Архитектурная спецификация и техническое руководство по разработке воксельного игрового движка

1. Введение и архитектурная парадигма

Разработка воксельного игрового движка, аналогичного Minecraft, представляет собой одну из наиболее сложных задач в области компьютерной графики и программной инженерии. В отличие от традиционных полигональных игр, где геометрия статична и создается художниками заранее, воксельный мир полностью динамичен, изменяем и состоит из миллионов дискретных единиц данных. Это накладывает уникальные требования к архитектуре памяти, алгоритмам рендеринга и организации файловой структуры.[1] В данном отчете представлен исчерпывающий технический анализ и план реализации движка, ориентированного на высокую производительность, модульность и масштабируемость.

1.1. Выбор архитектурного паттерна: ECS и DOD

Для управления сложностью игровых объектов и оптимизации использования кэша процессора рекомендуется отказаться от чистого объектно-ориентированного программирования (ООП) в пользу **Data-Oriented Design (DOD)** и архитектуры **Entity Component System (ECS)**.[3] В традиционном ООП объекты (например, class Zombie) хранят свои данные (позиция, здоровье, меш) и методы вместе. Это приводит к разбросу данных в памяти (cache misses) и затрудняет многопоточную обработку.

В предлагаемой архитектуре:

- **Entity (Сущность):** Просто уникальный идентификатор (ID).

- **Component (Компонент):** Структура данных без логики (например, PositionComponent, RenderComponent, PhysicsComponent).
- **System (Система):** Логика, обрабатывающая сущности с определенным набором компонентов (например, MovementSystem обновляет позицию всех сущностей, имеющих скорость).

Этот подход позволяет линейно располагать данные компонентов в памяти, что критически важно для обработки тысяч вокселей и сущностей.[5] Библиотека **EnTT** выбрана в качестве стандарта индустрии для реализации ECS в C++.

1.2. Технологический стек

Для обеспечения кроссплатформенности и прямого доступа к GPU выбран следующий стек [6]:

- **Язык:** C++20 (для использования концептов, модулей и улучшенной работы с шаблонами).
- **Графический API:** OpenGL 4.6 (Core Profile) или Vulkan (для продвинутой оптимизации). В данном отчете рассматривается OpenGL как баланс между сложностью и возможностями.
- **Управление окном:** GLFW.
- **Математика:** GLM.
- **Генерация шума:** FastNoiseLite.
- **GUI:** Dear ImGui (для отладки) + кастомный рендерер (для игрового HUD).

---

2. Организация файловой структуры и проекта

Правильная иерархия файлов критична для поддержки проекта. Мы используем структуру, разделяющую публичные интерфейсы, приватные реализации, ресурсы и сторонние библиотеки.[8]

2.1. Корневая директория (VoxelEngine_Root)

Корневая папка содержит мета-информацию проекта и сборочные скрипты.

- CMakeLists.txt — Основной файл конфигурации сборки.
- .gitignore — Правила исключения файлов из контроля версий.
- README.md, LICENSE — Документация и лицензия.
- vcpkg.json — Манифест зависимостей (если используется пакетный менеджер vcpkg).

2.2. Директория исходного кода (src)

Код разделен на функциональные модули. Это предотвращает создание монолитных файлов и упрощает навигацию.

```
src/
├── Core/ # Ядро движка
│  ├── Application.cpp/.h # Главный класс приложения
│  ├── Window.cpp/.h # Обертка над GLFW
│  ├── Input.cpp/.h # Обработка ввода (клавиатура, мышь)
│  ├── Logger.cpp/.h # Система логирования (spdlog)
│  └── Timer.cpp/.h # Управление временем (DeltaTime)
├── Renderer/ # Графическая подсистема
│  ├── Shader.cpp/.h # Компиляция и линковка шейдеров
│  ├── Texture.cpp/.h # Управление текстурами и атласами
│  ├── Buffer.cpp/.h # VBO, VAO, EBO, SSBO
│  ├── Camera.cpp/.h # Frustum Culling, View Matrix
│  └── BatchRenderer.cpp/.h # Рендеринг UI и частиц
├── Voxel/ # Воксельная логика
│  ├── Block.h # Определения типов блоков
│  ├── Chunk.cpp/.h # Данные чанка, RLE-сжатие
│  ├── ChunkMesh.cpp/.h # Greedy Meshing алгоритм
│  ├── World.cpp/.h # Хранение чанков, управление загрузкой
│  └── ChunkManager.cpp/.h # Многопоточная очередь генерации
├── Generation/ # Процедурная генерация
│  ├── Noise.cpp/.h # Обертка FastNoiseLite
│  ├── Biome.cpp/.h # Определение биомов
│  ├── Structure.cpp/.h # Парсинг.schematic файлов
│  └── TerrainGen.cpp/.h # Пайплайн генерации ландшафта
├── Entity/ # ECS и игровой процесс
│  ├── Components.h # POD-структуры компонентов
│  ├── Systems/ # Логические системы
```

```
│ │ ├── PhysicsSystem.cpp
│ │ ├── RenderSystem.cpp
│ │ └── PlayerControlSystem.cpp
│ └── Player.cpp/.h # Специфичная логика игрока
├── GUI/ # Пользовательский интерфейс
│ ├── ImGuiLayer.cpp/.h # Слой отладки
│ ├── MenuSystem.cpp/.h # Машина состояний меню
│ └── FontRenderer.cpp/.h # Рендеринг текста (FreeType)
├── Utils/ # Вспомогательные утилиты
│ ├── Math.h # Расширения GLM
│ ├── FileIO.cpp/.h # Чтение/запись бинарных/JSON файлов
│ └── Localization.cpp/.h # Менеджер языков
└── main.cpp # Точка входа
```

2.3. Директория ресурсов (assets)

Структура ассетов должна зеркалировать логику их использования. Эти файлы копируются в выходную директорию при сборке.

```
assets/
├── textures/
│ ├── atlas_blocks.png # Основной атлас 1024x1024
│ ├── atlas_items.png # Атлас предметов
│ └── ui/ # Элементы интерфейса
├── shaders/
│ ├── chunk.vert/.frag # Шейдеры мира
│ ├── ui.vert/.frag # Шейдеры интерфейса
│ └── skybox.vert/.frag # Шейдеры неба
├── structures/ # Файлы генерации
│ ├── trees/
│ │ ├── oak_small.json
│ │ └── birch_large.json
│ └── buildings/
│ └── village_house.json
├── lang/ # Локализация
│ ├── en_US.json
│ └── ru_RU.json
└── fonts/ # Шрифты
└── main_font.ttf
```

3. Установка и интеграция библиотек

Интеграция сторонних библиотек является критическим этапом. Для C++ рекомендуется использовать менеджер пакетов (например, vcpkg или Conan) или подмодули Git для фиксации версий.

3.1. Необходимые библиотеки и их роль

Библиотека	Версия	Назначение	Обоснование
GLFW	3.3+	Оконный менеджер	Стандарт де-факто для OpenGL, кроссплатформенная обработка ввода.[6]
GLAD	2.0	Загрузчик OpenGL	Позволяет использовать функции современных версий OpenGL (4.5/4.6).
GLM	0.9.9	Математика	Полная совместимость с типами GLSL (vec3, mat4), оптимизация SIMD.
EnTT	3.11	ECS	Самая производительная

			реализация ECS на C++.[10]
FastNoiseLite	1.0	Шум	Высокопроизводительная генерация шума (Simplex, Perlin, Cellular) для ландшафта.
nlohmann/json	3.11	JSON Парсер	Удобная сериализация/десериализация конфигов и структур.[11]
stb_image	2.28	Загрузка изображений	Легковесная библиотека (один заголовочный файл) для декодирования PNG/JPG.
Dear ImGui	1.89	GUI	Инструментарий для создания отладочных панелей и редакторов внутри движка.[12]

3.2. Конфигурация CMake (CMakeLists.txt)

Пример скрипта сборки, связывающего все компоненты. Мы используем современный CMake с явным указанием стандартов C++.

CMake

```
cmake_minimum_required(VERSION 3.20)
project(VoxelEngine VERSION 1.0.0 LANGUAGES CXX C)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Опции
option(BUILD_SHARED_LIBS "Build using shared libraries" OFF)

# Подключение зависимостей (предполагается наличие в папке libs/)
add_subdirectory(libs/glfw)
add_subdirectory(libs/glad) # Или подключение сгенерированного.c файла
add_subdirectory(libs/imgui)

# Подключение header-only библиотек
include_directories(libs/glm)
include_directories(libs/entt/src)
include_directories(libs/fastnoise)
include_directories(libs/json/include)
include_directories(libs/stb)

# Сбор всех исходных файлов
file(GLOB_RECURSE SOURCES "src/*.cpp" "src/*.h")

# Создание исполняемого файла
add_executable(VoxelEngine ${SOURCES})

# Линковка
target_link_libraries(VoxelEngine PRIVATE
    glfw
    glad
    imgui
    ${CMAKE_DL_LIBS} # Для Linux
)

# Копирование ассетов после сборки
add_custom_command(TARGET VoxelEngine POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_directory
    ${CMAKE_SOURCE_DIR}/assets ${CMAKE_BINARY_DIR}/assets)
```

---

4. Спецификация блоков и текстурного атласа

Согласно требованиям, используется текстурный атлас размером 1024x1024 пикселя. Атлас разбит на сетку 16x16 ячеек. Это означает, что размер каждой индивидуальной текстуры блока составляет:

$$1024 \div 16 = 64 \text{ пикселя}$$

Всего доступно $16 \times 16 = 256$ уникальных текстурных слотов.

4.1. Математика UV-координат

Для корректного отображения текстуры на грани вокселя необходимо преобразовать целочисленные координаты ячейки (Col, Row) в нормализованные текстурные координаты (UV) в диапазоне [0.0, 1.0].[13]

Параметры:

- ATLAS_SIZE = 1024.0
- CELL_SIZE = 64.0
- GRID_DIM = 16.0
- UV_STEP = $1.0 / 16.0 = 0.0625$

Для избежания артефактов "кровоточения" (texture bleeding) при использовании MIP-маппинга, необходимо добавлять небольшой отступ (padding) внутрь текстуры или использовать технику Texture Arrays. Однако, для классического атласа формула расчета UV для блока с индексом ID выглядит так:

$$Col = ID \% 16 \\ Row = ID / 16 \\ U_{min} = Col \times UV\_STEP + \epsilon \\ V_{min} = Row \times UV\_STEP + \epsilon \\ U_{max} = (Col + 1) \times UV\_STEP - \epsilon \\ V_{max} = (Row + 1) \times UV\_STEP - \epsilon$$
Где ϵ (epsilon) — малая величина (например, 0.0001 или половина текселя 0.5 / 1024), предотвращающая выборку пикселей из соседних ячеек.

4.2. Таблица распределения блоков (Block ID List)

Ниже представлена детальная карта распределения 256 текстур по атласу. Индекс ID рассчитывается как (Row * 16) + Col. Координаты (0,0) соответствуют левому верхнему углу (в зависимости от API, OpenGL может требовать инверсии по Y).[15]

Группа 0: Природные блоки (Row 0)

ID	Название	Col	Row	Описание
0	**Air**	0	0	Пустой блок (не рендерится)
1	**Stone**	1	0	Базовый камень
2	**Dirt**	2	0	Земля
3	**Grass_Top**	3	0	Трава сверху (окрашиваемая биомом)
4	**Grass_Side**	4	0	Трава сбоку
5	**Sand**	5	0	Песок
6	**Gravel**	6	0	Гравий
7	**Bedrock**	7	0	Коренная порода
8	**Water_Still**	8	0	Вода (статичная)
9	**Lava_Still**	9	0	Лава

				(статичная)
10	**Snow**	10	0	Снег (блок)
11	**Ice**	11	0	Лед
12	**Log_Oak**	12	0	Кора дуба (бок)
13	**Log_Top**	13	0	Срез дерева (годичные кольца)
14	**Leaves_Oak**	14	0	Листва дуба (прозрачность)
15	**Cactus**	15	0	Кактус (бок)

Группа 1: Руды и минералы (Row 1)

ID	Название	Col	Row	Описание
16	**Ore_Coal**	0	1	Угольная руда
17	**Ore_Iron**	1	1	Железная руда
18	**Ore_Gold**	2	1	Золотая руда
19	**Ore_Diamond**	3	1	Алмазная руда
20	**Ore_Emerald**	4	1	Изумрудная

				руда
21	**Ore_Redstone**	5	1	Редстоун руда
22	**Ore_Lapis**	6	1	Лазурит
23	**Block_Iron**	7	1	Железный блок
24	**Block_Gold**	8	1	Золотой блок
25	**Block_Diam**	9	1	Алмазный блок
26	**Obsidian**	10	1	Обсидиан
27	**Glowstone**	11	1	Светящийся камень
28	**Netherrack**	12	1	Адский камень
29	**Soul_Sand**	13	1	Песок душ
30	**Quartz_Ore**	14	1	Кварцевая руда
31	**Magma**	15	1	Магма блок

Группа 2: Строительные материалы (Row 2)

ID	Название	Col	Row	Описание
32	**Planks_Oak**	0	2	Доски

				дубовые
33	**Planks_Spruce**	1	2	Доски еловые
34	**Planks_Birch**	2	2	Доски березовые
35	**Cobblestone**	3	2	Булыжник
36	**Mossy_Cobble**	4	2	Мшистый булыжник
37	**Brick_Clay**	5	2	Кирпич (глиняный)
38	**Stone_Brick**	6	2	Каменный кирпич
39	**Bookshelf**	7	2	Книжная полка
40	**Sandstone_Sd**	8	2	Песчаник (бок)
41	**Sandstone_Tp**	9	2	Песчаник (верх)
42	**Glass**	10	2	Стекло (рамка)
43	**Wool_White**	11	2	Белая шерсть
44	**Wool_Red**	12	2	Красная шерсть
45	**Wool_Green**	13	2	Зеленая шерсть

46	**Wool_Blue**	14	2	Синяя шерсть
47	**Wool_Black**	15	2	Черная шерсть

... (Продолжение таблицы по аналогии для строк 3-14 с различными вариациями блоков, растений, механизмов)...

Группа 15: Интерфейс и технические текстуры (Row 15)

Эта строка резервируется для элементов GUI, которые рендерятся тем же шейдером, или для технических нужд.

ID	Название	Col	Row	Описание
240	**UI_Crosshair**	0	15	Прицел
241	**UI_Heart_Full**	1	15	Иконка здоровья (полное)
242	**UI_Heart_Half**	2	15	Иконка здоровья (половина)
243	**UI_Heart_Emp**	3	15	Иконка здоровья (пустое)
244	**UI_Food_Full**	4	15	Иконка голода
...	...	...	...	...
254	**Destroy_Stg9**	14	15	Анимация разрушения (стадия 9)

255	**Missing_Tex**	15	15	Текстура ошибки (розово-черная шахматка)

---

5. Движок генерации мира и структур

Генерация мира является одной из самых ресурсоемких задач. Архитектура должна поддерживать асинхронную генерацию, чтобы не блокировать основной поток рендеринга.

5.1. Пайплайн генерации чанка

Используется многоступенчатый подход:

1. **Shape Generation (Форма):** Определение плотности породы с использованием 3D шума Перлина/Симплекса.[17]
   - Density = Noise3D(x, y, z) - (y / WorldHeight)
   - Если Density > 0, блок устанавливается как Stone.
2. **Biome Map (Биомы):** Генерация 2D карт температуры и влажности. На их основе выбирается тип поверхности (песок для пустыни, трава для леса, снег для тундры).
3. **Surface Pass (Поверхность):** Замена верхних слоев камня на блоки, соответствующие биому (3-4 слоя земли/травы).
4. **Cave Generation (Пещеры):** Вычитание "червей" шума (Perlin Worms) из массива блоков для создания полостей.

5.2. Система генерации построек (Structure Engine)

Постройки (деревья, дома, данжи) накладываются поверх сгенерированного ландшафта. Основная проблема здесь — выход структуры за границы чанка.

5.2.1. Формат файлов структур (.json)

В соответствии с требованием читаемости и простоты редактирования, для хранения структур выбран формат JSON. Хотя бинарные форматы эффективнее, JSON позволяет легко отлаживать смещения и ID блоков.[18]

Пример файла assets/structures/tree_oak.json:

JSON

```
{
  "meta": {
    "name": "Oak_Tree_Default",
    "author": "System",
    "dimensions": { "x": 5, "y": 6, "z": 5 },
    "center_offset": { "x": 2, "y": 0, "z": 2 }
  },
  "blocks": [
    { "x": 2, "y": 0, "z": 2, "id": "log_oak" },
    { "x": 2, "y": 1, "z": 2, "id": "log_oak" },
    { "x": 2, "y": 2, "z": 2, "id": "log_oak" },
    { "x": 2, "y": 3, "z": 2, "id": "log_oak" },
    { "x": 1, "y": 3, "z": 1, "id": "leaves_oak" },
    { "x": 1, "y": 3, "z": 2, "id": "leaves_oak" },
    { "x": 1, "y": 3, "z": 3, "id": "leaves_oak" },
    ...
  ]
}
```

5.2.2. Алгоритм вставки структур

1. При генерации чанка проверяются точки потенциального спавна структур (на основе фиксированного сида).
2. Если структура должна появиться, её данные считываются.
3. **Cross-Chunk Placement:** Если блок структуры попадает в координаты соседнего чанка, который еще не сгенерирован, он добавляется в глобальную карту

PendingModifications. При последующей генерации соседнего чанка, движок сначала применяет модификации из этой карты.

---

6. Персонаж и физика (Character Engine)

Персонаж реализуется как отдельный модуль, независимый от сетки вокселей, но взаимодействующий с ней.

6.1. Класс игрока (Player)

Игрок не привязан к целочисленным координатам. Он имеет float-позицию и AABB (Axis-Aligned Bounding Box) для коллизий.

C++

```
class Player {
public:
    glm::vec3 position;
    glm::vec3 velocity;
    glm::vec3 rotation; // Pitch, Yaw
    AABB boundingBox = { glm::vec3(-0.3f, 0.0f, -0.3f), glm::vec3(0.3f, 1.8f, 0.3f) };

    bool isGrounded;
    float walkSpeed = 4.3f;
    float jumpHeight = 1.1f;

    void Update(float dt, World& world);
    void ProcessInput(Input& input);
};
```

6.2. Физика вокселей

Используется алгоритм "Sweep and Prune" или дискретная проверка пересечений.

1. Определяются целочисленные координаты вокселей, которые пересекает AABB игрока.
2. Для каждого твердого вокселя создается статический AABB.
3. Проверяется пересечение AABB игрока с AABB блоков.
4. При коллизии позиция игрока корректируется ("выталкивается") по оси минимального проникновения, а скорость по этой оси обнуляется.

---

7. Интерфейс (UI) и Меню

Система интерфейса должна быть разделена на два уровня:

1. **Debug/Editor UI:** Реализуется через **Dear ImGui**. Позволяет менять параметры генерации мира, настройки графики и просматривать состояние ECS в реальном времени.
2. **Game HUD/Menu:** Реализуется через собственный рендерер (BatchRenderer) для стилистического соответствия игре.

7.1. Машина состояний (State Machine)

Для управления переходами между меню и игрой используется паттерн State Machine.[20]

C++

```
// Базовый класс состояния
class IGameState {
public:
    virtual void OnEnter() = 0;
    virtual void OnUpdate(float dt) = 0;
    virtual void OnRender() = 0;
    virtual void OnExit() = 0;
};
```

```
// Менеджер состояний
class StateManager {
    std::stack<std::unique_ptr<IGameState>> states;
public:
    void PushState(std::unique_ptr<IGameState> state);
    void PopState();
    void Update(float dt) { states.top()->OnUpdate(dt); }
};
```

Реализуемые состояния:

- MainMenuState: Отрисовка кнопок "Play", "Settings", "Exit".
- GameState: Основной цикл игры, управление камерой, HUD.
- PauseMenuState: Оверлей поверх игры с заморозкой времени.

7.2. Рендеринг HUD

HUD (прицел, хотбар, здоровье) рендерится в ортогональной проекции. Текстуры берутся из строки 15 основного атласа или отдельного UI-атласа. Координаты вершин пересчитываются при изменении размера окна.

---

8. Локализация (Localization System)

Система локализации должна поддерживать динамическое переключение языков без перезагрузки. Формат хранения данных — JSON.[19]

8.1. Файловая структура языковых пакетов

Файлы хранятся в assets/lang/. Ключи строятся иерархически для удобства.

Файл en_US.json:

JSON

```
{
    "menu": {
        "title": "Voxel Game",
        "play": "Singleplayer",
        "quit": "Quit Game"
    },
    "block": {
        "stone": "Stone",
        "dirt": "Dirt",
        "diamond_ore": "Diamond Ore"
    },
    "message": {
        "respawn": "You died! Press R to respawn."
    }
}
```

Файл ru_RU.json:

JSON

```
{
    "menu": {
        "title": "Воксельная Игра",
        "play": "Одиночная игра",
        "quit": "Выход"
    },
    "block": {
        "stone": "Камень",
        "dirt": "Земля",
        "diamond_ore": "Алмазная руда"
    },
    "message": {
        "respawn": "Вы погибли! Нажмите R для возрождения."
    }
```

```
}
```

8.2. Реализация менеджера (LocalizationManager)

Менеджер при старте загружает список доступных файлов. При выборе языка, он парсит соответствующий JSON в std::unordered_map<std::string, std::string>.

- **Функция Get(const std::string& key):** Ищет ключ в карте. Если ключ не найден, возвращает сам ключ (для отладки пропущенных переводов).
- **Поддержка Unicode:** Для корректного отображения кириллицы и других символов необходимо использовать библиотеку **FreeType** для загрузки шрифтов .ttf и генерации текстурного атласа глифов на лету. Стандартные ASCII-шрифты не подойдут. Строки в C++ должны обрабатываться как std::string (UTF-8) или std::wstring.

---

9. Поэтапный план работы (Roadmap)

Разработка разбита на логические фазы (спринты), каждый из которых завершается рабочим билдом.

Фаза 1: Инициализация и Графическое Ядро (Недели 1-2)

- **Цель:** Окно, контекст OpenGL, рендеринг тестового куба.
- **Задачи:**
  1. Настройка CMake проекта, подключение GLFW, GLAD, GLM.
  2. Создание абстракций Window, Input, Shader.
  3. Реализация камеры Camera (First Person).
  4. Загрузка текстурного атласа 1024x1024 через stb_image.

Фаза 2: Воксельная База (Недели 3-4)

- **Цель:** Рендеринг одного чанка (16x16x256) с текстурами.

- **Задачи:**
    1. Создание класса Chunk и заполнение его тестовыми данными (шум 2D).
    2. Реализация алгоритма мешинга (Naive Meshing) — генерация только видимых граней.
    3. Расчет UV-координат для выборки текстур из атласа.
    4. Базовая оптимизация: использование glDrawElements и индексного буфера.

Фаза 3: Мир и Генерация (Недели 5-7)

- **Цель:** Бесконечный мир, процедурная генерация.
- **Задачи:**
    1. Реализация World для управления сеткой чанков (Dynamic Loading/Unloading).
    2. Интеграция FastNoiseLite. Генерация 3D ландшафта.
    3. Многопоточность: вынос генерации и мешинга чанков в ThreadPool.
    4. Frustum Culling: отсечение чанков, не попадающих в поле зрения камеры.

Фаза 4: Игрок и Физика (Недели 8-9)

- **Цель:** Перемещение по миру с коллизиями.
- **Задачи:**
    1. Реализация AABB физики и гравитации.
    2. Raycasting: выделение блока, на который смотрит игрок (Voxel traversal algorithm).
    3. Логика установки и разрушения блоков (обновление меша чанка).

Фаза 5: Структуры, UI и Локализация (Недели 10-12)

- **Цель:** Полноценный игровой цикл.
- **Задачи:**
    1. Парсер .json структур и алгоритм их вставки в мир.
    2. Реализация MainMenuState и GameState.
    3. Рендеринг HUD (прицел, выбор блока).
    4. Интеграция системы локализации и поддержка русского языка.

Фаза 6: Оптимизация и Полировка (Недели 13+)

- **Цель:** Стабильные 60+ FPS, визуальные улучшения.
- **Задачи:**
    1. Внедрение **Greedy Meshing** (объединение соседних граней в одну для уменьшения полигонов).
    2. Ambient Occlusion (затемнение углов) через атрибуты вершин.
    3. Сохранение и загрузка мира в бинарный формат (Region files).

Этот план обеспечивает структурированный подход к разработке, начиная с фундаментальных систем и заканчивая высокоуровневыми игровыми механиками.