МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №3 по курсу
“Компьютерная графика”

Основы 3D графики

Выполнил: К.А. Ефременко
Группа: М8О-312Б-23
Преподаватель: В. Д. Бахарев

Москва, 2025

Условие

Цель работы

Освоить технологию текстурирования 3D-объектов: загрузить изображение из файла, создать текстуру и наложить её на объекты с использованием текстурных координат и сэмплеров. Изучить работу с дескрипторами текстур в Vulkan API.

Выполненное базовое условие на 4

В данной работе реализовано базовое условие на оценку 4, включающее:

- Загрузка изображений из файлов (с использованием библиотеки stb_image)
- Создание текстур через класс `veekay::graphics::Texture`
- Создание объектов сэмплеров (`VkSampler`) с настройкой фильтрации и режимов адресации
- Запись дескрипторов типа `combined image sampler` в descriptor sets
- Вершины содержат текстурные координаты (UV)
- Fragment shader выполняет сэмплирование текстур с интерполированными UVкоординатами
- **Использование разных текстур и сэмплеров для разных объектов** через систему материалов с отдельными descriptor sets

Дополнительные задания

Выполнены оба дополнительных задания:

1. **Нетривиальное сэмплирование текстуры:**
   - Модуляция UV-координат синусоидальными функциями для создания эффекта искажения
   - Множественная выборка (multi-sampling) — три сэмпла с разными масштабами
   - Смешивание результатов сэмплирования для улучшения детализации
2. **Дополнительные текстуры материалов:**
   - **Albedo texture** — базовая диффузная текстура (загружается из файла)
   - **Specular texture** — карта отражений для бликов (процедурная, металлический паттерн)
   - **Emissive texture** — карта свечения для самосветящихся участков (процедурная, сетка)

Метод решения

Процесс текстурирования

1. Загрузка изображения

Для загрузки изображений используется библиотека stb_image, которая поддерживает множество форматов (PNG, JPEG, BMP и др.):

int width, height, channels; unsigned char* pixels = stbi_load("texture.png", &width, &height, &channels, STBI_rgb_alpha); if (!pixels) { std::cerr << "Failed to load texture" << std::endl; // Использовать fallback текстуру }

Важно: изображение загружается с принудительным форматом RGBA (4 канала), что упрощает работу с Vulkan, так как можно использовать единый формат `VK_FORMAT_R8G8B8A8_SRGB`.

2. Создание текстуры в Vulkan

Процесс создания текстуры включает несколько этапов:

// Класс Texture инкапсулирует следующие объекты: VkImage image; // Объект изображения GPU VkDeviceMemory memory; // Выделенная память GPU VkImageView view; // View для доступа к изображению // Этапы создания: 1. Создание VkImage с параметрами (размер, формат, usage flags) 2. Выделение и привязка GPU памяти (VkDeviceMemory) 3. Копирование данных из staging buffer в VkImage 4. Переход из UNDEFINED в SHADER_READ_ONLY_OPTIMAL layout 5. Создание VkImageView для использования в шейдерах

Класс `veekay::graphics::Texture` автоматизирует эти шаги, принимая command buffer, размеры и данные пикселей.

3. Создание сэмплера

Сэмплер определяет, как текстура будет считываться в шейдере:

VkSamplerCreateInfo sampler_info{ .sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO, .magFilter = VK_FILTER_LINEAR, // Увеличение: линейная интерполяция .minFilter = VK_FILTER_LINEAR, // Уменьшение: линейная интерполяция .mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR, .addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT, // Повтор по оси U .addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT, // Повтор по оси V .addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT, .mipLodBias = 0.0f, .anisotropyEnable = VK_TRUE, // Анизотропная фильтрация .maxAnisotropy = 16.0f, // Максимальный уровень анизотропии .compareEnable = VK_FALSE, .minLod = 0.0f, .maxLod = VK_LOD_CLAMP_NONE, .borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE, .unnormalizedCoordinates = VK_FALSE }; vkCreateSampler(device, &sampler_info, nullptr, &texture_sampler);

Параметры фильтрации:

- `VK_FILTER_LINEAR` — билинейная интерполяция между текселями для плавного результата
- `VK_FILTER_NEAREST` — выбор ближайшего текселя (пиксельный эффект) **Режимы адресации:**
- `REPEAT` — повтор текстуры (для tileable паттернов)
- `CLAMP_TO_EDGE` — ограничение краевыми пикселями
- `MIRRORED_REPEAT` — зеркальное повторение

Анизотропная фильтрация улучшает качество текстур, рассматриваемых под углом (например, пол), используя до 16 сэмплов.

4. Descriptor Sets для текстур

Для передачи текстур в шейдеры используются дескрипторы типа `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`:

// Описание привязки в descriptor set layout VkDescriptorSetLayoutBinding texture_binding{ .binding = 3, // Binding index в шейдере .descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, .descriptorCount = 1, .stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT }; // Обновление descriptor set VkDescriptorImageInfo image_info{ .sampler = texture_sampler, .imageView = albedo_texture->view, .imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL }; VkWriteDescriptorSet write{ .sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET, .dstSet = descriptor_set, .dstBinding = 3, .descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, .descriptorCount = 1, .pImageInfo = &image_info }; vkUpdateDescriptorSets(device, 1, &write, 0, nullptr);

Для использования разных текстур на разных объектах создаётся несколько descriptor sets, каждый из которых связывает свой набор текстур.

5. Текстурные координаты в вершинах

Каждая вершина содержит UV-координаты, которые определяют, какая часть текстуры соответствует этой точке поверхности:

struct Vertex { Vector position; // Позиция в 3D пространстве Vector normal; // Нормаль для освещения Vector uv; // Текстурные координаты (x, y) ∈ [0, 1] }; // Пример для плоскости (quad) Vertex plane_vertices[] = { {{-5.0f, -1.0f, -5.0f}, {0, 1, 0}, {0, 0}}, // Левый нижний {{ 5.0f, -1.0f, -5.0f}, {0, 1, 0}, {5, 0}}, // Правый нижний {{ 5.0f, -1.0f, 5.0f}, {0, 1, 0}, {5, 5}}, // Правый верхний {{-5.0f, -1.0f, 5.0f}, {0, 1, 0}, {0, 5}} // Левый верхний };

UV-координаты могут выходить за диапазон [0, 1] — в этом случае поведение определяется режимом адресации сэмплера (повтор, отражение и т.д.).

6. Сэмплирование в шейдерах

Vertex Shader:

// Передача UV-координат во fragment shader layout (location = 2) in vec2 v_uv; layout

(location = 2) out vec2 f_uv; void main() { // ... трансформация позиции f_uv = v_uv; // Передача UV для интерполяции }

Fragment Shader (базовое сэмплирование):

layout (binding = 3) uniform sampler2D albedo_texture; layout (location = 2) in vec2 f_uv; void main() { vec3 color = texture(albedo_texture, f_uv).rgb; // ... дальнейшая обработка }

Функция `texture()` выполняет сэмплирование с учётом всех параметров сэмплера (фильтрация, адресация, анизотропия).

Нетривиальное сэмплирование

Для выполнения первого дополнительного задания реализована сложная схема сэмплирования:

1. Модуляция координат

// Искажение UV-координат синусоидальными функциями vec2 distorted_uv = f_uv; distorted_uv.x += sin(f_uv.y * 10.0 + f_position.y * 2.0) * 0.02; distorted_uv.y += cos(f_uv.x * 8.0 + f_position.x * 1.5) * 0.015;

Это создаёт волнообразное искажение текстуры, зависящее от позиции фрагмента в пространстве.

2. Множественная выборка с разными масштабами

// Три сэмпла с разными частотами vec3 albedo_sample1 = texture(albedo_texture, distorted_uv).rgb; vec3 albedo_sample2 = texture(albedo_texture, f_uv * 2.0).rgb; vec3 albedo_sample3 = texture(albedo_texture, f_uv * 0.5).rgb; // Смешивание с весами для итогового результата vec3 final_albedo = mix(mix(albedo_sample1, albedo_sample2, 0.3), albedo_sample3, 0.2) * albedo_color; Эта техника:

- Повышает детализацию за счёт комбинирования разных масштабов
- Создаёт более сложный визуальный эффект
- Уменьшает видимость повторяющихся паттернов

Система материалов с множественными текстурами

Для второго дополнительного задания реализована система материалов с тремя типами текстур:

Albedo Texture (Диффузная карта)

Определяет базовый цвет поверхности. Загружается из внешнего файла изображения.

// Используется для диффузной составляющей освещения vec3 final_albedo = texture(albedo_texture, f_uv).rgb * albedo_color; vec3 diffuse = light.color * light.intensity * diff * final_albedo;

Specular Texture (Карта отражений)

Определяет области с сильными бликами. Реализована как процедурная текстура с металлическим паттерном:

// Генерация процедурной specular карты for (int y = 0; y < 256; y++) { for (int x = 0; x < 256; x++) { float pattern = sin(x * 0.1f) * cos(y * 0.1f); float metallic = (pattern + 1.0f) * 0.5f; specular_data[(y * 256 + x) * 4 + 0] = metallic * 255; specular_data[(y * 256 + x) * 4 + 1] = metallic * 255; specular_data[(y * 256 + x) * 4 + 2] = metallic * 255; specular_data[(y * 256 + x) * 4 + 3] = 255; } } // В шейдере модулирует силу бликов vec3 specular_map = texture(specular_texture, f_uv).rgb * specular_color; vec3 specular = light.color * light.intensity * spec * specular_map;

Emissive Texture (Карта свечения)

Определяет самосветящиеся области, которые игнорируют расчёт освещения. Процедурная текстура с паттерном сетки:

// Генерация процедурной emissive карты (сетка) for (int y = 0; y < 256; y++) { for (int x = 0; x < 256; x++) { bool is_grid_line = (x % 32 < 2) || (y % 32 < 2); float emission = is_grid_line ? 1.0f : 0.0f; emissive_data[(y * 256 + x) * 4 + 0] = 0; emissive_data[(y * 256 + x) * 4 + 1] = emission * 200; emissive_data[(y * 256 + x) * 4 + 2] = emission * 255; emissive_data[(y * 256 + x) * 4 + 3] = 255; } } // В шейдере добавляется в конце, игнорируя освещение vec3 emissive_map = texture(emissive_texture, f_uv).rgb; result += emissive_map * 2.0; // Усиление свечения

Emissive карта позволяет создавать светящиеся элементы (например, неоновые линии) без расчёта освещения.

Организация Descriptor Sets для материалов

Каждый материал (набор текстур) требует отдельного descriptor set:

// Создание нескольких descriptor sets VkDescriptorSet descriptor_sets[3]; // Для плоскости и двух кубов // Обновление каждого набора со своими текстурами for (int i = 0; i < 3; i++) { VkDescriptorImageInfo texture_infos[] = { {texture_sampler, albedo_textures[i]->view, ...}, {texture_sampler, specular_texture->view, ...}, {texture_sampler, emissive_texture->view, ...} }; // Обновление bindings 3, 4, 5 vkUpdateDescriptorSets(device, writes, ...); } // При рендеринге привязываем нужный набор vkCmdBindDescriptorSets(cmd, ..., descriptor_sets[material_index], ...); vkCmdDrawIndexed(cmd, ...);

Архитектура решения

Компонент	Описание
Texture класс	Инкапсуляция VkImage, VkDeviceMemory, VkImageView для удобной работы
Staging Buffer	Промежуточный буфер для копирования данных из CPU в GPU
Image Layout Transitions	Pipeline barriers для перехода между UNDEFINED → TRANSFER_DST → SHADER_READ_ONLY

Sampler	Объект VkSampler с настройками фильтрации и адресации
Descriptor Sets	Множественные наборы для разных материалов (текстур)
Vertex Attributes	Position (vec3), Normal (vec3), UV (vec2)
Shader Bindings	3: Albedo, 4: Specular, 5: Emissive, 6: Shadow Map

Результаты

Демонстрация работы программы

Скриншот 1

Рис. 1. Демонстрация текстурированной сцены с использованием albedo текстуры красных роз. На переднем плане виден большой куб с детализированной текстурой цветов, демонстрирующей качественное билинейное сэмплирование с анизотропной фильтрацией. Справа видны два куба меньшего размера с другими текстурами. Плоскость-пол также текстурирована и отражает освещение сцены. Панель управления показывает настройки освещения: Directional Light (направленный свет) с белым цветом и интенсивностью 2.0. Видна сетка эмиссивного освещения на полу (голубые светящиеся линии), созданная с помощью emissive текстуры.

Скриншот 2

Рис. 2. Вид сцены с другого ракурса, демонстрирующий работу системы материалов с разными текстурами. Плоскость пола текстурирована изображением аниме-персонажа в синих тонах, что контрастирует с кубами. Передний красный куб использует текстуру роз, два задних куба имеют фиолетово-синюю цветовую схему с процедурными текстурами. Хорошо видно, как работает нетривиальное сэмплирование: на переднем кубе заметны искажения текстуры от синусоидальной модуляции UV-координат, создающие волнообразный эффект. Specular блики на кубах демонстрируют работу карты отражений с металлическим паттерном. Освещение настроено идентично первому скриншоту.

Функциональные возможности

Реализованная программа предоставляет следующие возможности:

Система текстурирования

- **Загрузка внешних изображений:** поддержка PNG, JPEG и других форматов через stb_image
- **Процедурные текстуры:** генерация текстур напрямую в коде (specular и emissive карты)
- **Множественные текстуры на объект:** каждый объект использует 3 текстуры одновременно
- **Разные материалы:** плоскость и кубы имеют различные наборы текстур через отдельные descriptor sets
- **Fallback текстура:** автоматическая замена при ошибке загрузки (розово-чёрный шахматный паттерн)

Качество сэмплирования

- **Билинейная фильтрация:** плавная интерполяция между текселями
- **Анизотропная фильтрация 16x:** улучшенное качество при взгляде под углом
- **Режим повтора (REPEAT):** бесшовное тайлинг текстур
- **Нетривиальное сэмплирование:** искажение UV-координат и мультисэмплинг

Интеграция с освещением

- **Albedo текстура** влияет на диффузную составляющую освещения
- **Specular текстура** модулирует силу и распределение бликов
- **Emissive текстура** добавляет свечение независимо от источников света
- **Shadow mapping** корректно работает с текстурированными объектами

Технические характеристики

Параметр	Значение
Формат текстур	VK_FORMAT_R8G8B8A8_SRGB (8 бит на канал, sRGB)
Размер albedo текстуры	Зависит от загруженного файла (рекомендуется степень двойки)
Размер процедурных текстур	256 × 256 пикселей
Фильтрация	Linear (mag/min) + Anisotropic 16x
Режим адресации	VK_SAMPLER_ADDRESS_MODE_REPEAT
Mipmap режим	VK_SAMPLER_MIPMAP_MODE_LINEAR
Количество descriptor sets	3 (для разных материалов)
Текстур на материал	4 (albedo, specular, emissive, shadow map)
Библиотека загрузки	stb_image (header-only)

Выводы

В ходе выполнения лабораторной работы была успешно освоена технология текстурирования 3D-объектов с использованием Vulkan API. Реализовано базовое условие на оценку 4 и выполнены оба дополнительных задания.

Приобретённые знания и навыки:

- **Загрузка изображений:** Освоил работу с библиотекой stb_image для загрузки различных форматов изображений. Понял важность формата RGBA и причины его предпочтительного использования в графических API
- **Создание текстур в Vulkan:** Изучил полный цикл создания текстуры:
  - Создание VkImage с правильными параметрами (usage, format, tiling)
  - Выделение и привязка GPU памяти
  - Использование staging buffer для передачи данных
  - Управление image layouts через pipeline barriers – Создание VkImageView для доступа из шейдеров

- **Сэмплеры:** Понял роль и настройку VkSampler:
  - Режимы фильтрации (nearest vs linear) и их визуальное влияние
  - Режимы адресации (repeat, clamp, mirror) для разных сценариев
  - Анизотропная фильтрация для улучшения качества под углом
  - Настройка mipmap levels и LOD (Level of Detail)
- **Descriptor Sets для текстур:** Освоил работу с дескрипторами типа COMBINED_IMAGE_SAMPLER:
  - Создание descriptor set layouts с текстурными привязками
  - Выделение descriptor sets из descriptor pool
  - Обновление дескрипторов через VkWriteDescriptorSet
  - Использование множественных descriptor sets для разных материалов
- **Текстурные координаты:** Изучил работу с UV-координатами:
  - Правильное добавление UV-атрибутов в вершинный буфер
  - Настройка vertex input state для передачи UV в шейдер
  - Интерполяция UV между вершинами треугольника
  - Масштабирование UV для повторения текстуры (tiling)
- **Сэмплирование в GLSL:** Освоил функции работы с текстурами в шейдерах:
  - Базовая функция `texture(sampler2D, vec2)`
  - Модуляция UV-координат через математические функции
  - Множественное сэмплирование с разными параметрами – Смешивание результатов для улучшения визуального качества • **Процедурные текстуры:** Научился генерировать текстуры программно: – Создание математических паттернов (синусоиды, сетки)
  - Контроль над каждым пикселем для специфических эффектов
  - Преимущества процедурной генерации (малый размер, параметризация)
- **Система материалов:** Реализовал PBR-подобную систему с множественными картами:
  - Разделение цветовой информации (albedo, specular, emissive)
  - Комбинирование текстурных данных с освещением
  - Понимание физической основы каждого типа текстуры • **Оптимизация памяти:** Понял важные аспекты работы с текстурами:
  - Использование степеней двойки для размеров (memory alignment)
  - Выбор подходящего формата (SRGB vs linear)
  - Переиспользование сэмплеров между текстурами
  - Освобождение ресурсов в правильном порядке

Технические достижения:

- Построена гибкая система текстурирования с поддержкой множественных материалов
- Реализовано продвинутое сэмплирование с модуляцией координат и мультисэмплингом
- Создана система из трёх взаимодополняющих карт текстур (albedo, specular, emissive)
- Корректная интеграция текстур с системой освещения Блинна-Фонга
- Достигнуто высокое визуальное качество благодаря анизотропной фильтрации
- Реализован fallback механизм при ошибках загрузки **Понимание концепций:**

Работа дала глубокое понимание того, как текстуры хранятся в GPU памяти и как происходит их выборка. Особенно ценным оказалось понимание разницы между image layouts (UNDEFINED, TRANSFER_DST, SHADER_READ_ONLY) и необходимости явных переходов между ними через pipeline barriers.

Важным инсайтом стало понимание того, что сэмплер — это не просто фильтр, а сложный объект, управляющий множеством аспектов доступа к текстуре: от выбора mipmap level до обработки граничных случаев при выходе UV за пределы [0, 1].

Практическое применение:

Полученные знания являются фундаментальными для любой работы с 3D-графикой. Текстурирование — это основа создания визуально привлекательных сцен, будь то игры, симуляторы или системы визуализации данных. Понимание работы с descriptor sets для текстур критично для эффективной организации материалов в больших проектах.

Опыт создания процедурных текстур открывает путь к генеративной графике, где сложные паттерны создаются математически, экономя память и предоставляя параметрический контроль.