Документация графического конвейера (Software Renderer)

Максим Лобанов

3 декабря 2025 г.

1 Введение

В данном документе описывается математический аппарат, используемый в функции `main` программного рендерера, а также реализация базовой линейной алгебры.

2 Модуль математических примитивов (Math 3D)

Для работы графического движка был реализован собственный минималистичный модуль линейной алгебры.

2.1 Структуры данных

2.1.1 Вектор (Vec3)

Используется структура из трех компонент типа `float`.

```
struct Vec3 {
    float x, y, z;

    // Конструктор
    Vec3(float _x = 0, float _y = 0, float _z = 0) : x(_x), y(_y), z(_z
       ) {}

    // Операторы сложения и вычитания
    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); }

    // Умножение на число (скаляр)
    Vec3 operator*(float f) const { return Vec3(x * f, y * f, z * f); }
...
}
```

Реализация перегружает стандартные операторы для удобства записи формул в коде в векторном виде: $\vec{a} + \vec{b}$, $\vec{a} - \vec{b}$, $k \cdot \vec{a}$.

2.1.2 Матрица 4x4 (Mat4)

Матрица трансформации для однородных координат.

```
struct Mat4 {
    float m[4][4]; // Строка, Столбец
    // ... конструкторы и методы
};
```

Выбран формат хранения **row-major** (строка-столбец), где `m[row][col]`. Это отличается от стандартного OpenGL (column-major), но более интуитивно понятно при ручном написании кода и индексации (например, T_x это `m[0][3]`, а не `m[3][0]`).

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 расчетов освещения (угол падения света) и отсечения 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 (определитель матрицы с базисными векторами):

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

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:

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

Необходима для приведения векторов направлений (света, взгляда, нормалей) к единичной длине.

2.2.4 Матрицы вращения (Rotation Matrices)

Вращение в трехмерном пространстве описывается с помощью матриц 4×4. Структура каждой матрицы обусловлена тем, что при вращении вокруг одной из координатных осей координаты этой оси остаются неизменными.

2.2.5 Вращение вокруг оси X (RotateX)

Исходный 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:

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

Почему так:

- Первая строка и первый столбец — это вектор $(1, 0, 0, 0)$, так как координата x при вращении вокруг оси X не меняется ($x' = x$).
- Блок 2×2 в центре (строки/столбцы 1 и 2, или y, z) представляет собой стандартную двумерную матрицу поворота в плоскости YZ:

$$\begin{pmatrix} y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix}$$

2.2.6 Вращение вокруг оси Y (RotateY)

Исходный 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:

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

Почему так:

- Вторая строка и второй столбец — $(0, 1, 0, 0)$, так как координата y не меняется ($y' = y$).
- Знаки синусов поменялись местами относительно R_x и R_z (минус стоит в элементе $(2, 0)$, а не $(0, 2)$). Это связано с правилом буравчика и тем, что поворот в плоскости ZX (если смотреть с положительной оси Y) идет от оси Z к оси X.

2.2.7 Вращение вокруг оси Z (RotateZ)

Исходный 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:

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

Почему так:

- Третья строка и третий столбец фиксированы, так как $z' = z$.
- Верхний левый блок 2×2 описывает стандартный поворот в плоскости XY.

2.2.8 Матрица переноса (Translation Matrix)

Исходный код:

```
Mat4 mat = Mat4::Identity();
mat.m[0][3] = x; mat.m[1][3] = y; mat.m[2][3] = z;
```

Матрица сдвига на вектор (t_x, t_y, t_z):

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

Почему так: Матричное умножение работает как линейная комбинация столбцов. Если умножить эту матрицу на вектор $(x, y, z, 1)^T$, получим:

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

Именно для возможности записи сложения (сдвига) через умножение матриц и вводятся однородные координаты (четвертая компонента $w = 1$).

2.2.9 Матрица проекции (Projection Matrix)

Исходный код:

```
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;
```

Матрица перспективной проекции выполняет две задачи: учитывает угол обзора (FOV) и подготавливает координату z для перспективного деления.

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

Почему она выглядит именно так:

- **Масштабирование X и Y (элементы [0][0] и [1][1]):** Коэффициент $f = \cot(FOV/2)$ масштабирует координаты так, чтобы объекты, находящиеся под углом $FOV/2$, попадали на край экрана (в координату 1). Деление на $aspect$ компенсирует неквадратность экрана.
- **Элемент [3][2] = 1:** Это ключевой элемент для перспективы. При умножении на вектор $(x, y, z, 1)^T$ в компоненту w результата попадает исходное значение z ($w' = 1 \cdot z$). В дальнейшем, когда мы будем делить вектор на w', мы фактически разделим x и y на z, что сделает далекие объекты меньше.
- **Преобразование Z (третья строка):** Элементы q и $-z_{near} \cdot q$ нужны для того, чтобы перевести координату z из диапазона $[z_{near}, z_{far}]$ в диапазон $[0, 1]$ (или $[-1, 1]$ в OpenGL), что необходимо для работы Z-буфера (глубины).

2.2.10 Умножение Матрицы на Вектор

Код:

```
Vec3 MultiplyMatrixVector(const Vec3& i, const Mat4& m) {
    // ... умножение 4x4 на 4x1 ...
    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; ... }
}
```

Математически это умножение матрицы M на вектор-столбец v:

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

В реализации вектор (x, y, z) неявно дополняется до $(x, y, z, 1)$. Особенность реализации — встроенное **перспективное деление** на компоненту w. Это упрощает использование функции в пайплайне, сразу возвращая нормализованные координаты устройства (NDC).

2.2.11 Перемножение Матриц

Код:

```
Mat4 Mat4::operator*(const Mat4& other) const {
    // Тройной цикл перемножения: строка * столбец
}
```

Формула:

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

Используется наивный алгоритм $O(N^3)$ для простоты реализации, так как матриц немного и они размерности всего 4x4.

3 Матрицы трансформации (World Transformation)

Исходный код:

```
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;
```

3.1 Вращение вокруг осей (Rotation)

Вращение точки (x, y, z) в трехмерном пространстве.

Вращение вокруг оси Y: Код: `Mat4::RotateY(rotY)`
Матрица $R_y(\theta)$:

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{12}$$

Вращение вокруг оси X: Код: `Mat4::RotateX(rotX)`
Матрица $R_x(\phi)$:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{13}$$

3.2 Перенос (Translation)

Код: `Mat4::Translate(0.0f, 0.0f, cameraZoom)`

Матрица сдвига $T(t_x, t_y, t_z)$:

$$T(0,0,t_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14}$$

В реализации элементы смещения находятся в последнем столбце (`m[i][3]`), что соответствует умножению на вектор-столбец справа.

3.3 Итоговая матрица мира (Model Matrix)

В коде матрицы перемножаются последовательно. Порядок важен.

$$M_{world} = T \cdot R_x \cdot R_y \tag{15}$$

Это означает, что вершина сначала вращается вокруг своей оси Y, затем поворачивается вокруг оси X, и наконец сдвигается.

4 Перспективная проекция (Projection)

Исходный код:

```
float aspect = (float)WINDOW_HEIGHT / (float)WINDOW_WIDTH;
Mat4 matProj = Mat4::Projection(1.57f, aspect, 0.1f, 100.0f);
```

Матрица проекции превращает усеченную пирамиду видимости (frustum) в куб нормализованных координат (NDC). Параметры:

- $FOV = 1.57$ радиан ($\approx 90°$)
- $Aspect = H/W$
- $z_{near} = 0.1$, $z_{far} = 100.0$

Формула матрицы P (OpenGL-style):

$$P = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far}-z_{near}} & \frac{-z_{far}\cdot z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{16}$$

где $f = \cot(FOV/2)$. Примечание: Элемент $(3,2) = 1$ сохраняет исходную Z-координату в компоненту W результата для последующего деления.

5 Конвейер обработки вершины (Vertex Pipeline)

В цикле по граням (`face`) происходит обработка каждой вершины.

5.1 1. Трансформация в мировые координаты

Код:

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

Математически:

$$\vec{v}_{world} = M_{world} \times \vec{v}_{local} \tag{17}$$

Здесь исходная вершина (x, y, z) неявно дополняется до $(x, y, z, 1)$, умножается на матрицу 4×4, и результат возвращается как 3D вектор (деление на W здесь еще не меняет сути, так как для аффинных преобразований $W = 1$). Результат — координаты в мировом пространстве.

5.2 2. Расчет нормали и Backface Culling

Код:

```
Vec3 edge1 = v1 - v0;
Vec3 edge2 = v2 - v0;
Vec3 normal = CrossProduct(edge1, edge2).Normalize();
Vec3 viewDir = (v0 * -1.0f).Normalize();
if (DotProduct(normal, viewDir) > 0.0f) { ... }
```

Нормаль поверхности: В коде используется векторное произведение сторон треугольника. Стороны: $\vec{e}_1 = \vec{v}_1 - \vec{v}_0$ и $\vec{e}_2 = \vec{v}_2 - \vec{v}_0$.

$$\vec{N} = \text{Normalize}(\text{CrossProduct}(\vec{e}_1, \vec{e}_2)) \tag{18}$$

То есть:

$$\vec{N} = \frac{(\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)}{||(\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)||} \tag{19}$$

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

$$\vec{V} = \text{Normalize}(O - \vec{v}_0) = \text{Normalize}(-\vec{v}_0) \tag{20}$$

Отсечение граней (Culling): Проверяется знак скалярного произведения:

$$d = \vec{N} \cdot \vec{V} \tag{21}$$

В реализации используется условие $d > 0$. Это означает, что угол между нормалью и взглядом острый ($< 90°$), то есть грань "смотрит"в сторону камеры.

5.3 3. Освещение (Lighting)

Код:

```
float dot = DotProduct(normal, lightDir);
float intensity = std::max(0.0f, dot);
```

Модель Ламберта (диффузное отражение):

$$I_{diffuse} = \max(0, \vec{N} \cdot \vec{L}) \tag{22}$$

Где $\vec{L}$ — нормализованный вектор на источник света (`lightDir`). Цвет пикселя вычисляется как:

$$Color = Color_{base} \cdot (I_{ambient} + I_{diffuse} \cdot k_{diffuse}) \tag{23}$$

В коде это реализовано строкой `intensity = 0.1f + (0.9f * intensity);`.

6 Проекция в экранные координаты (Viewport)

Код:

```
Vec3 p0 = MultiplyMatrixVector(v0, matProj);
auto ToScreen = [&](Vec3& p) {
    p.x = (p.x + 1.0f) * 0.5f * WINDOW_WIDTH;
    p.y = (p.y + 1.0f) * 0.5f * WINDOW_HEIGHT;
};
```

6.1 Перспективное деление (Perspective Divide)

Функция `MultiplyMatrixVector` выполняет умножение на матрицу проекции и деление на W:

$$\vec{v}_{clip} = P \times \vec{v}_{world} \tag{24}$$

Нормализованные координаты устройства (NDC) получаются делением компонент на w_{clip}:

$$x_{ndc} = \frac{x_{clip}}{w_{clip}}, \quad y_{ndc} = \frac{y_{clip}}{w_{clip}}, \quad z_{ndc} = \frac{z_{clip}}{w_{clip}} \tag{25}$$

Это действие "сплющивает"усеченную пирамиду в куб $[-1, 1]^3$.

6.2 Преобразование в порт просмотра (Viewport Transform)

Перевод из NDC $[-1, 1]$ в пиксельные координаты $[0, W] \times [0, H]$. Формула линейной интерполяции:

$$x_{screen} = (x_{ndc} + 1) \cdot \frac{W}{2} \tag{26}$$

$$y_{screen} = (y_{ndc} + 1) \cdot \frac{H}{2} \tag{27}$$

Полученные координаты (x_{screen}, y_{screen}) передаются в растеризатор. Координата z отбрасывается (так как Z-буфер не реализован в явном виде для сортировки пикселей в данном примере).