# Документация 3D движка

## Версия 1.0

---

## 1. Введение

В данном документе описывается математический аппарат и структура 3D графического движка на основе трансформации вершин и растеризации треугольников. Движок включает модуль линейной алгебры (Math 3D), систему загрузки сеток (Mesh) и пайплайн обработки вершин с поддержкой рендеринга через OpenGL.

---

## 2. Модуль математических примитивов (Math 3D)

Модуль Math 3D реализует базовые структуры и операции линейной алгебры, необходимые для работы графического конвейера.

### 2.1. Структуры данных

#### 2.1.1. Вектор (Vec3)

Структура, представляющая трехмерный вектор с тремя компонентами типа float.

```
struct Vec3 {
    float x, y, z;
    Vec3(float _x = 0, float _y = 0, float _z = 0) : x(_x), y(_y), z(_z) {}
    Vec3 operator+(const Vec3& v) const { return Vec3(x + v.x, y + v.y, z +
v.z); }
    Vec3 operator-(const Vec3& v) const { return Vec3(x - v.x, y - v.y, z -
v.z); }
    Vec3 operator*(float f) const { return Vec3(x * f, y * f, z * f); }
    Vec3 Normalize() const;
};
```

Перегруженные операторы позволяют записывать формулы в стандартном векторном виде:

$$\vec{a} + \vec{b}, \quad \vec{a} - \vec{b}, \quad k \cdot \vec{a}.$$

### 2.1.2. Матрица 4x4 (Mat4)

Матрица трансформации для однородных координат, используемая для аффинных преобразований в трехмерном пространстве.

```
struct Mat4 {
    float m[4][4];  // Формат row-major: m[row][col]
    Mat4();
    static Mat4 Identity();
    static Mat4 RotateX(float angleRadians);
    static Mat4 RotateY(float angleRadians);
    static Mat4 RotateZ(float angleRadians);
    static Mat4 Translate(float x, float y, float z);
    static Mat4 Projection(float fov, float aspectRatio, float zNear, float
zFar);
    Mat4 operator*(const Mat4& other) const;
};
```

Выбран формат хранения **row-major** (строка-столбец), где `m[row][col]`. При индексации элемент трансляции расположен в последнем столбце: `m[i][3]`, что соответствует умножению матрицы на вектор-столбец справа.

## 2.2. Операции линейной алгебры

### 2.2.1. Скалярное произведение (Dot Product)

**Код:**

```
float DotProduct(const Vec3& a, const Vec3& b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

**Формула:**

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z \tag{1}$$

Используется для расчета угла между векторами (освещение, отсечение невидимых граней).

### 2.2.2. Векторное произведение (Cross Product)

**Код:**

```
Vec3 CrossProduct(const Vec3& a, const Vec3& b) {
    return Vec3(
        a.y * b.z - a.z * b.y,
        a.z * b.x - a.x * b.z,
        a.x * b.y - a.y * b.x
    );
}
```

**Формула:**

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \tag{2}$$

Результат — вектор, перпендикулярный обоим исходным. Критически важен для вычисления нормалей поверхностей.

### 2.2.3. Нормализация вектора

**Код:**

```
Vec3 Normalize() const {
    float length = std::sqrt(x*x + y*y + z*z);
    return Vec3(x / length, y / length, z / length);
}
```

**Формула:**

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||} = \frac{\vec{v}}{\sqrt{x^2 + y^2 + z^2}} \tag{3}$$

Приводит векторы направлений (свет, взгляд, нормали) к единичной длине.

### 2.2.4. Матрицы вращения

Вращение в трехмерном пространстве описывается матрицами 4 × 4, где при вращении вокруг одной оси координаты этой оси остаются неизменными.

#### 2.2.4.1. Вращение вокруг оси X (RotateX)

**Код:**

```
Mat4 Mat4::RotateX(float theta) {
    Mat4 mat = Mat4::Identity();
```

```
    mat.m[1][1] = cosf(theta);  mat.m[1][2] = -sinf(theta);
    mat.m[2][1] = sinf(theta);  mat.m[2][2] = cosf(theta);
    return mat;
}
```

**Матрица вращения вокруг оси X на угол θ:**

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Первая строка и столбец остаются единичными, так как x-координата не меняется. Блок 2×2 в центре представляет двумерный поворот в плоскости YZ.

### 2.2.4.2. Вращение вокруг оси Y (RotateY)

**Код:**

```
Mat4 Mat4::RotateY(float theta) {
    Mat4 mat = Mat4::Identity();
    mat.m[0][0] = cosf(theta);  mat.m[0][2] = sinf(theta);
    mat.m[2][0] = -sinf(theta); mat.m[2][2] = cosf(theta);
    return mat;
}
```

**Матрица вращения вокруг оси Y на угол θ:**

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Вторая строка и столбец фиксированы (y-координата не меняется). Знаки синусов поменялись местами из-за правила буравчика.

### 2.2.4.3. Вращение вокруг оси Z (RotateZ)

**Код:**

```
Mat4 Mat4::RotateZ(float theta) {
    Mat4 mat = Mat4::Identity();
    mat.m[0][0] = cosf(theta);  mat.m[0][1] = -sinf(theta);
    mat.m[1][0] = sinf(theta);  mat.m[1][1] = cosf(theta);
```

```
    return mat;
}
```

**Матрица вращения вокруг оси Z на угол θ:**

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Третья строка и столбец фиксированы. Верхний левый блок 2×2 описывает стандартный поворот в плоскости XY.

## 2.2.5. Матрица переноса (Translation Matrix)

**Код:**

```
Mat4 Mat4::Translate(float x, float y, float z) {
    Mat4 mat = Mat4::Identity();
    mat.m[0][3] = x;
    mat.m[1][3] = y;
    mat.m[2][3] = z;
    return mat;
}
```

**Матрица сдвига на вектор (tx, ty, tz):**

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

При умножении на вектор $(x, y, z, 1)^T$:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} \tag{8}$$

Однородные координаты (четвертая компонента w = 1) позволяют записывать аффинные преобразования через матричное умножение.

## 2.2.6. Матрица проекции (Projection Matrix)

**Код:**

```
Mat4 Mat4::Projection(float fov, float aspectRatio, float zNear, float zFar) {
    Mat4 mat;
    float f = 1.0f / tanf(fov * 0.5f);
    float q = zFar / (zFar - zNear);
    mat.m[0][0] = aspectRatio * f;
    mat.m[1][1] = f;
    mat.m[2][2] = q;
    mat.m[2][3] = -zNear * q;
    mat.m[3][2] = 1.0f;
    return mat;
}
```

**Матрица перспективной проекции:**

$$P = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}} & -\frac{z_{\text{far}} \cdot z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{9}$$

$$\text{где } f = \cot(\text{FOV}/2)$$

**Назначение элементов:**

- **Элементы [0][0] и [1][1]:** Масштабирование X и Y с учетом угла обзора и соотношения сторон экрана.
- **Элемент [3][2] = 1:** Ключевой элемент для перспективы. При умножении вектор z попадает в компоненту w результата.
- **Третья строка:** Преобразует z из диапазона [zNear, zFar] в [0, 1] для работы Z-буфера.

## 2.2.7. Умножение матрицы на вектор

**Код:**

```
Vec3 MultiplyMatrixVector(const Vec3& i, const Mat4& m) {
    Vec3 v;
    float x = i.x * m.m[0][0] + i.y * m.m[0][1] + i.z * m.m[0][2] + m.m[0][3];
    float y = i.x * m.m[1][0] + i.y * m.m[1][1] + i.z * m.m[1][2] + m.m[1][3];
    float z = i.x * m.m[2][0] + i.y * m.m[2][1] + i.z * m.m[2][2] + m.m[2][3];
    float w = i.x * m.m[3][0] + i.y * m.m[3][1] + i.z * m.m[3][2] + m.m[3][3];
    if (w != 0.0f) {
        v.x = x / w;
        v.y = y / w;
        v.z = z / w;
```

```
    }
    return v;
}
```

**Формула:**

$$\overrightarrow{v'} = M \times \overrightarrow{v} \tag{10}$$

Вектор (x, y, z) неявно дополняется до (x, y, z, 1), умножается на матрицу 4 × 4, и результат нормализуется делением на компоненту w. Для аффинных преобразований w = 1, но при перспективной проекции w может отличаться от 1, что обеспечивает эффект перспективы.

### 2.2.8. Перемножение матриц

**Код:**

```
Mat4 Mat4::operator*(const Mat4& other) const {
    Mat4 res;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            res.m[i][j] = 0;
            for (int k = 0; k < 4; k++) {
                res.m[i][j] += m[i][k] * other.m[k][j];
            }
        }
    }
    return res;
}
```

**Формула:**

$$(AB)_{ij} = \sum_{k=0}^{3} A_{ik} B_{kj} \tag{11}$$

Используется наивный алгоритм (O(n^3)) для простоты, так как матрицы имеют малый размер 4×4.

---

## 3. Система загрузки сеток (Mesh)

Модуль Mesh отвечает за загрузку трехмерных моделей из файлов формата OBJ и их триангуляцию.

## 3.1. Структуры данных

### 3.1.1. Сетка (Mesh)

```
struct Mesh {
    std::vector<Vec3> vertices;      // Массив вершин
    struct Face {
        int v[3];                    // Индексы трех вершин треугольника
    };
    std::vector<Face> faces;         // Массив треугольных граней
    static Mesh LoadFromObj(const std::string& filename);
};
```

## 3.2. Загрузка из OBJ

**Код (ключевые части):**

```
Mesh Mesh::LoadFromObj(const std::string& filename) {
    Mesh mesh;
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        if (line.empty() || line[0] == '#') continue;

        std::stringstream ss(line);
        std::string prefix;
        ss >> prefix;

        if (prefix == "v") {
            Vec3 v;
            ss >> v.x >> v.y >> v.z;
            mesh.vertices.push_back(v);
        }
        else if (prefix == "f") {
            // Парсинг индексов вершин...
            // Триангуляция многоугольников...
        }
    }
    return mesh;
}
```

**Процесс загрузки:**

1. Для каждой строки вида `v x y z` добавляется вершина в вектор vertices.
2. Для строк вида `f idx1 idx2 idx3 ...` (могут содержать текстурные координаты и нормали) извлекаются только индексы вершин.
3. Если полигон содержит более 3 вершин, выполняется **триангуляция** (разбиение на треугольники).

### 3.2.1. Триангуляция

Многоугольник с индексами вершин

$$(i_0, i_1, i_2, i_3, \ldots, i_n)$$

преобразуется в набор треугольников:

$$(i_0, i_1, i_2), (i_0, i_2, i_3), \ldots, (i_0, i_{n-1}, i_n) \tag{12}$$

Первая вершина соединяется со всеми последующими парами вершин, образуя "вееровидную" структуру.

---

## 4. Трансформация вершин (World Transformation)

Преобразование вершин из локальных координат модели в мировые координаты.

### 4.1. Матрица мира

В основном цикле рендеринга строится матрица мира из отдельных трансформаций:

```
Mat4 matRotY = Mat4::RotateY(rotY);
Mat4 matRotX = Mat4::RotateX(rotX);
Mat4 matTrans = Mat4::Translate(0.0f, 0.0f, cameraZoom);

Mat4 matWorld = Mat4::Identity();
matWorld = matRotY * matWorld;
matWorld = matRotX * matWorld;
matWorld = matTrans * matWorld;
```

**Итоговая матрица:**

$$M_{\text{world}} = T \cdot R_x \cdot R_y \tag{13}$$

Порядок умножения важен: вершина сначала вращается вокруг оси Y, затем вокруг оси X, затем сдвигается.

### 4.2. Применение матрицы

```
Vec3 v0 = MultiplyMatrixVector(myMesh.vertices[face.v[0]], matWorld);
```

**Формула:**

$$\vec{v}_{\text{world}} = M_{\text{world}} \times \vec{v}_{\text{local}} \tag{14}$$

---

# 5. Расчет нормали и отсечение невидимых граней (Backface Culling)

## 5.1. Вычисление нормали поверхности

Для треугольника с вершинами $v_0$, $v_1$, $v_2$:

```
Vec3 edge1 = v1 - v0;
Vec3 edge2 = v2 - v0;
Vec3 normal = CrossProduct(edge1, edge2).Normalize();
```

**Формула:**

$$\vec{N} = \frac{(\vec{v_1} - \vec{v_0}) \times (\vec{v_2} - \vec{v_0})}{||(\vec{v_1} - \vec{v_0}) \times (\vec{v_2} - \vec{v_0})||} \tag{15}$$

## 5.2. Отсечение задней грани

Камера расположена в начале координат O(0, 0, 0). Вектор от поверхности к камере:

```
Vec3 viewDir = (v0 * -1.0f).Normalize();
```

**Формула:**

$$\vec{V} = \text{Normalize}(-\vec{v_0}) \tag{16}$$

**Проверка видимости:**

```
if (DotProduct(normal, viewDir) > 0.0f) { /* грань видима */ }
```

**Формула:**

$$d = \vec{N} \cdot \vec{V} > 0 \tag{17}$$

Если d > 0, угол между нормалью и взглядом острый (< 90°), грань "смотрит" в сторону камеры и должна быть отрисована.

---

# 6. Освещение

Используется модель диффузного отражения Ламберта:

```
Vec3 lightDir = Vec3(0.5f, 1.0f, -1.0f).Normalize();
float dot = DotProduct(normal, lightDir);
float intensity = std::max(0.0f, dot);
intensity = 0.2f + (0.8f * intensity);
```

**Формула интенсивности:**

$$I_{\text{diffuse}} = \max(0, \vec{N} \cdot \vec{L}) \tag{18}$$

**Формула цвета пикселя:**

$$\text{Color} = \text{Color}_{\text{base}} \cdot (I_{\text{ambient}} + I_{\text{diffuse}} \cdot k_{\text{diffuse}}) \tag{19}$$

В коде:

$$I_{\text{ambient}} = 0.2, k_{\text{diffuse}} = 0.8$$

---

# 7. Перспективная проекция

## 7.1. Применение матрицы проекции

```
Mat4 matProj = Mat4::Projection(1.57f, aspect, 0.1f, 100.0f);
Vec3 p0 = MultiplyMatrixVector(v0, matProj);
```

**Формула:**

$$\vec{v}_{\text{clip}} = P \times \vec{v}_{\text{world}} \tag{20}$$

Функция `MultiplyMatrixVector` выполняет умножение на матрицу проекции и перспективное деление на w.

## 7.2. Перспективное деление (Perspective Divide)

После умножения на матрицу проекции w ≠ 1. Нормализованные координаты устройства (NDC) вычисляются делением:

```
if (w != 0.0f) {
    v.x = x / w;
    v.y = y / w;
    v.z = z / w;
}
```

**Формула:**

$$x_{\text{ndc}} = \frac{x_{\text{clip}}}{w_{\text{clip}}}, \quad y_{\text{ndc}} = \frac{y_{\text{clip}}}{w_{\text{clip}}}, \quad z_{\text{ndc}} = \frac{z_{\text{clip}}}{w_{\text{clip}}} \tag{21}$$

Это "сплющивает" усеченную пирамиду (frustum) видимости в куб [-1, 1]³. Значение w, полученное из элемента [3][2] матрицы проекции, равно z-координате мировых координат, что обеспечивает эффект перспективы (далекие объекты становятся меньше).

---

# 8. Преобразование в экранные координаты (Viewport Transform)

## 8.1. Преобразование из NDC в пиксели

```
auto ToScreen = [&](Vec3& p) {
    p.x = (p.x + 1.0f) * 0.5f * WINDOW_WIDTH;
    p.y = (p.y + 1.0f) * 0.5f * WINDOW_HEIGHT;
};
```

**Формулы линейной интерполяции:**

$$x_{\text{screen}} = (x_{\text{ndc}} + 1) \cdot \frac{\text{WIDTH}}{2} \tag{22}$$

$$y_{\text{screen}} = (y_{\text{ndc}} + 1) \cdot \frac{\text{HEIGHT}}{2} \tag{23}$$

Преобразует координаты из диапазона [-1, 1] в пиксельные координаты [0, WIDTH] × [0, HEIGHT].

## 8.2. Z-буфер

Z-координата сохраняется для целей глубинного тестирования:

```
float z0 = p0.z;
float z1 = p1.z;
float z2 = p2.z;
renderer.DrawTriangle(
    (int)p0.x, (int)p0.y, z0,
    (int)p1.x, (int)p1.y, z1,
    (int)p2.x, (int)p2.y, z2,
    color
);
```

Z-значения передаются в растеризатор для определения, какие пиксели ближе к камере.

---

## 9. Полный пайплайн обработки вершины

1. **Трансформация в мировые координаты:**

$$\vec{v}_{\text{world}} = M_{\text{world}} \times \vec{v}_{\text{local}}$$

2. **Расчет нормали поверхности:**

$$\vec{N} = \frac{(\vec{v_1} - \vec{v_0}) \times (\vec{v_2} - \vec{v_0})}{||(\vec{v_1} - \vec{v_0}) \times (\vec{v_2} - \vec{v_0})||}$$

3. **Отсечение задней грани (Backface Culling):**

$$\text{if } \vec{N} \cdot (-\vec{v_0}) > 0 \text{ then draw}$$

4. **Расчет освещения:**

$$I = 0.2 + 0.8 \cdot \max(0, \vec{N} \cdot \vec{L})$$

5. **Перспективная проекция:**

$$\vec{v}_{\text{clip}} = P \times \vec{v}_{\text{world}}$$

6. **Перспективное деление:**

$$\vec{v}_{\text{ndc}} = \frac{\vec{v}_{\text{clip}}}{w}$$

7. **Преобразование в экранные координаты:**

$$\vec{v}_{\text{screen}} = (\text{ndc} + 1) \cdot 0.5 \cdot \text{resolution}$$

8. **Растеризация и вывод треугольника с глубиной.**

---

# 10. Параметры системы

## 10.1. Константы экрана

- **WINDOW_WIDTH** = 1200
- **WINDOW_HEIGHT** = 800
- **Aspect Ratio** = HEIGHT / WIDTH = 0.667

## 10.2. Параметры проекции

- **FOV** = 1.57 радиан (≈ 90°)
- **zNear** = 0.1
- **zFar** = 100.0

## 10.3. Параметры освещения

- **Направление света:** (0.5, 1.0, -1.0) (нормализованный вектор)
- **Коэффициент окружающего света (ambient):** 0.2
- **Коэффициент диффузного света (diffuse):** 0.8

---

# 11. Итоговая математическая последовательность

Финальное преобразование вершины может быть записано как единая композиция:

$$\overrightarrow{v}_{\text{screen}} = \text{Viewport} \left( \frac{\text{Projection} \times M_{\text{world}} \times \overrightarrow{v}_{\text{local}}}{w} \right) \tag{24}$$

где Projection — матрица перспективной проекции, M_world — матрица трансформации мира, Viewport — преобразование в экранные координаты, а w получается из четвертой компоненты вектора после применения матриц трансформации.