Документация к Mezera Engine

Романов Иннокентий Александрович
НИЯУ МИФИ, Институт интеллектуальных кибернетических систем
группа Б24-527

20 апреля 2025 г.

Содержание

1 Этапы пайплайна рендеринга

Данная глава описывает полный путь инициализации Vulkan и построения рендер-пайплайна, начиная с создания `Instance` и заканчивая выводом первого треугольника на экран.

1.1 Создание Vulkan Instance

Первым шагом является создание объекта `VkInstance`, который представляет собой подключение к драйверу Vulkan и инициализирует среду.

Реализация: `core/Instance`

- Устанавливаются необходимые расширения (например, для поддержки окна через GLFW).
- Включается слой валидации (если сборка отладочная).
- Производится вызов `vkCreateInstance`.

1.2 Выбор физического устройства

Следующий шаг — выбор физического устройства (`VkPhysicalDevice`) с нужными характеристиками.

Реализация: `core/PhysicalDevice`

- Производится перечисление всех доступных GPU.
- Проверяется наличие нужных очередей и поддержка swapchain.

1.3 Создание логического устройства

Создаётся `VkDevice`, логическое представление выбранного физического устройства.

Реализация: `core/LogicalDevice`

- Указываются очереди (графическая, презентационная).
- Устанавливаются нужные device-extensions.

1.4 Создание окна

Для вывода требуется окно (создаётся через GLFW) и Vulkan-совместимая поверхность `VkSurfaceKHR`.

Реализация: `window/WindowInit`, `core/Surface`

1.5 Создание swapchain

Swapchain управляет изображениями, которые выводятся на экран. Он связывается с окном и поверхностью.

Реализация: `core/Swapchain`

- Настраивается формат изображения, количество буферов.
- Создаются образы и image views.

1.6 Создание render pass и framebuffer'ов

Рендер-проход описывает, как будут использоваться буферы (цвета, глубины и т.д.)
Реализация: `pipeline/RenderPass`, `pipeline/FrameBuffers`

1.7 Построение графического пайплайна

На этом этапе настраивается весь GPU-процесс рендеринга: шейдеры, входные данные, примитивы, буферы.
Реализация: `pipeline/PipelineBuilder`

- Загружаются SPIR-V шейдеры.
- Описываются вершинные атрибуты (`render/Vertex`).
- Устанавливаются push constants и uniform'ы.

1.8 Создание буферов и загрузка вершин

Создаются vertex-буферы и загружаются координаты треугольника.
Реализация: `renderer/Mesh`

1.9 Командные буферы

Всё рисование описывается в `CommandBuffers`, включая привязку пайплайна, буферов и вызов `vkCmdDraw`.
Реализация: `commands/CommandBuffers`

1.10 Синхронизация

Используются семафоры и фэнсы для корректной синхронизации CPU и GPU.
Реализация: `sync/SyncObjects`

1.11 Первый кадр: DrawFrame

Основная функция отрисовки кадра. Она берёт swapchain image, записывает команды и выводит их на экран.
Реализация: `renderer/DrawFrame`

2 Камера и матрицы трансформации (MVP)

В этом разделе рассматриваются все этапы формирования финального положения объектов в пространстве. Используется концепция **MVP**-матриц: Model, View, Projection.

Реализация камеры и матричных преобразований находится в файлах:

- `include/renderer/Camera.hpp`
- `src/renderer/Camera.cpp`

2.1 Model-матрица

Model-матрица задаёт локальное преобразование объекта в мировое пространство. Она определяет:

- положение объекта (трансляция),
- его ориентацию (вращение),
- масштаб (scale).

Пример модельной матрицы:

$$\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$$

где:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

А $\mathbf{R}$ — матрица поворота, зависящая от углов Эйлера или кватерниона.

2.2 View-матрица (матрица камеры)

Матрица вида перемещает и поворачивает сцену так, чтобы она была "сфокусирована"с точки зрения камеры. В коде создается с помощью функции lookAt() `lookAt()`:

$$\mathbf{V} = \texttt{lookAt}(\mathbf{eye}, \mathbf{center}, \mathbf{up})$$

Где:

- **eye** — позиция камеры,
- **center** — точка, на которую смотрит камера,
- **up** — вектор "вверх".

Формула матрицы вида:

$$\mathbf{V} = \begin{bmatrix} \mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & -\mathbf{x} \cdot \mathbf{eye} \\ \mathbf{y}_x & \mathbf{y}_y & \mathbf{y}_z & -\mathbf{y} \cdot \mathbf{eye} \\ \mathbf{z}_x & \mathbf{z}_y & \mathbf{z}_z & -\mathbf{z} \cdot \mathbf{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

где $\mathbf{z}$ — направление "назад"(нормализованный вектор $\mathbf{eye} - \mathbf{center}$), $\mathbf{x}$ и $\mathbf{y}$ — базис, полученный через векторное произведение.

2.3 Projection-матрица

Проекционная матрица проецирует 3D-координаты в 2D-пространство экрана.

Для перспективной проекции (используется в движке):

$$\mathbf{P} = \begin{bmatrix} \frac{1}{a \cdot \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{z_f + z_n}{z_f - z_n} & -\frac{2 z_f z_n}{z_f - z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Где:

- fov — угол обзора,
- a — аспектное соотношение (ширина / высота),
- z_n, z_f — ближняя и дальняя плоскости отсечения.

2.4 Сборка полной MVP-матрицы

Окончательное преобразование:

$$\mathbf{MVP} = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M}$$

Порядок умножения важен: сначала применяется Model, затем View, затем Projection.

2.5 Практическое применение

Камера реализована в классе `Camera` с возможностью передвижения, вращения и настройки проекции. Матрицы создаются с использованием библиотеки `glm`.

Методы:

- `getViewMatrix()`
- `getProjectionMatrix()`
- `getPosition()`, `lookAt()`, `move()`, `rotate()`

Трансформации объектов (не камеры) задаются через структуру `Transform`, содержащую:

- позицию,
- поворот (матрицы Эйлера),
- масштаб.

`Transform::getMatrix()` возвращает итоговую Model-матрицу.

3 Сцена, объекты и отрисовка

Основу визуализации в движке составляет система сцены, включающая в себя:

- коллекцию объектов (мешей);
- камеру;
- функциональность для обновления и отрисовки.

Все объекты сцены реализованы через классы `Scene` и `Mesh`.

3.1 Класс Scene

Файл: `vulkan/renderer/Scene`

Класс `Scene` содержит всё необходимое для управления сценой:

- указатель на `GLFWwindow`;
- ссылку на экземпляр `VulkanApp`;
- камеру;
- список мешей — `std::vector<std::unique_ptr<Mesh»`;
- функции инициализации, обновления и отрисовки;
- данные для обработки взаимодействия с мышью.

Основные методы:

- `void init()` — инициализация сцены.
- `void update(float time)` — обновление положения камеры, объектов и обработка времени.
- `void draw(VkCommandBuffer cmd, VkPipelineLayout layout)` — отрисовка всех объектов.
- `void addMesh(std::unique_ptr<Mesh> mesh)` — добавление объекта в сцену.
- `Camera &getCamera()` — доступ к камере для управления.

Поддержка пользовательского взаимодействия: Класс также хранит флаг перетаскивания мышью и координаты указателя — это позволяет реализовать управление камерой.

3.2 Класс Mesh

Файл: `vulkan/renderer/Mesh.hpp`

Класс `Mesh` отвечает за представление отдельного объекта на сцене. Он хранит:

- вершины (`std::vector<Vertex>`);
- индексы (`std::vector<uint32_t>`);
- имя меша (для отладки/поиска);
- буферы вершин и индексов на GPU;
- объект трансформации `Transform`.

Буферы: При вызове `create()`, создаются Vulkan-буферы:

- `VkBuffer vertexBuffer`, `VkDeviceMemory vertexBufferMemory`
- `VkBuffer indexBuffer`, `VkDeviceMemory indexBufferMemory`

Рисование: Метод `draw(VkCommandBuffer commandBuffer)` записывает команды для отрисовки:

1. привязка vertex и index буферов;
2. вызов `vkCmdDrawIndexed`.

Конструкторы: Поддерживается создание меша как по вершинам и индексам, так и только по вершинам. Во втором случае — используется неиндексированная отрисовка (реже).

```
// :
std::unique_ptr<Mesh> mesh = std::make_unique<Mesh>(app, verts, inds, "Toru
scene.addMesh(std::move(mesh));
```

3.3 Взаимодействие между Scene и Mesh

`Scene::draw` вызывает метод `draw` у каждого объекта. При этом передаётся:

- `VkCommandBuffer` — для записи команд;
- `VkPipelineLayout` — необходим для push constants с матрицами.

`Scene` сама управляет аспектом окна, масштабом камеры и её положением. Обновление матриц трансформации и отправка их в шейдер происходит на этом этапе.

3.4 Особенности

- Каждому мешу соответствует уникальная трансформация: позиция, поворот, масштаб.
- Возможна реализация иерархий объектов (в будущем — через дочерние `SceneNode`).
- Трансформации меша применяются перед отрисовкой через push constants.

Очистка ресурсов: `Scene::cleanup()` вызывает `Mesh::cleanup()` для всех объектов и освобождает GPU-буферы.

4 Параметрические поверхности

Для одной из демонстраций визуализации 3D-объектов в движке используются параметрические поверхности. Каждая из них представлена в виде функции двух параметров u и v, принимающих значения на заданном отрезке, и задаёт трёхмерную форму.

4.1 Общий подход к построению поверхностей

Каждая поверхность реализована в виде функции, возвращающей:

- массив вершин (`std::vector<Vertex>`),
- массив индексов (`std::vector<uint32_t>`) для треугольников.

Все поверхности построены по схеме:

- два вложенных цикла по u и v;
- вычисление координат $x(u, v)$, $y(u, v)$, $z(u, v)$;
- раскладка вершин в сетку с треугольниками.

Примеры поверхностей:

- бутылка Клейна (`KleynData.hpp`); (Имеет небольшую специфику в силу сложности построения)
- лента Мёбиуса (`MobiusData.hpp`);
- тор (`ToroidalData.hpp`);
- спиральная поверхность (`SpiralData.hpp`);
- раковина (`ShellData.hpp`).

4.2 Пример: генерация поверхности раковины

Формула поверхности использует параметрическое описание, основанное на экспоненте:

$$\begin{aligned} x(u, v) &= \alpha \cdot e^{\beta v} \cdot \cos(v) \cdot (1 + \cos(u)) \\ y(u, v) &= \alpha \cdot e^{\beta v} \cdot \sin(v) \cdot (1 + \cos(u)) \\ z(u, v) &= \alpha \cdot e^{\beta v} \cdot \sin(u) \end{aligned}$$

Здесь α и β — управляющие параметры, задающие форму раковины (толщина, радиальное расширение и вытянутость).

Параметры сетки:

- $u \in [0, 2\pi]$ — угол по горизонтали (долота);
- $v \in [0, 2\pi]$ — угол "раскрутки"по вертикали;
- $uSegments, vSegments$ — количество разбиений по каждой оси.

```
float x = alpha * pow(e, beta * v) * cos(v) * (1 + cos(u));
float y = alpha * pow(e, beta * v) * sin(v) * (1 + cos(u));
float z = alpha * pow(e, beta * v) * sin(u);
```

Цветовая схема: Каждая вершина может получать цвет в зависимости от координат сетки. Например, каждая пятая по u или v может быть окрашена в чёрный:

```
if (i % 5 == 0 || j % 5 == 0)
    color = glm::vec3(0.0f);
else
    color = glm::vec3(0.75f);
```

Это создаёт визуальный эффект "сеточки помогающий анализировать геометрию формы.

Индексация: Треугольники генерируются с использованием стандартной двумерной сетки:

- четыре вершины — один квад;
- два треугольника на квад;
- всего: $2 \cdot uSegments \cdot vSegments$ треугольников.

Возвращаемое значение: `std::pair<std::vector<Vertex>, std::vector<uint32_t»`

5 Загрузка моделей в формате glTF

Для поддержки внешних моделей в движке используется формат **glTF 2.0** — компактный, кроссплатформенный формат хранения 3D-сцен и геометрии, разработанный Khronos Group.

Загрузка осуществляется с помощью библиотеки `tinygltf`, интегрированной в движок. Основная функция, отвечающая за обработку моделей, — это:

```
std::pair<std::vector<Vertex>, std::vector<uint32_t>>
LoadGLTFMesh_All(const std::string &path);
```

5.1 Общий принцип загрузки

1. Попытка загрузки бинарного glb-файла или текстового gltf.
2. Перебор всех узлов сцены.
3. Игнорирование служебных мешей (например, фона).
4. Построение матрицы трансформации узла.
5. Загрузка вершин: позиции и цвета.
6. Загрузка индексов с учётом типа (uint16/uint32).

5.2 Матрица трансформации узла

Каждый `node` может иметь:

- полную матрицу 4×4 (поле `node.matrix`),
- или отдельные поля: `translation`, `rotation` (в виде матриц Эйлера), `scale`.

Если матрица не задана явно, она формируется вручную:

$$\mathbf{T}_{node} = \mathbf{T}_{scale} \cdot \mathbf{R}_{quat} \cdot \mathbf{T}_{translate}$$

где:

- $\mathbf{T}_{scale} =$ `glm::scale(...)`
- $\mathbf{R}_{quat} =$ `glm::mat4_cast(glm::quat)`
- $\mathbf{T}_{translate} =$ `glm::translate(...)`

Эта трансформация применяется ко всем вершинам текущего примитива.

5.3 Загрузка вершин

Для каждой вершины:

- Позиции считываются из атрибута `"POSITION"`.
- При наличии цвета (`"COLOR_0"`) — также считывается цвет.
- Если цвет не задан, используется простое чередование оттенков.

```
//
glm::vec3 pos = glm::make_vec3(&posData[i * 3]);
glm::vec3 transformed = glm::vec3(nodeTransform * glm::vec4(pos, 1.0f));
vertices[startVertex + i].pos = transformed;

if (hasColor)
    vertices[startVertex + i].color = glm::make_vec3(&colorData[i * 3]);
else
    vertices[startVertex + i].color = glm::vec3(0.45f);
```

5.4 Загрузка индексов

В зависимости от типа индексов (uint16 или uint32), выбирается соответствующий способ интерпретации:

```
if (idxAccessor.componentType == TINYGLTF_COMPONENT_TYPE_UNSIGNED_SHORT) {
    const uint16_t *buf = ...;
    indices[i] = static_cast<uint32_t>(buf[i]) + baseVertex;
} else if (idxAccessor.componentType == TINYGLTF_COMPONENT_TYPE_UNSIGNED_IN
    const uint32_t *buf = ...;
    indices[i] = buf[i] + baseVertex;
}
```

Смещение индексов: Так как все меши грузятся в один массив, важно учитывать смещение `baseVertex`, чтобы индексы не ссылались на предыдущие вершины.

5.5 Фильтрация служебных объектов

Фоновые меши (например, плоскость с именем `"Plane001"`) отфильтровываются по имени:

```
if (node.name.find("Plane001") != std::string::npos)
    continue;
```

5.6 Итог

Функция возвращает:

- массив вершин с позициями и цветами;
- массив индексов для построения треугольников.

```
[GLTF::ALL] Loaded 1564 vertices, 2802 indices from model.gltf
```

6 Заключение и планы на развитие

Движок **Mezera Engine** на момент написания этой документации реализует полноценную Vulkan-пайплайн-сцену: с камерой, загрузкой параметрических поверхностей и моделей в формате glTF, системой трансформаций, отрисовкой по индексам и управлением ресурсами.

Планы на будущее

В рамках дальнейшей разработки планируется реализовать:

- **Собственный бинарный формат сцены** — для хранения информации об объектах, их позициях, связях, материалах и ресурсах. Это позволит быстро загружать сцены без парсинга glTF.
- **Поддержку текстурирования** — загрузка и привязка 2D- и возможно 3D-текстур к материалам, поддержка UV-координат в шейдерах.
- **Реалистичное освещение** — система источников света, расчёт освещённости.
- **Генерацию динамических эффектов** — например, процедурная жидкость, волны, частицы.
- **Интеграцию с AR-платформами** — возможность вывода сцены на реальные устройства дополненной реальности и взаимодействие с реальным миром.

Обновление документации

Документация создаётся вручную и не является автоматически синхронизированной с кодом. Ввиду темпов разработки и постоянных изменений в архитектуре движка:

Нет гарантии, что эта документация будет обновляться синхронно с кодовой базой.

Добавление новых подсистем, эксперименты с технологиями и параллельные прототипы могут идти быстрее, чем оформляется документация. Поэтому этот файл следует рассматривать как **частичный снимок** на конкретный этап развития проекта.

Несмотря на это, предпринимаются попытки сохранить читаемость и общую структуру движка, чтобы даже при неактуальности некоторых деталей он оставался понятным для сторонних разработчиков и для самого автора в будущем.