax(A.y, B.y) && O.y > fmin(A.y, B.y) &&  
 O.y < fmax(C.y, D.y) && O.y > fmin(C.y, D.y), O};  
 } else {  
 O.y = ((D.y - C.y) \* (A.x - A.y \* (B.x - A.x) / (B.y - A.y) - C.x) / (D.x - C.x) + C.y) /  
 (1 - (D.y - C.y) \* (B.x - A.x) / ((D.x - C.x) \* (B.y - A.y)));  
 O.x = (O.y - C.y) \* (D.x - C.x) / (D.y - C.y) + C.x;  
  
 return {O.x < fmax(A.x, B.x) && O.x > fmin(A.x, B.x) &&  
 O.x < fmax(C.x, D.x) && O.x > fmin(C.x, D.x) &&  
 O.y < fmax(A.y, B.y) && O.y > fmin(A.y, B.y) &&  
 O.y < fmax(C.y, D.y) && O.y > fmin(C.y, D.y), O};  
 }  
}

Ещё одна вспомогательная функция, которая проверяет принадлежность точки к плоскости:

bool is\_point\_inside\_polygon(Vector O, std::vector<Vector> polygon) {  
 if (std::abs(O.x - polygon[0].x) < EPS && std::abs(O.x - polygon[0].x) < EPS ||  
 std::abs(O.x - polygon[1].x) < EPS && std::abs(O.x - polygon[1].x) < EPS ||  
 std::abs(O.x - polygon[2].x) < EPS && std::abs(O.x - polygon[2].x) < EPS)  
 return true;  
  
 int neg\_count = 0, pos\_count = 0;  
 std::vector<Vector> vectors;  
 for (int i = 0; i < polygon.size(); i++) {  
 vectors.push\_back(  
 {  
 polygon[i].x - O.x,  
 polygon[i].y - O.y,  
 polygon[i].z - O.z  
 });  
 }  
  
 for (int i = 0; i < polygon.size(); i++) {  
 Vector a = vectors[i], b = vectors[(i + 1) % polygon.size()];  
 float cz = a.x \* b.y - b.x \* a.y;  
 if (cz < 0) {  
 neg\_count++;  
 } else {  
 pos\_count++;  
 }  
 }  
  
 return neg\_count == polygon.size() || pos\_count == polygon.size();  
}

Основная функция определения удалённости полигонов, она рассматривает 3 случая: пересечение сторон, плоскость находится внутри плоскости, стороны не пересекаются. Первый случай, проверяем пары сторон функцией cmp\_sides, Второй случай, проверяем функцией is\_point\_insight\_polygon. Функция cmp\_triangles:

// Возвращает 1, если A ближе B  
// Возвращает -1, если B ближе A  
// Возвращает 0, если A и B не пересекаются.  
int cmp\_triangles(std::vector<Vector> polygonA, std::vector<Vector> polygonB) {  
 Vector Ea;  
 Vector Eb;  
 bool Eb\_first = true;  
 bool a\_in\_b = true;  
 bool b\_in\_a = true;  
  
 for (int i = 0; i < polygonA.size(); i++) {  
 Ea.x += polygonA[i].x / polygonA.size();  
 Ea.y += polygonA[i].y / polygonA.size();  
 Ea.z += polygonA[i].z / polygonA.size();  
 a\_in\_b &= is\_point\_inside\_polygon(polygonA[i], polygonB);  
  
 for (int j = 0; j < polygonB.size(); j++) {  
 auto compare\_sides = cmp\_sides(  
 polygonA[i], polygonA[(i + 1) % polygonA.size()],  
 polygonB[j], polygonB[(j + 1) % polygonB.size()]  
 );  
  
 if (compare\_sides) return compare\_sides;  
  
 if (!Eb\_first) continue;  
  
 Eb.x += polygonB[j].x / polygonB.size();  
 Eb.y += polygonB[j].y / polygonB.size();  
 Eb.z += polygonB[j].z / polygonB.size();  
  
 b\_in\_a &= is\_point\_inside\_polygon(polygonB[j], polygonA);  
 }  
  
 Eb\_first = false;  
 }  
  
 bool swapped = false;  
  
 if (!a\_in\_b && !b\_in\_a) return 0;  
  
 // Делаем так, чтобы b был внутри a  
 if (a\_in\_b) {  
 std::swap(Ea, Eb);  
 std::swap(polygonA, polygonB);  
 swapped = true;  
 }  
  
 auto P0 = polygonA[0];  
 auto P1 = polygonA[1];  
 auto P2 = polygonA[2];  
  
 float z = ((Eb.y - P0.y) \* ((P1.x - P0.x) \* (P2.z - P0.z) - (P2.x - P0.x) \* (P1.z - P0.z)) -  
 (Eb.x - P0.x) \* ((P1.y - P0.y) \* (P2.z - P0.z) - (P2.y - P0.y) \* (P1.z - P0.z))) /  
 ((P1.x - P0.x) \* (P2.y - P0.y) - (P2.x - P0.x) \* (P1.y - P0.y)) + P0.z;  
  
 if (Eb.z < z) {  
 if (swapped) {  
 return 1;  
 } else {  
 return -1;  
 }  
 } else {  
 if (swapped) {  
 return -1;  
 } else {  
 return 1;  
 }  
 }  
}

Алгоритм художника. Первым делом формируем множество полигонов и нумеруем их. Далее заполняем матрицу H с информацией о перекрываемых гранях, по ней определяем удалённость граней, потом начнём отрисовку с неё и затем по приближению. После отрисовки, мы должны исключить из рассмотрения столбец и строку, относящиеся к этой грани.

Логика алгоритма художника:

std::vector<std::vector<Vector>> polygons\_vals;  
std::vector<int> polygons\_indices;  
for (int i = 0; i < polygons.size(); i++) {  
 // Формируем список полигонов  
 polygons\_vals.push\_back(  
 **{** B[std::get<0>(polygons[i])],  
 B[std::get<1>(polygons[i])],  
 B[std::get<2>(polygons[i])]  
 **}**);  
 polygons\_indices.push\_back(i);  
}  
  
auto H = std::vector<std::vector<int>>(polygons.size(), std::vector<int>(polygons.size(), 0));  
for (int i = 0; i < polygons.size(); i++) {  
 for (int j = 0; j < i; j++) {  
 auto res = cmp\_triangles(polygons\_vals[i], polygons\_vals[j]);  
 H[i][j] = res;  
 H[j][i] = -res;  
 }  
}  
  
while (H.size() != 0) {  
 int index\_to\_del = 0;  
 int min\_ones = H.size();  
 for (int i = 0; i < H.size(); i++) {  
 int ones = 0;  
  
 for (int j = 0; j < H[i].size(); j++) {  
 if (H[i][j] == 1) {  
 ones++;  
 }  
 }  
  
 if (ones < min\_ones) {  
 min\_ones = ones;  
 index\_to\_del = i;  
 }  
 }  
  
 frame.Triangle(  
 polygons\_vals[index\_to\_del][0],  
 polygons\_vals[index\_to\_del][1],  
 polygons\_vals[index\_to\_del][2],  
 materials[polygons\_indices[index\_to\_del]]);  
  
 H.erase(H.begin() + index\_to\_del);  
 for (int i = 0; i < H.size(); i++) {  
 H[i].erase(H[i].begin() + index\_to\_del);  
 }  
  
 polygons\_vals.erase(polygons\_vals.begin() + index\_to\_del);  
 polygons\_indices.erase(polygons\_indices.begin() + index\_to\_del);  
}

**Вывод:** в ходе работы получены навыки использования аффинных преобразований в пространстве и создания графического приложения с использованием GDI в среде Visual Studio для визуализации простейших трѐхмерных объектов.