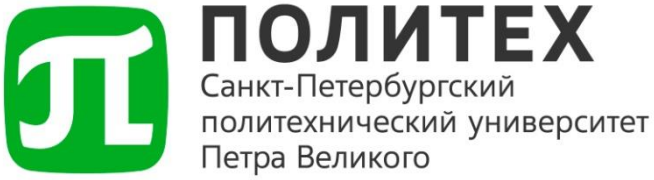
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»



Физико-механический институт
Высшая школа прикладной математики и вычислительной физики

Отчет о научно-исследовательской работе

по теме:

Исследование и анализ основных принципов и ключевых компонент DirectX для разработки оптимизированных графических приложений.

Место выполнения СПбПУ Петра Великого, кафедра ВШ ПМ и ВФ

Студент группы 5040102/50201_____ А.А. Худина

Оценка руководителя НИР: Отлично_____В.С. Чуканов

к.ф.-м.н., доц. ВШ ПМ и ВФ

Санкт-Петербург
2026 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современные графические приложения, от видеоигр и систем виртуальной реальности до инструментов научной визуализации, предъявляют высокие требования к производительности и реалистичности формируемого изображения. Эти требования удовлетворяются за счёт сложного взаимодействия программного обеспечения с графическими процессорами (*GPU*) через специализированные интерфейсы – графические *API* (*Application Programming Interface*). Эволюция данных интерфейсов привела к возникновению как высокоуровневых, так, впоследствии, и низкоуровневых моделей программирования, каждая из которых решает определённый круг задач и требует от разработчика соответствующей подготовки.

Актуальность темы обусловлена необходимостью формирования системного понимания фундаментальных принципов, лежащих в основе графических *API*. Эффективное освоение и использование современных низкоуровневых технологий (например, *DirectX 12*) возможно лишь при условии ясного понимания тех базовых концепций, архитектурных решений и моделей программирования, которые были сформированы в их предшественниках. В данном контексте *DirectX 11* представляет собой оптимальный объект для исследования: являясь высокоуровневым, но сохраняющим непосредственную близость к аппаратному обеспечению, он позволяет изучить ключевые концепции графического конвейера, управления ресурсами, шейдерного программирования и оптимизации. Это делает его идеальной платформой для формирования методологической базы, необходимой для последующего перехода к низкоуровневым *API*, где требуется глубокое понимание этих же принципов, но в условиях ручного управления памятью и синхронизацией.

Таким образом, актуальной целью настоящей научно-

исследовательской работы является проведение комплексного теоретико-прикладного исследования архитектуры, основных принципов и ключевых компонент графического *API DirectX 11*. Исследование направлено на формирование целостной методологической базы для последующего изучения более сложных и низкоуровневых графических технологий.

Для достижения поставленной цели необходимо решить следующие сформулированные задачи:

1) проведение системнного анализа архитектуры и базовых механизмов *DirectX 11*, включая устройство, контекст, систему ресурсов и их представлений и организацию графического конвейера;

2) исследование и практическая реализация таких ключевых техник рендеринга, как работа с системами координат и геометрическими преобразованиями, текстурирование, модели освещения с применением карты нормалей (модель Фонга), управление порядком отрисовки через буфер глубины и работа с прозрачностью и смешиванием;

3) анализ и реализация на практике таких методов оптимизации рендеринга, как инстанцирование, отсечение по пирамиде видимости (*frustum culling*), многоэтапный рендеринг с использованием промежуточных целей отрисовки (*render targets*), а также выполнение асинхронных вычислений на GPU с помощью вычислительных шейдеров.

ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ГРАФИЧЕСКОГО *API DIRECTX 11* И КОМПЬЮТЕРНОЙ ГРАФИКИ

1.1. Архитектура *DirectX 11*: ключевые абстракции и управление ресурсами

Графический *API* (*Application Programming Interface*) *DirectX 11* представляет собой высокоуровневую, но при этом детализированную платформу для программирования графических приложений в среде *Windows*. Его архитектура строится на объектно-ориентированной модели *COM* (*Component Object Model*) и реализует чёткое разделение ответственности между основными компонентами, что упрощает управление сложным графическим конвейером.

1.1.1. Модель программирования: устройство и контекст

Основными объектами, через которые осуществляется взаимодействие с графическим оборудованием (*GPU*), являются устройство (*ID3D11Device*) и контекст устройства (*ID3D11DeviceContext*). Устройство выступает в роли фабрики, ответственной за создание всех видов ресурсов: буферов, текстур, шейдеров и объектов, определяющих состояние конвейера. Этот интерфейс потокобезопасен и предназначен для инициализационных задач.

Непосредственное управление графическим конвейером и выполнение операций рендеринга возложено на контекст устройства. Контекст инкапсулирует текущее состояние конвейера (привязанные шейдеры, ресурсы, настройки) и предоставляет методы для выполнения отрисовки (*Draw, DrawIndexed*), обновления ресурсов и управления вычислительными шейдерами. Работа с контекстом, в отличие от устройства, должна выполняться из одного потока, что формирует более простую и предсказуемую модель выполнения по сравнению с низкоуровневыми *API*.

1.1.2. Система ресурсов и представления

Ресурсы в *DirectX 11* – это абстракции данных, размещённых в памяти *GPU*. Ключевыми их типами являются буферы (*ID3D11Buffer*) для линейных данных (вершины, индексы, константы) и текстуры (*ID3D11Texture1D/2D/3D*) для многомерных данных, таких как изображения. Принципиально важной концепцией является разделение ресурса и его представления (*Resource View*). Один и тот же физический ресурс может использоваться на разных этапах конвейера по-разному благодаря специальным объектам-представлениям:

1) *Render Target View* (*RTV*) позволяет использовать текстуру как цель для вывода цветовой информации;

2) *Depth Stencil View* (*DSV*) предназначено для использования текстуры в качестве буфера глубины и трафарета;

3) *Shader Resource View* (*SRV*) позволяет шейдерам читать данные из текстуры или буфера;

4) *Unordered Access View* (*UAV*) предоставляет возможность вычислительным шейдерам производить запись в ресурс в произвольном порядке.

Такая архитектура обеспечивает гибкость, безопасность типов и позволяет эффективно реализовывать многоэтапные алгоритмы рендеринга. Для достижения нетривиального графического эффекта или корректного отображения объекта со сложным затенением конвейер рендеринга должен быть тщательно настроен: должны быть установлены правильные цели рендеринга, состояния и шейдерные программы, входные данные шейдеров должны быть привязаны к корректным слотам и так далее [1].

1.1.3. Цепочка обмена и организация вывода

Для связи с оконной системой и вывода конечного изображения используется объект *IDXGISwapChain*, являющийся частью инфраструктуры *DXGI* (*DirectX Graphics Infrastructure*). Цепочка обмена управляет набором

текстур (обычно двумя или тремя), связанных с окном приложения. Пока в один буфер (*back buffer*) происходит отрисовка текущего кадра, содержимое другого (*front buffer*) отображается на экране. По завершении кадра происходит операция представления (*Present*), в результате которой буферы меняются местами. Такая двойная или тройная буферизация устраняет артефакты разрыва кадра и обеспечивает плавную анимацию.

1.1.4. Отладочный слой

Для облегчения разработки *DirectX 11* включает мощный отладочный слой (*Debug Layer*). При его активации во время создания устройства (флаг *D3D11_CREATE_DEVICE_DEBUG*) *runtime* начинает валидировать все вызовы *API*, проверяя корректность передаваемых параметров, отслеживая утечки ресурсов и предупреждая о потенциально неоптимальных действиях. Все диагностические сообщения выводятся в отладочную консоль среды разработки, что делает этот инструмент незаменимым для поиска и устранения ошибок.

1.2. Графический конвейер *DirectX 11*: этапы и программируемые компоненты

Графический конвейер представляет собой последовательность этапов, через которые данные преобразуются из описания трёхмерной сцены в итоговое растровое изображение. Конвейер *DirectX 11* является гибридным, сочетая фиксированные и программируемые ступени.

1.2.1. Этапы графического конвейера

Конвейер можно разделить на следующие ключевые этапы:

1) *Input Assembler* (*IA*, сборщик входных данных): на этом фиксированном этапе происходит сборка вершинных данных из буферов в примитивы (треугольники, линии) в соответствии с заданной топологией; также здесь определяется, как данные из вершинного буфера сопоставляются с входными параметрами вершинного шейдера, с использованием объекта разметки (*ID3D11InputLayout*);

2) *Vertex Shader* (*VS*, вершинный шейдер): программируемый этап, выполняемый для каждой вершины, чья основная задача – преобразование координат вершины из локального пространства объекта в однородное пространство отсечения (*clip space*) путём умножения на комбинированную матрицу вида-проекции-модели (здесь также подготавливаются другие данные (нормали, текстурные координаты) для передачи в следующие этапы);

3) *Rasterizer* (*RS*, растеризатор): фиксированный этап, определяющий, какие пиксели экрана покрывает каждый треугольник – здесь происходит отсечение геометрии за пределами видимой области, собственно растеризация (преобразование в набор фрагментов) и интерполяция атрибутов вершин (цвет, текстурные координаты) для каждого фрагмента с учётом перспективной коррекции;

4) *Pixel Shader* (*PS*, пиксельный шейдер): программируемый этап, вызываемый для каждого фрагмента – на его основе вычисляется итоговый цвет пикселя; здесь выполняются наиболее ресурсоёмкие визуальные вычисления: выборка из текстур, расчёт освещения и применение материалов;

5) *Output Merger* (*OM*, блок слияния вывода): финальный фиксированный этап, где происходит проверка глубины (*depth test*) для отбрасывания невидимых фрагментов, работа с трафаретным буфером и, при необходимости, смешивание (*blending*) вычисленного цвета с цветом, уже находящимся в цели рендеринга.

1.2.2. Шейдерное программирование и *HLSL*

Программируемые этапы конвейера реализуются с помощью шейдеров, написанных на языке высокого уровня *HLSL* (*High-Level Shading Language*). Шейдеры представляют собой специальные программы, исполняемые непосредственно графическим процессором, изначально предназначенные для реализации сложных графических эффектов посредством тонкого управления светом и тенью [2,3]. Процесс разработки

шейдера включает его компиляцию из исходного кода в промежуточный байт-код (*DXBC*) с помощью утилиты *D3DCompile*, а затем создание соответствующего объекта шейдера (*ID3D11VertexShader*, *ID3D11PixelShader*) через методы устройства, что инициирует финальную компиляцию драйвером в машинный код конкретного *GPU*.

1.2.3. Константные буферы

Для передачи данных из приложения в шейдеры используются константные буферы (*ID3D11Buffer* с флагом *D3D11_BIND_CONSTANT_BUFFER*). Это высокоскоростные области памяти, предназначенные для хранения информации, общей для множества вызовов шейдера. Эффективная организация предполагает группировку данных по частоте обновления:

1) обновляемые каждый кадр: матрица вида-проекции, параметры камеры, глобальные источники света;

2) обновляемые для каждого объекта: матрица модели, свойства материала;

3) статические: неизменные настройки сцены.

1.3. Математические основы *3D*-графики

Основой рендеринга трёхмерной сцены является строгая иерархия систем координат и линейно-алгебраический аппарат для преобразований между ними. Когда речь идёт о трёхмерной графике, любая сцена, которую мы пытаемся визуализировать, изначально представляет собой набор вершин, заданных в локальной системе координат объекта [4].

1.3.1. Иерархия систем координат

Вершина последовательно проходит через несколько пространств:

1) Локальное пространство (*Local/Object Space*) – исходные координаты вершины, определённые при моделировании объекта;

2) Мировое пространство (*World Space*) – координаты вершины

после применения матрицы модели (*M*), включающей перемещение, вращение и масштабирование объекта в виртуальном мире;

3) Пространство вида (*View/Camera Space*) – координаты относительно камеры; преобразование задаётся матрицей вида (*V*), которая является обратной к матрице положения и ориентации камеры в мире;

4) Однородное пространство отсечения (*Clip Space*) – результат умножения на матрицу проекции (*P*); видимая область в этом пространстве представляет собой куб, вершины за его пределами отсекаются;

5) Нормализованное координатное пространство устройства (*NDC*) получается перспективным делением координат из *clip space* на компоненту *w*: в *NDC* видимая область – это куб с координатами от (-1, -1, 0) до (1, 1, 1);

6) Экранное пространство (*Screen Space*) – финальные координаты пикселя в целевом буфере.

1.3.2. Матричные преобразования и *DirectXMath*

Ключевые преобразования (модели, вида, проекции) реализуются с помощью матриц 4×4 в однородных координатах. В *DirectX* принят *column-major* порядок хранения матриц и умножение матрицы на вектор-столбец. Итоговая цепочка преобразований записывается как формуле (1):

$$v' = P \cdot V \cdot M \cdot v. \quad (1)$$

Для эффективной работы с математическим аппаратом используется библиотека *DirectXMath*, предоставляющая оптимизированные типы (*XMVECTOR, XMMATRIX*) и функции, задействующие *SIMD*-инструкции процессора.

1.4. Основные техники повышения реалистичности изображения

1.4.1. Текстурирование и карты нормалей

Текстурирование – это фундаментальная техника наложения двумерных изображений на трёхмерную геометрию для придания ей

детализации без увеличения полигональной сетки. Координаты текстуры (*UV*) задаются для каждой вершины и интерполируются по поверхности примитива. Для управления процессом выборки цвета используются объекты *SamplerState*, определяющие тип фильтрации (линейная, по точкам), режим адресации (повтор, закрепление) и уровень анизотропной фильтрации. Для оптимизации используются *MIP*-уровни – предварительно рассчитанные уменьшенные копии текстуры.

Особый вид текстур – карта нормалей. Карта нормалей – это текстура, в которой вместо данных о цвете (*RGB*) в каждом пикселе текстуры (текселе) хранится сжатая x-, y- и z-координата в красной, зелёной и синей компонентах соответственно. Эти координаты определяют вектор нормали. Таким образом, карта нормалей хранит вектор нормали для каждого пикселя [5]. Это позволяет имитировать мелкий рельеф поверхности, изменяя нормаль, используемую в расчётах освещения, без модификации геометрии.

Для создания бесконечно удалённого фона (*skybox*) вместо обычных *2D*-текстур используются кубические текстуры (*cubemap*). Кубическую текстуру можно представить как коллекцию из шести отдельных квадратных изображений (граней), соответствующих направлениям в трёхмерном пространстве (+X, -X, +Y, -Y, +Z, -Z). Собранные вместе, они формируют замкнутый куб, создающий иллюзию полного окружающего пространства [6].

1.4.2. Модель освещения по Фонгу

Классическая модель Фонга эмпирически аппроксимирует освещение поверхности как сумму трёх компонент [7]:

1) фоновое (*Ambient*) освещение: моделирует рассеянный свет, многократно отражённый от всех поверхностей в сцене – это «свечение», существующее в освещённой среде из-за бесчисленных взаимодействий

света между поверхностями;

2) диффузное (*Diffuse*) освещение: описывает матовые поверхности, равномерно рассеивающие свет – его интенсивность зависит от косинуса угла между нормалью к поверхности и направлением на источник света (закон Ламберта);

3) зеркальное (*Specular*) освещение: создаёт блики на глянцевых поверхностях, интенсивность зависит от угла между направлением взгляда и идеальным отражением светового луча, а резкость блика контролируется коэффициентом блеска.

1.5. Основные Методы оптимизации рендеринга

1.5.1. Управление порядком отрисовки: буфер глубины

Для автоматического определения видимых поверхностей в *DirectX 11* используется алгоритм *Z*-буферизации. Буфер глубины (*Depth Buffer*) – это специальная текстура, хранящая для каждого пикселя значение его удалённости от камеры. При отрисовке нового фрагмента его глубина сравнивается с хранимой в буфере, и фрагмент отбрасывается, если он находится дальше (при стандартной функции сравнения *D3D11_COMPARISON_LESS*). Это решает проблему корректного порядка отрисовки непрозрачных объектов. Отрисовка прозрачных объектов требует активации смешивания (*Blending*) и правильного порядка. Полупрозрачная геометрия должна выводиться после непрозрачной, отсортированной от дальних объектов к ближним (*back-to-front*), с отключённой записью в буфер глубины, чтобы не мешать корректному смешиванию цветов. Настройки смешивания задаются объектом *ID3D11BlendState*.

Однако на больших расстояниях из-за нелинейного распределения глубины и ограниченной точности форматов с фиксированной запятой (например, *D24_UNORM_S8_UINT*) может возникать артефакт *z-fighting*, когда глубины двух поверхностей становятся неразличимыми и происходит

мерцание. Решением является использование формата с плавающей запятой (*DXGI_FORMAT_D32_FLOAT*) и техники обратной глубины (*Reversed Depth*). При этом в матрице проекции ближняя и дальняя плоскости меняются местами, а функция сравнения меняется на *D3D11_COMPARISON_GREATER*. Это обеспечивает более равномерное распределение точности во всём диапазоне глубин.

1.5.2. Управление порядком отрисовки: работа с прозрачностью

Отрисовка полупрозрачных объектов требует активации смешивания (*blending*). В отличие от непрозрачных геометрий, цвет прозрачного фрагмента должен комбинироваться с цветом, уже находящимся в цели рендеринга. Простейшая модель альфа-смешивания задаётся формулой (2):

$$\text{FinalColor} = \text{SrcAlpha} \cdot \text{SrcColor} + (1 - \text{SrcAlpha}) \cdot \text{DestColor}. \quad (2)$$

Для корректного физического результата прозрачные объекты должны отрисовываться после всех непрозрачных и в порядке от дальних к ближним (*back-to-front*). Часто для них также отключается запись в буфер глубины, чтобы они не мешали корректному смешиванию друг с другом. Все параметры смешивания настраиваются через объект состояния *ID3D11BlendState*.

1.5.3. Инстансинг (*instancing*) как метод снижения нагрузки на *CPU*

Инстансинг – это ключевая техника оптимизации, предназначенная для эффективного рендеринга множества геометрически идентичных объектов (например, деревья, элементы травы, снаряды). Вместо множества отдельных вызовов отрисовки (*DrawIndexed*) используется один вызов *DrawIndexedInstanced*. Данные, уникальные для каждого экземпляра (матрица модели, цвет, индекс текстуры), упаковываются в отдельный буфер (например, структурированный массив в константном буфере). В шейдере доступ к этим данным осуществляется через системное значение *SV_InstanceID*, которое автоматически передаёт индекс текущего

экземпляра. Этот подход резко сокращает нагрузку на центральный процессор, связанную с подготовкой и отправкой команд отрисовки, перенося основную работу на *GPU*.

1.5.4. Отсечение невидимой геометрии (*culling*)

Отсечение по пирамиде видимости (*Frustum Culling*) – это процесс предварительного определения объектов, которые гарантированно не попадают в поле зрения камеры, и их исключение из пайплайна рендеринга. Наиболее распространённый подход на *CPU* предполагает проверку пересечения ограничивающего параллелепипеда объекта, выровненного по осям (*Axis-Aligned Bounding Box, AABB*), с шестью плоскостями пирамиды видимости. Эти плоскости могут быть выведены из комбинированной матрицы вида-проекции. Если *AABB* объекта полностью находится по отрицательную сторону от любой из плоскостей, объект считается невидимым. Более сложные методы могут выполняться на *GPU* с использованием вычислительных шейдеров, что особенно эффективно в сочетании с инстансингом для обработки тысяч объектов.

1.5.5. Многоэтапный рендеринг и постобработка (*Post-Processing*)

Многие современные графические эффекты требуют рендеринга в несколько проходов. Для этого используется техника работы с промежуточными целями отрисовки (*Render Targets*). Приложение создаёт текстуру с флагами *D3D11_BIND_RENDER_TARGET* и *D3D11_BIND_SHADER_RESOURCE*, что позволяет сначала отрендерить в неё сцену, а затем использовать полученное изображение как входную текстуру для следующего шага.

Классическим применением является постобработка – применение экранных фильтров к итоговому кадру. Реализуется через полноэкранный пасс (*fullscreen pass*): отрисовка простейшего примитива (прямоугольника или треугольника), покрывающего весь экран, с пиксельным шейдером,

который для каждого пикселя «семплирует» промежуточную текстуру и применяет к цвету нужные преобразования (тонирование, размытие, добавление шума, эффект *bloom* и т.д.). Организация полноэкранного пасса является стандартным и высокоэффективным паттерном в графическом программировании.

ГЛАВА 2. МЕТОДОЛОГИЯ РАЗРАБОТКИ И АРХИТЕКТУРА ГРАФИЧЕСКОГО ПРИЛОЖЕНИЯ

2.1. Постановка задачи и выбор инструментов

Практической целью данной работы являлась последовательная разработка функционального графического приложения, демонстрирующего ключевые техники и принципы рендеринга, изложенные в теоретической главе. Задачи были конкретизированы в виде восьми поэтапных заданий:

1) создать оконное графическое приложение (проинициализировать *DirectX*, заливать всё фиксированным цветом на каждом кадре, поддержать изменение размера окна);

2) добавить рисование треугольника в *NDC* в графическое приложение;

3) вывести кубик в трехмерном пространстве и добавить управление камерой;

4) добавить *DDS* текстуру на кубик и *skybox* с *cubemap* текстурой;

5) добавить второй кубик или сферу к сцене, поддержать буфер глубины формата *D32_FLOAT* и *reversed depth*, добавить прозрачные объекты с правильной сортировкой и без записи в буфер глубины;

6) добавить в приложение точечные источники света и материал с картой нормалей;

7) добавить в проект *instancing* для вывода непрозрачных кубиков с разными текстурами и использованием *frustum culling*;

8) добавить фильтр постпроцессинга: перевод в оттенки серого, изменение яркости или контраста, сепию или что-то еще.

Каждое из этих заданий вводило новый концептуальный слой сложности: от инициализации графического конвейера и отрисовки простейшего треугольника до реализации оптимизированного рендеринга множества объектов с динамическим освещением и постобработкой. Такой инкрементальный подход позволил обеспечить глубокое усвоение материала

и создать устойчивую, расширяемую кодовую базу.

Выбор *Microsoft DirectX 11* в качестве целевого графического *API* был обусловлен несколькими ключевыми факторами. Во-первых, *DirectX 11* представляет собой «золотую середину» между высокоуровневым управлением, характерным для более ранних версий API, и необходимостью понимания фундаментальных принципов работы графического процессора. Во-вторых, его модель программирования с чётким разделением между устройством (*ID3D11Device*) и контекстом (*ID3D11DeviceContext*) формирует ясную архитектурную парадигму, которая является основой для понимания ещё более низкоуровневых *API*, где управление памятью и синхронизацией осуществляется вручную. В-третьих, *DirectX 11* обладает развитым отладочным слоем, что критически важно для учебного процесса.

Технический стек разработки включал язык *C++* как основной язык реализации логики приложения и язык шейдеров *HLSL*. Среда разработки *Microsoft Visual Studio* обеспечивала необходимые инструменты для компиляции, компоновки и, что особенно важно, интеграции с отладочным слоем *DirectX*. Структура проекта была организована вокруг единственного исходного файла на начальных этапах для наглядности, с последующей логической группировкой функционала (инициализация, рендеринг, управление ресурсами) внутри отдельных функций и структур данных.

2.2. Архитектура разработанного приложения

Архитектура приложения сформирована в соответствии с классической моделью оконного приложения *Windows* и стандартным графическим конвейером *DirectX 11*. Её можно представить в виде следующих ключевых модулей, взаимодействующих в рамках главного цикла приложения (табл. 1).

Таблица 1

Основные модули архитектуры графического приложения

Модуль	Основные функции/компоненты	Назначение
Управление окном и сообщениями	*WinMain*, *InitWindow*, *WndProc*	Создание и обслуживание окна, обработка пользовательского ввода (мышь, клавиатура, изменение размера).
Инициализация графического подсистемы	*InitDirectX*, *SetupBackBuffer*	Создание устройств *Direct3D* и *DXGI*, настройка цепочки обмена (*Swap Chain*), создание целей рендеринга и буфера глубины.
Менеджмент ресурсов	*InitCube*, *InitSkybox*, *LoadTexture*, *InitBuffers*	Централизованное создание и конфигурация всех графических ресурсов: буферов вершин/индексов, текстур, шейдеров, константных буферов, состояний конвейера.
Игровой цикл и рендеринг	Цикл сообщений, *UpdateCamera*, *Render*	Обновление состояния сцены (камера, анимация) и непосредственное выполнение команд рендеринга через графический конвейер.
Подсистема отладки и *UI*	Отладочный слой *DirectX 11*, библиотека *ImGui*	Валидация вызовов *API*, отслеживание утечек ресурсов, предоставление интерактивного интерфейса для управления параметрами сцены.

В основе реализации лежали принципы модульности и чёткого разделения ответственности. Фаза инициализации (*Init** функции) строго отделена от фазы исполнения (*Update*, *Render*). Все ресурсы создаются один

раз при старте приложения и освобождаются в функции *Cleanup* в порядке, обратном созданию, что предотвращает утечки. Состояние конвейера (привязанные шейдеры, буферы, ресурсы) явно настраивается каждый кадр в функции *Render*, что обеспечивает предсказуемость и упрощает отладку.

Ключевым паттерном, использованным в проекте, стало разделение данных и кода через специализированные структуры для константных буферов (например, *GeomBuffer*, *SceneBuffer*). Это не только отражает теоретический принцип группировки данных по частоте обновления, но и напрямую транслируется в организацию шейдерных констант, обеспечивая типобезопасность и удобство при модификации. Обработка ошибок реализована через повсеместную проверку возвращаемых значений *HRESULT* после вызовов *DirectX API* и использование макроса *SAFE_RELEASE* для безопасного освобождения *COM*-объектов.

2.3. Методы тестирования и отладки

Разработка велась с постоянным активированным отладочным слоем *DirectX 11* (*D3D11_CREATE_DEVICE_DEBUG*). Этот слой выступал в роли основного инструмента валидации, выполняя следующие функции:

1) проверка параметров вызовов *API*: немедленное предупреждение о передаче некорректных указателей, несовместимых флагов или нарушении контрактов интерфейсов;

2) отслеживание утечек ресурсов: при завершении работы приложения отладочный слой выводил в консоль *Visual Studio* список всех неосвобождённых *COM*-объектов *DirectX*, что позволяло оперативно находить и исправлять ошибки управления памятью;

3) предупреждения о потенциально неоптимальных действиях: например, создание ресурсов с неидеальными для производительности флагами или избыточные изменения состояния конвейера.

Визуальная верификация результатов каждого из восьми этапов была

вторым ключевым методом тестирования. Корректность реализации подтверждалась непосредственным наблюдением ожидаемого результата: появление цветного треугольника, вращение текстурированного куба под управлением камеры, корректное наложение освещения и карт нормалей, визуальный эффект от включения различных режимов постобработки. Интеграция библиотеки *ImGui* на поздних этапах проекта предоставила мощный интерактивный инструмент для динамической настройки параметров сцены (позиция источников света, выбор эффекта постобработки, управление инстансингом), что также способствовало глубокому тестированию и тонкой настройке функционала.

Такой комплексный подход к отладке, сочетающий автоматизированную валидацию *API* и интерактивный визуальный контроль, обеспечил высокую надёжность и корректность разработанного приложения.

ГЛАВА 3. ПРАКТИЧЕСКАЯ РАЗРАБОТКА: ЭТАПЫ, РЕАЛИЗАЦИЯ И РЕЗУЛЬТАТЫ

Практическая часть исследования представляет собой последовательную реализацию графического приложения на *DirectX 11*, где каждый этап соответствует решению конкретной задачи и наращиванию функциональности. Весь код реализованных этапов представлен в репозитории [8]. Ниже проводится детальный анализ каждого этапа разработки, включая ключевые решения, особенности реализации и полученные результаты.

3.1. Инициализация графического конвейера и вывод примитивного изображения

Начальный этап разработки был посвящен созданию стабильной основы приложения с корректно инициализированным графическим *API*. Этот этап заложил фундамент для всех последующих модификаций, обеспечив базовое взаимодействие с оконной системой *Windows* и графическим конвейером *DirectX 11*.

Инициализация начинается с функции *WinMain*, которая является точкой входа в приложение и координирует все этапы запуска: создание окна, настройку *DirectX*, загрузку ресурсов и запуск главного цикла сообщений. Функция *InitWindow* отвечает за регистрацию класса окна и его создание с использованием *API Windows*, с правильным расчетом размеров клиентской области через *AdjustWindowRect*.

Центральное место в инициализации занимает функция *InitDirectX*, которая создает все основные объекты *DirectX 11*. Ключевыми из них являются устройство (*ID3D11Device*), выступающее в роли фабрики ресурсов, и контекст устройства (*ID3D11DeviceContext*), управляющий состоянием конвейера и исполняющий команды рисования. Для связи с оконной системой создается цепочка обмена (*IDXGISwapChain*), реализующая двойную буферизацию с современным алгоритмом смены

буферов *DXGI_SWAP_EFFECT_FLIP_DISCARD*.

Особое внимание было уделено активации отладочного слоя *DirectX 11* через флаг *D3D11_CREATE_DEVICE_DEBUG*. Этот инструмент обеспечивает валидацию вызовов *API*, отслеживание утечек ресурсов и вывод диагностических сообщений, что значительно упрощает процесс отладки на ранних этапах разработки.

Функция *SetupBackBuffer* создает представления ресурсов для *back buffer* и буфера глубины. *Back buffer* получается из цепочки обмена, после чего для него создается *Render Target View* (*ID3D11RenderTargetView*), определяющий, как буфер будет использоваться в качестве цели рендеринга. Обработка оконных сообщений в функции *WndProc* включает реакцию на изменение размера окна, что приводит к пересозданию буферов через *ResizeSwapChain* для сохранения корректного отображения.

На этом этапе графический конвейер выполняет минимальные действия: очистка *back buffer* заданным цветом и вывод через *Present*. Хотя визуальный результат представляет собой просто залитый цветом экран, технически это означает полную готовность конвейера к расширению функционала.

Перечень ключевых функций представлен в табл. 2:

Таблица 2

Ключевые функции инициализации графического конвейера

Функция	Назначение	Создаваемые объекты
WinMain	Точка входа, координация запуска	Нет объектов
InitWindow	Создание и регистрация окна	Класс окна, экземпляр окна
InitDirectX	Инициализация графического *API*	Устройство, контекст, цепочка обмена
SetupBackBuffer	Настройка целей рендеринга	*Render Target View*, буфер глубины
WndProc	Обработка сообщений окна	Нет объектов

Результат данного этапа представлен на рис. 1. Получено стабильное оконное приложение с полным циклом обработки сообщений,

инициализированными основными объектами *DirectX 11* и готовое к добавлению геометрии и шейдеров.

Рис. 1. Результат первого этапа – оконное приложение

3.2. Ввод геометрии: отрисовка треугольника

Второй этап разработки ввел ключевые концепции работы с геометрией в *DirectX 11* – создание и отрисовку треугольника. Этот этап продемонстрировал работу с вершинными и индексными буферами, шейдерами и входным макетом, которые являются фундаментальными для любого графического приложения.

Основным нововведением стала функция *InitTriangle*, ответственная за создание всех ресурсов для отрисовки треугольника. Была определена структура вершины, содержащая позицию в трехмерном пространстве и цвет. Особенностью реализации стало использование типа *COLORREF*, стандартного для *Windows*, но требующего специальной обработки в контексте *DirectX* из-за различий в форматах хранения цвета (*BGRA* вместо *RGBA*).

Вершины треугольника были определены в нормализованных координатах устройства (*NDC*), что позволяет отображать геометрию без дополнительных матричных преобразований на начальном этапе. Для

создания буфера вершин использовалась структура *D3D11_BUFFER_DESC* с флагом *D3D11_BIND_VERTEX_BUFFER* и режимом использования *D3D11_USAGE_IMMUTABLE*, что указывает на неизменяемость данных после создания и позволяет драйверу оптимизировать размещение в памяти *GPU*.

Вершинный и пиксельный шейдеры были реализованы на языке *HLSL* и скомпилированы во время выполнения. Вершинный шейдер выполняет простое преобразование координат, а пиксельный шейдер возвращает цвет вершины, который затем интерполируется по поверхности треугольника. Ключевым моментом стало создание входного макета (*ID3D11InputLayout*) через функцию *CreateInputLayout*, который описывает соответствие между структурой вершин в буфере и входными параметрами вершинного шейдера.

В функции рендеринга были добавлены вызовы для настройки *Input Assembler* – этапа графического конвейера, ответственного за сборку примитивов из вершинных данных. Методы *IASetVertexBuffers*, *IASetIndexBuffer* и *IASetInputLayout* устанавливают соответствующие ресурсы, а *IASetPrimitiveTopology* определяет тип примитивов. Отрисовка осуществляется через *DrawIndexed*, который использует индексный буфер для определения порядка соединения вершин.

Перечень ключевых компонентов этапа представлен в табл. 3:

Таблица 3

Компоненты системы отрисовки треугольника

Компонент	Тип	Назначение
Структура *Vertex*	Структура данных	Хранение позиции и цвета вершины
ID3D11Buffer (вершинный)	Ресурс *GPU*	Хранение данных вершин треугольника
ID3D11Buffer (индексный)	Ресурс *GPU*	Определение порядка соединения вершин
ID3D11InputLayout	Объект состояния	Связь формата вершины с шейдером
ID3D11VertexShader	Шейдер	Преобразование координат вершин
ID3D11PixelShader	Шейдер	Определение цвета пикселей

Результат данного этапа представлен на рис. 2. На экране отображается треугольник с плавным интерполированным цветом от красного к зеленому и синему. Этот этап продемонстрировал полный цикл работы с геометрией в *DirectX 11*.

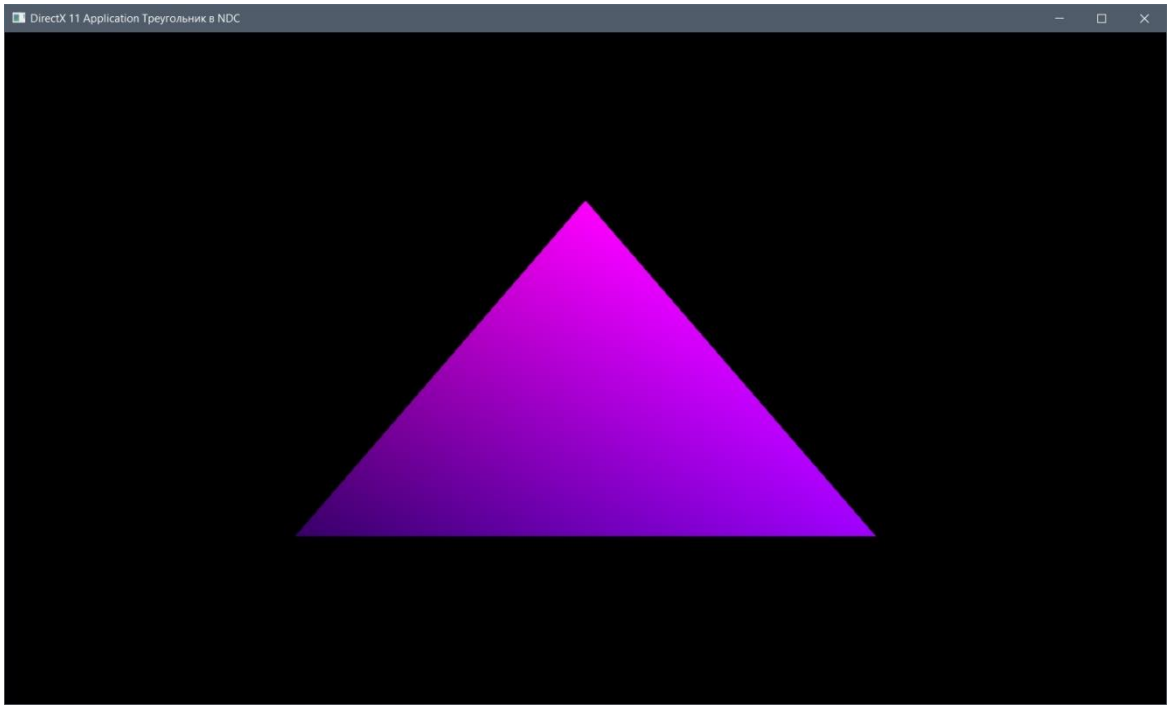


Рис. 2. Результат второго этапа – треугольник

3.3. Трехмерная сцена: куб, камера и матричные преобразования

Третий этап разработки знаменует переход от двухмерной графики к полноценной трехмерной сцене. Основными задачами стали: создание трехмерной геометрии куба, реализация матричных преобразований и внедрение управляемой камеры с перспективной проекцией.

Геометрия куба была определена как набор из 24 вершин (по 4 на каждую грань) и 36 индексов (по 12 треугольников, по 2 на каждую грань). Структура вершины осталась аналогичной предыдущему этапу, но позиции были заданы в трехмерном пространстве для формирования объемного объекта.

Ключевым нововведением стало внедрение системы матричных преобразований для перевода вершин из локальной системы координат объекта в экранные координаты. В *DirectX 11* используется иерархия

преобразований: локальные координаты → мировые координаты → координаты вида → координаты отсечения. Эти преобразования реализуются через умножение вершин на соответствующие матрицы.

Для передачи матриц в шейдеры были созданы константные буферы – специальные области памяти *GPU*, оптимизированные для частого чтения шейдерами. Были созданы два буфера: для матрицы модели и для комбинированной матрицы вида-проекции. Матрица модели отвечает за положение, вращение и масштаб объекта в мировом пространстве, а матрица вида-проекции является результатом умножения матрицы вида и матрицы проекции.

Управление камерой было реализовано через систему сферических координат, что позволяет удобно вращать камеру вокруг объекта. Функция *UpdateCamera* вычисляет позицию камеры на основе расстояния и углов, после чего создает матрицу вида с помощью *XMMatrixLookAtLH*. Матрица проекции создается с помощью *XMMatrixPerspectiveFovLH*, которая учитывает угол обзора, соотношение сторон окна и ближнюю с дальнюю плоскости отсечения.

Вершинный шейдер был модифицирован для выполнения матричных преобразований. Каждая вершина последовательно умножается на матрицу модели, затем на матрицу вида-проекции. Важным дополнением стала реализация буфера глубины для корректного определения видимых поверхностей. Буфер глубины создается как текстура и привязывается к конвейеру как *Depth Stencil View*.

Система матричных преобразований представлена в табл. 4:

Таблица 4

Система матричных преобразований и камеры

Компонент	Назначение	Реализация
Константный буфер *GeomBuffer*	Хранение матрицы модели	Структура с матрицей 4×4
Константный буфер *SceneBuffer*	Хранение матрицы вида-проекции	Структура с матрицей 4×4
Функция *UpdateCamera*	Обновление матриц камеры	Расчет на основе сферических координат
Матрица вида	Преобразование в систему координат камеры	*XMMatrixLookAtLH*
Матрица проекции	Перспективное преобразование	*XMMatrixPerspectiveFovLH*
Буфер глубины	Определение видимых поверхностей	Текстура формата *D24_UNORM_S8_UINT*

Управление камерой было интегрировано с системой обработки сообщений: вращение камеры осуществляется при зажатой правой кнопке мыши, а прокрутка колесика мыши изменяет расстояние до объекта. Это обеспечивает интерактивное взаимодействие с трехмерной сценой.

Результат данного этапа представлен на рис. 3. На экране отображается трехмерный куб с перспективными искажениями, который можно вращать с помощью мыши. Цвета граней куба интерполируются между вершинами, создавая плавные градиенты.

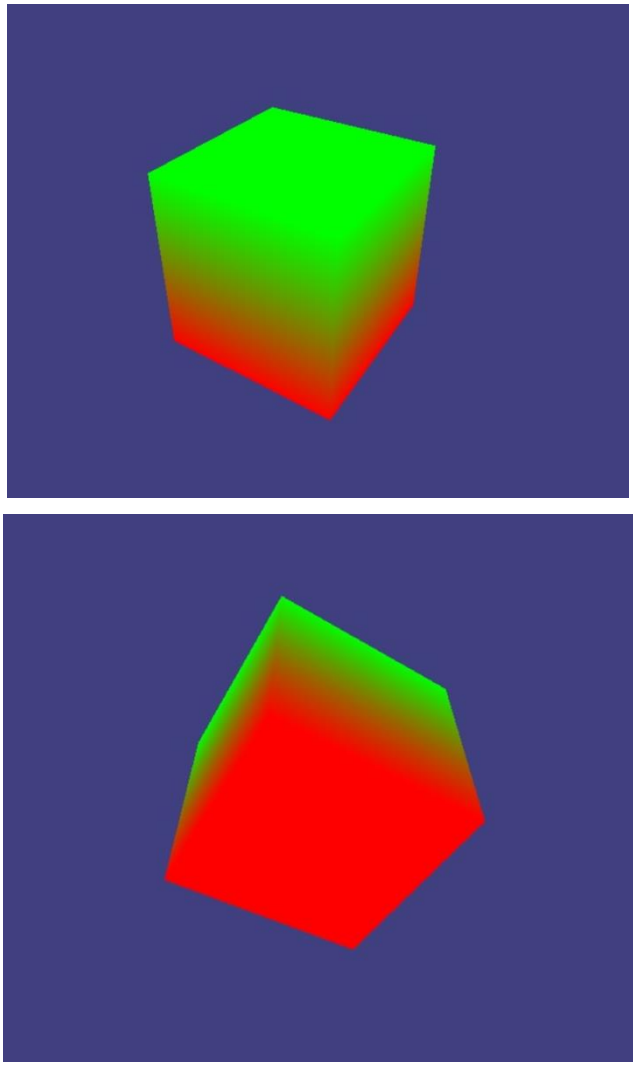


Рис. 3. Результат третьего этапа – *3D*-кубик и камера

3.4. Текстурирование и окружение *Skybox*

Четвертый этап разработки был направлен на повышение реалистичности сцены за счет внедрения текстур и создания бесконечного фона (*skybox*). Основными задачами стали: реализация системы текстурирования с поддержкой аппаратного сжатия, создание механизма загрузки текстур и разработка *skybox* на основе кубических текстур.

Ключевым изменением в структуре данных стало модифицирование вершины для хранения текстурных координат (*UV*-координат). Новая структура содержит позицию в пространстве и две координаты текстуры, что позволяет сопоставлять точки на поверхности геометрии с точками на текстуре.

Текстуры загружаются из файлов формата *DDS* (*DirectDraw Surface*) с помощью функции *LoadDDS*, которая анализирует заголовок файла, определяет формат сжатия и читает данные текстуры вместе с *MIP*-уровнями. Формат *DDS* был выбран благодаря его прямой совместимости с аппаратным обеспечением *GPU* – текстуры в этом формате могут быть напрямую загружены в видеопамять без дополнительной обработки.

Для работы со сжатыми текстурами были реализованы вспомогательные функции для определения размера сжатого блока и выполнения деления с округлением вверх. Форматы семейства *BC* (*Block Compression*) используют блочное сжатие с фиксированным размером блока 4×4 пикселя.

Процесс загрузки текстуры включает несколько этапов: чтение файла *DDS*, создание ресурса текстуры, создание представления для шейдера и настройка семплера. Семплер определяет параметры фильтрации и адресации текстуры, такие как тип фильтра, режим наложения и уровень анизотропной фильтрации.

Вершинный шейдер был модифицирован для передачи текстурных координат в пиксельный шейдер, где происходит выборка цвета из текстуры. Пиксельный шейдер заменяет вершинные цвета на цвета из текстуры, что позволяет накладывать детализированные изображения на геометрию без увеличения полигональной сетки.

Отдельным значимым достижением стала реализация *skybox* – бесконечного фона, создающего иллюзию окружающего пространства. *Skybox* реализован как сфера с наложенной кубической текстурой (*cubemap*). Кубическая текстура состоит из шести отдельных граней, соответствующих направлениям в трехмерном пространстве.

Геометрия сферы генерируется алгоритмически через параметризацию по широте и долготе. Для *skybox* был создан отдельный набор шейдеров и отдельное состояние глубины с функцией сравнения *D3D11_COMPARISON_LESS_EQUAL*, что позволяет *skybox* отображаться

позади всех объектов, но не влиять на буфер глубины.

Компоненты системы текстурирования представлены в табл. 5:

Таблица 5

Компоненты системы текстурирования

Компонент	Назначение	Особенности
Структура *TextureVertex*	Хранение позиции и *UV*-координат	Добавлены поля *u*, *v* для текстурных координат
Функция *LoadDDS*	Загрузка текстур из файлов	Поддержка форматов *DXT/BC*, *MIP*-уровни
ID3D11ShaderResourceView	Представление текстуры для шейдера	Доступ к текстуре из шейдеров
ID3D11SamplerState	Настройка параметров выборки	Фильтрация, адресация, анизотропия
Кубическая текстура	Основа для *skybox*	6 граней (+X, -X, +Y, -Y, +Z, -Z)
Skybox шейдеры	Специальные шейдеры для фона	Отдельный вершинный и пиксельный шейдеры

Особенностью рендеринга *skybox* является его положение относительно камеры – *skybox* всегда центрирован на камере и движется вместе с ней, создавая иллюзию бесконечно удаленного фона. Порядок рендеринга был оптимизирован: сначала отрисовывается *skybox*, затем непрозрачные объекты.

Результат данного этапа представлен на рис. 4 и 5. На экране отображается текстурированный куб, помещенный в реалистичное окружение *skybox*. Текстура куба демонстрирует детализацию поверхности, а *skybox* создает ощущение трехмерного пространства.

Рис. 4. Результат четвёртого этапа – текстуры по заданию

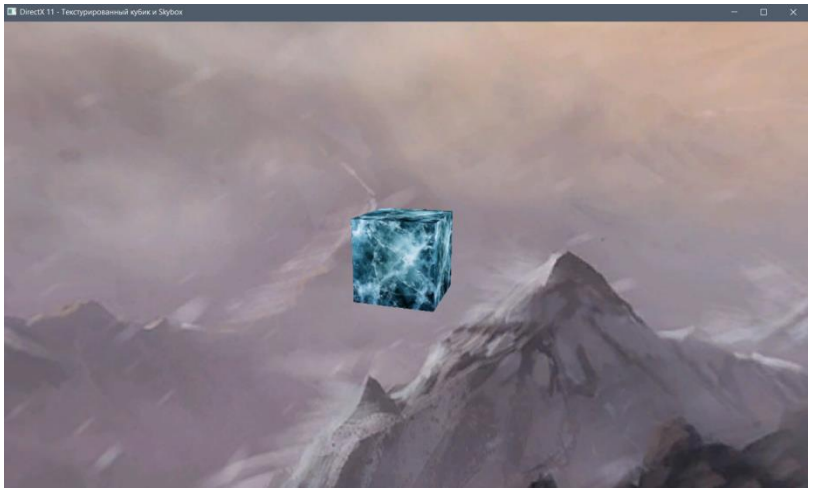


Рис. 5. Результат четвёртого этапа – мои текстуры

3.5. Управление глубиной и рендеринг прозрачных объектов

Пятый этап разработки был посвящен решению проблем корректного отображения сложных сцен с несколькими объектами и различными типами прозрачности. Основными задачами стали: устранение артефактов *z-fighting*, внедрение обратной глубины и реализация системы рендеринга прозрачных объектов с альфа-смешиванием.

Проблема *z-fighting* возникает, когда два объекта находятся на очень близком расстоянии друг от друга, и из-за ограниченной точности буфера глубины их глубины становятся неразличимыми, вызывая мерцание пикселей. Для решения этой проблемы был внедрен формат буфера глубины с плавающей запятой, который обеспечивает более высокую точность, особенно на больших расстояниях.

Вторым ключевым решением стало использование техники обратной глубины (*reversed depth*). В традиционном подходе ближняя плоскость отсечения соответствует значению 0.0, а дальняя – 1.0, что приводит к нелинейному распределению точности. При обратной глубине это соотношение меняется на противоположное, что обеспечивает более равномерное распределение точности по всему диапазону глубин.

Реализация обратной глубины потребовала изменений в нескольких компонентах системы. Матрица проекции теперь создается с обратным порядком параметров *near* и *far*. Функция сравнения глубины была изменена для непрозрачных объектов и для *skybox*. Очистка буфера глубины теперь выполняется нулевым значением вместо единичного.

Для поддержки прозрачных объектов была создана система смешивания (*blending*) на основе объекта состояния *ID3D11BlendState*. Смешивание позволяет комбинировать цвет текущего фрагмента с цветом, уже находящимся в цели рендеринга, согласно заданной формуле. Для альфа-смешивания используется формула, которая учитывает альфа-канал источника для определения пропорции смешивания.

Для корректного отображения прозрачных объектов критически важен порядок их отрисовки. Прозрачные объекты должны рендериться после всех непрозрачных и в порядке от дальних к ближним. Это обеспечивает физически корректное наложение полупрозрачных поверхностей. В реализации была добавлена сортировка прозрачных объектов на основе их расстояния до камеры.

Сцена была расширена добавлением второго куба и двух прозрачных прямоугольников. Для каждого типа объектов были созданы отдельные состояния глубины: с записью глубины для непрозрачных объектов, без записи для прозрачных, и специальное состояние для *skybox*. Это позволяет оптимизировать процесс рендеринга и избежать артефактов.

Skybox был адаптирован для работы с обратной глубиной через принудительное выставление координаты *Z* в ноль в вершинном шейдере и

использование соответствующей функции сравнения. Это гарантирует, что *skybox* будет отображаться позади всех объектов, но не будет влиять на буфер глубины.

Система управления из данного этапа представлена в табл. 6:

Таблица 6

Система управления глубиной и прозрачностью

Техника	Проблема	Решение
Формат *D32_FLOAT*	Ограниченная точность фиксированной запятой	Использование формата с плавающей запятой
Обратная глубина	Нелинейное распределение точности	Инвертирование *near/far* плоскостей
Раздельные состояния глубины	Конфликты между типами объектов	Отдельные состояния для непрозрачных, прозрачных объектов и *skybox*
Альфа-смешивание	Некорректное наложение прозрачностей	Формула с учетом альфа-канала источника
Сортировка *back-to-front*	Некорректный порядок отрисовки прозрачных объектов	Сортировка по расстоянию до камеры

Порядок рендеринга в функции *Render* был организован следующим образом: сначала непрозрачные объекты, затем *skybox*, и наконец прозрачные объекты с сортировкой. Такой порядок минимизирует *overdraw* и обеспечивает корректное смешивание.

Результат данного этапа представлен на рис. 6. На экране отображается сцена с двумя непрозрачными кубами, *skybox* и двумя прозрачными прямоугольниками с разной степенью прозрачности. Отсутствие артефактов *z-fighting* и корректное наложение прозрачных объектов демонстрируют эффективность примененных решений.

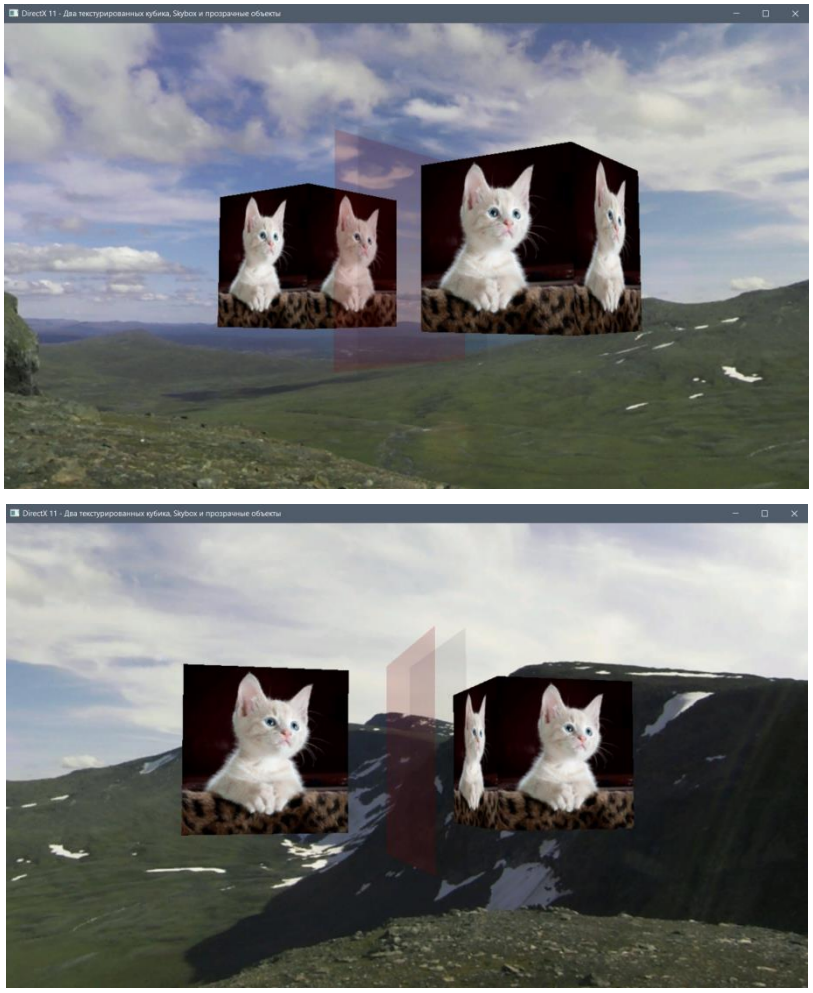


Рис. 6. Результат пятого этапа – несколько видов объектов

3.6. Динамическое освещение по модели Фонга с использованием карт нормалей

Шестой этап разработки был направлен на создание реалистичной системы освещения, имитирующей взаимодействие света с материалами объектов. Основными задачами стали: реализация модели освещения по Фонгу, внедрение карт нормалей, создание системы управления источниками света и обеспечение интерактивной настройки параметров освещения.

Модель освещения по Фонгу эмпирически аппроксимирует освещение поверхности как сумму трех компонент: фонового (*ambient*), диффузного (*diffuse*) и зеркального (*specular*) освещения. Фоновое освещение моделирует рассеянный свет, диффузное описывает матовые поверхности, а зеркальное создает блики на глянцевых материалах.

Для реализации этой модели структура вершины была расширена до

TextureTangentVertex, которая теперь содержит не только позицию и текстурные координаты, но также нормаль и касательный вектор. Нормали и касательные векторы необходимы для работы с картами нормалей – специальными текстурами, которые хранят информацию о направлении нормали для каждого текселя.

Карты нормалей позволяют имитировать мелкий рельеф поверхности без увеличения полигональной сетки, значительно повышая визуальную детализацию. Для корректного использования карт нормалей необходимо преобразование нормалей из касательного пространства (в котором они хранятся в текстуре) в мировое пространство. Это преобразование выполняется с помощью *TBN*-матрицы, которая строится в пиксельном шейдере на основе интерполированных нормалей и касательных векторов.

Константные буферы были расширены для передачи параметров освещения. Буфер геометрии теперь содержит не только матрицу модели, но и матрицу для преобразования нормалей, а также коэффициент блеска материала. Буфер сцены был значительно расширен и теперь включает информацию об источниках света, фоновом освещении и параметрах сцены.

Была реализована система управления источниками света, поддерживающая до 10 точечных источников. Каждый источник характеризуется позицией и цветом. Для визуализации позиций источников были созданы маленькие сферы, которые отрисовываются в тех же позициях, что и источники света.

Расчет освещения в пиксельном шейдере выполняется в функции *CalculateLighting*, которая для каждого источника света вычисляет диффузную и зеркальную составляющие с учетом квадратичного затухания. Квадратичное затухание моделирует уменьшение интенсивности света с квадратом расстояния, что соответствует физической реальности.

Для интерактивного управления параметрами освещения была интегрирована библиотека *ImGui*, предоставляющая простой в использовании интерфейс для создания панелей управления. Через *ImGui*

реализованы следующие возможности: включение и выключение карт нормалей, переключение режима отображения нормалей, добавление и удаление источников света, настройка цвета и позиции каждого источника, регулировка фонового освещения.

Библиотека *ImGui* была интегрирована в цикл рендеринга: в начале каждого кадра вызывается функция нового кадра, затем создаются окна управления, и в конце отрисовывается интерфейс поверх сцены. Это обеспечивает плавное взаимодействие с параметрами освещения в реальном времени.

Система освещения из данного этапа представлена в табл. 7:

Таблица 7

Компоненты системы освещения

Компонент	Назначение	Реализация
Модель Фонга	Расчет освещения поверхности	*Ambient* + *Diffuse* + *Specular* компоненты
Карты нормалей	Имитация рельефа поверхности	Текстура с нормалями в касательном пространстве
TBN-матрица	Преобразование нормалей	Матрица 3 × 3 из *tangent*, *bitangent*, *normal*
Система источников света	Управление точечными источниками	Массив структур с позицией и цветом
Квадратичное затухание	Физическая модель ослабления света	Интенсивность ~ 1/расстояние²
Интерфейс *ImGui*	Интерактивное управление параметрами	Окна с элементами управления

Результат данного этапа представлен на рис. 7–11. На экране отображается сцена с динамически освещенными объектами, демонстрирующими эффекты диффузного и зеркального отражения. Карты нормалей создают иллюзию рельефа на поверхности кубов. Панель управления *ImGui* позволяет интерактивно настраивать все параметры освещения.

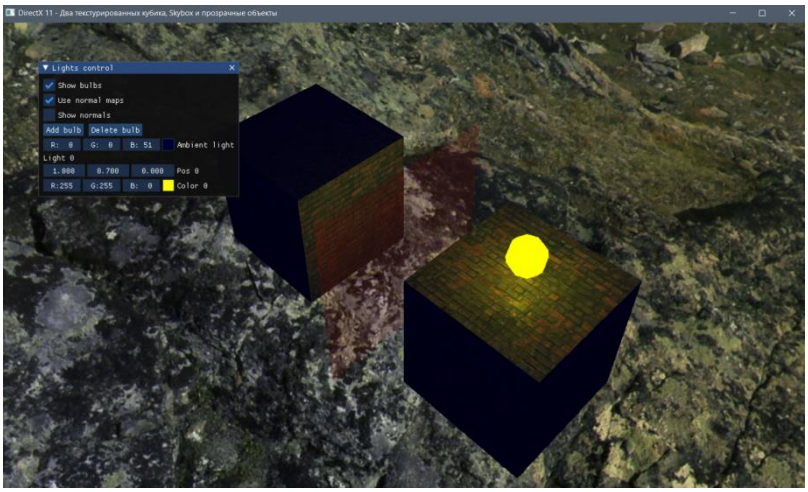


Рис. 7. Результат шестого этапа – освещение

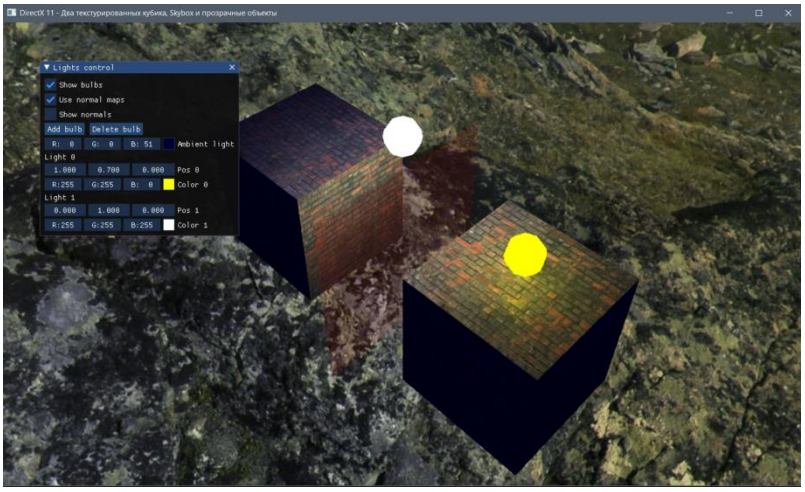


Рис. 8. Результат шестого этапа – добавление источника

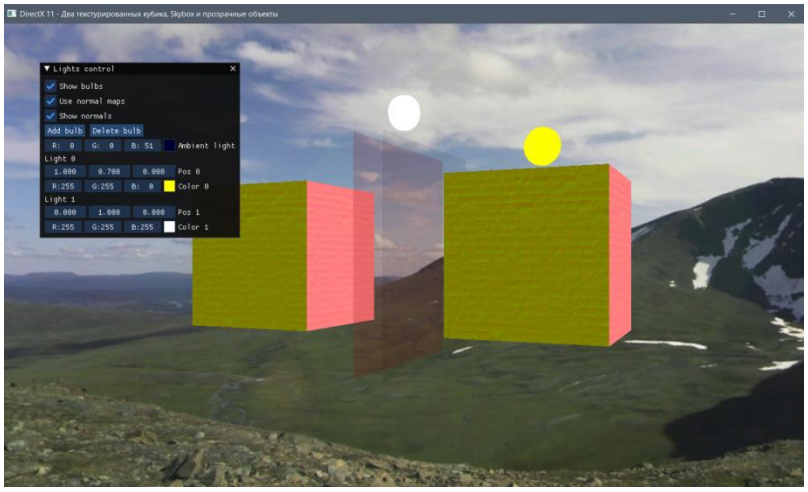


Рис. 9. Результат шестого этапа – нормали

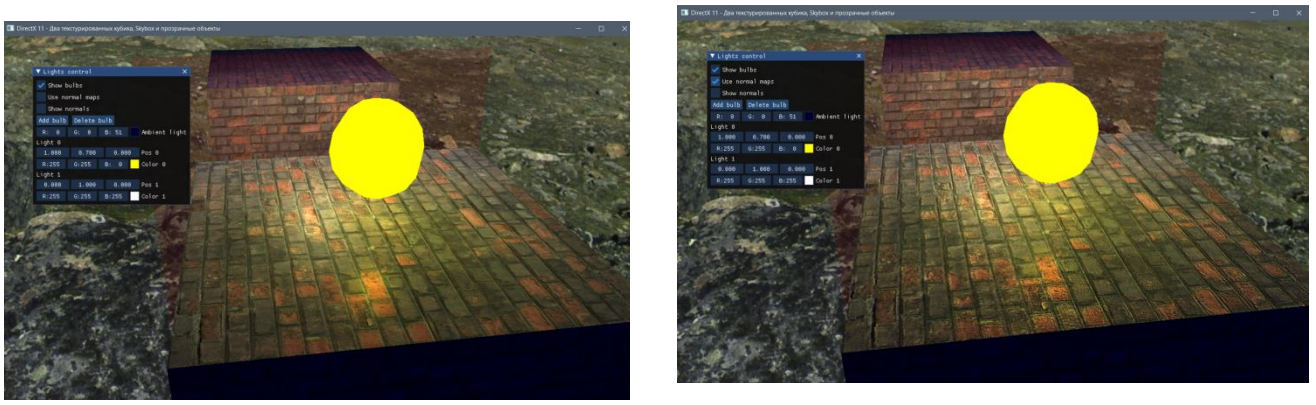


Рис. 10. Результат шестого этапа – освещение без и с картой нормалей

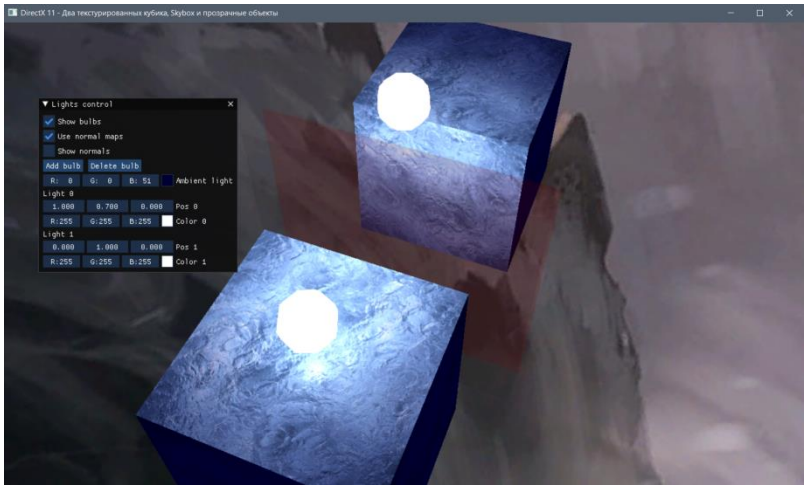


Рис. 11. Результат шестого этапа – мои текстуры (с картой нормалей)

3.7. Оптимизация рендеринга множества объектов: *Instancing* и *Frustum Culling*

Седьмой этап разработки был посвящен решению проблемы эффективного рендеринга большого количества геометрически схожих объектов. Основными задачами стали: внедрение техники инстансинга, реализация отсечения по пирамиде видимости на *CPU* и создание системы управления множеством инстансов.

Инстансинг – это ключевая техника оптимизации в компьютерной графике, позволяющая отрисовывать множество одинаковых объектов за один вызов *GPU*. Вместо множества отдельных вызовов отрисовки используется один вызов *DrawIndexedInstanced*, который автоматически создает указанное количество экземпляров геометрии. Данные, уникальные для каждого экземпляра, упаковываются в отдельный буфер.

Для реализации инстансинга была создана структура, содержащая все параметры инстанса: матрицу модели, матрицу нормалей, параметры материала и позицию. Все инстансы хранятся в едином константном буфере, что позволяет эффективно передавать данные на *GPU*. Вершинный шейдер был модифицирован для работы с инстансингом через системное значение *SV_InstanceID*, которое автоматически передает индекс текущего экземпляра.

Для поддержки разных текстур для разных инстансов была реализована загрузка массива текстур. Массив текстур позволяет хранить несколько текстур в одном ресурсе *GPU*, с возможностью выборки по индексу в шейдере. В данном случае были загружены две текстуры, что позволяет разным инстансам использовать разные материалы.

Отсечение по пирамиде видимости (*frustum culling*) – это техника оптимизации, которая исключает из процесса рендеринга объекты, гарантированно находящиеся вне поля зрения камеры. На *CPU* строится пирамида видимости как набор из шести плоскостей, и для каждого объекта проверяется, находится ли его ограничивающий параллелепипед внутри этой пирамиды.

Функция *CullBoxes* реализует алгоритм *frustum culling*. Сначала строятся шесть плоскостей пирамиды видимости на основе матрицы вида-проекции и параметров камеры. Затем для каждого инстанса проверяется пересечение его *AABB* с пирамидой. Индексы видимых инстансов записываются в отдельный буфер, который передается в шейдер для двухступенчатой индексации.

Для интерактивного управления инстансами была расширена панель *ImGui*. Добавлены кнопки для добавления и удаления инстансов, сброса к исходному состоянию, и чекбокс для включения и выключения *frustum culling*. Также отображается информация о количестве всех инстансов и видимых после отсечения.

При добавлении нового инстанса генерируются случайные параметры: позиция в определенном диапазоне, коэффициент блеска, индекс текстуры и

наличие карты нормалей. Это создает разнообразную сцену для демонстрации возможностей системы.

Методы оптимизации рендеринга множества объектов представлены в табл. 8:

Таблица 8

Методы оптимизации рендеринга множества объектов

Метод	Принцип работы	Эффект
Инстансинг	Один вызов отрисовки для множества объектов	Сокращение вызовов *GPU*, снижение нагрузки на *CPU*
Массив текстур	Несколько текстур в одном ресурсе	Эффективное использование памяти, быстрая смена материалов
Frustum culling	Отсечение невидимых объектов на *CPU*	Снижение нагрузки на графический конвейер
Двухступенчатая индексация	Буфер видимых индексов и буфер данных	Минимизация передачи данных на *GPU*
Управление через *ImGui*	Динамическое добавление и удаление инстансов	Интерактивное тестирование производительности

Результат данного этапа представлен на рис. 12. На экране отображается сцена с множеством кубов (до 100 инстансов), эффективно отрисованных с использованием техники инстансинга. Включение *frustum culling* значительно снижает нагрузку на *GPU*, особенно когда многие объекты находятся вне поля зрения.

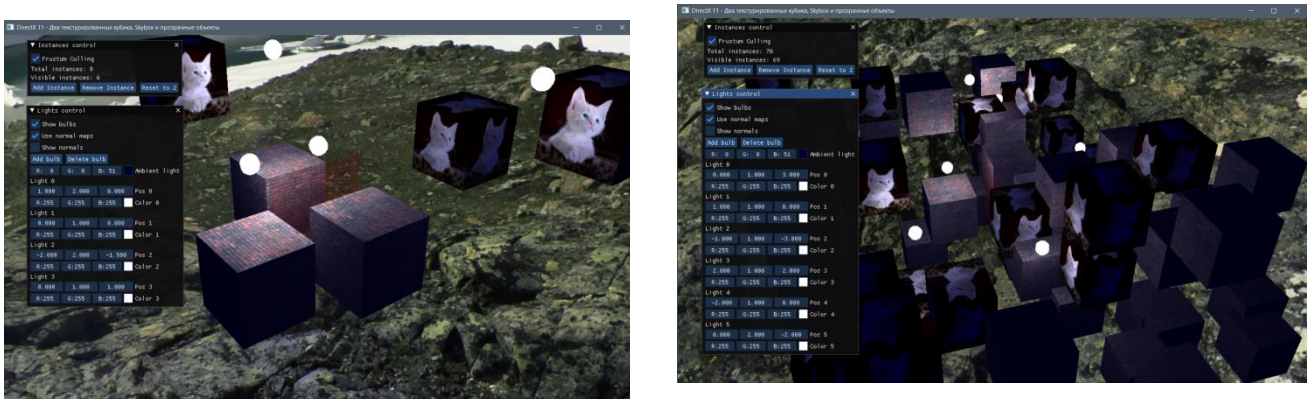


Рис. 12. Результат седьмого этапа – множество кубов

3.8. Постобработка изображения

Восьмой, завершающий этап разработки, был посвящен внедрению системы постобработки – техники применения экранных фильтров к уже

отрендеренному изображению. Основными задачами стали: создание архитектуры многоэтапного рендеринга, реализация нескольких эффектов постобработки, оптимизация полноэкранного рендеринга и интеграция управления эффектами через интерфейс.

Ключевым изменением архитектуры рендеринга стал переход от прямой отрисовки в *back buffer* к двухэтапному процессу. На первом этапе вся сцена рендерится в промежуточную текстуру, которая создается с флагами, позволяющими использовать ее как цель рендеринга и как входной ресурс для шейдера. Для промежуточной текстуры создаются соответствующие представления: *Render Target View* для этапа рендеринга сцены и *Shader Resource View* для этапа постобработки.

Вершинный шейдер постобработки был оптимизирован через использование одного треугольника вместо двух треугольников, образующих квадрат. Треугольник с определенными координатами покрывает весь экран при использовании правильного алгоритма растеризации, но требует только трех вершин вместо шести. Текстурные координаты вычисляются из позиции вершины в *clip space*, что переводит координаты из диапазона [-1, 1] в диапазон [0, 1], пригодный для выборки из текстуры.

Был реализован пиксельный шейдер постобработки, содержащий три эффекта: сепия, холодный тон и ночное видение. Выбор эффекта осуществляется через константный буфер, который содержит поле для типа эффекта. Каждый эффект реализован в отдельной функции. Эффект сепии преобразует изображение в коричневатые тона, имитируя старинные фотографии. Холодный тон усиливает синюю и зеленую составляющие, создавая холодную цветовую гамму. Ночное видение преобразует изображение в зеленоватые тона, имитируя приборы ночного видения.

Константный буфер обновляется каждый кадр через функцию, которая использует механизм отображения памяти для записи данных в буфер с динамическим доступом. Это позволяет изменять эффект постобработки в реальном времени без перекомпиляции шейдеров.

Порядок рендеринга в функции *Render* был реорганизован: сначала сцена рендерится в промежуточную текстуру, затем вызывается функция постобработки, которая применяет выбранный эффект и выводит результат в *back buffer*. Между этими этапами происходит смена целей рендеринга и состояний конвейера.

Для управления эффектами постобработки была расширена панель *ImGui*. Добавлено окно с выпадающим списком для выбора эффекта. Текущий выбранный эффект отображается цветным текстом, что улучшает пользовательский опыт. Архитектура системы постобработки была спроектирована с учетом расширяемости, что соответствует принципам модульности и поддерживаемости кода.

Эффекты постобработки и их характеристики представлены в табл. 9:

Таблица 9

Эффекты постобработки и их характеристики

Эффект	Визуальный результат	Математическая основа
Сепия	Коричневатые тона, имитация старых фотографий	Линейное преобразование цветовых каналов
Холодный тон	Усиление синих и зеленых оттенков	Умножение на вектор холодных тонов
Ночное видение	Зеленоватое монохромное изображение	Преобразование в градации серого с зеленым оттенком
Без эффекта	Исходное изображение	Прямая передача цвета

Результат данного этапа представлен на рис. 13–16. На экране отображается сцена с примененным эффектом постобработки. Панель управления позволяет переключаться между эффектами в реальном времени, демонстрируя немедленное визуальное изменение сцены. Этот этап завершает разработку графического приложения, предоставляя инструмент для творческого преобразования уже отрендеренного изображения.

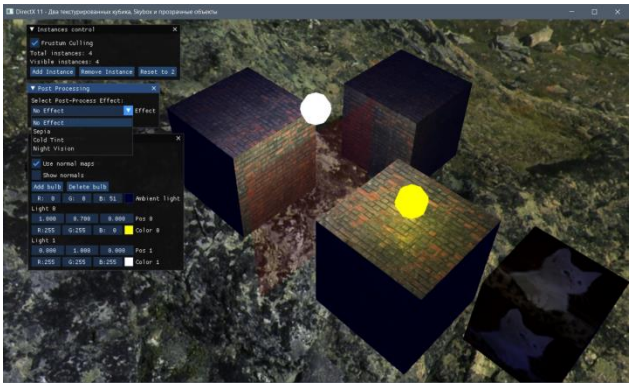


Рис. 13. Результат восьмого этапа – без эффекта

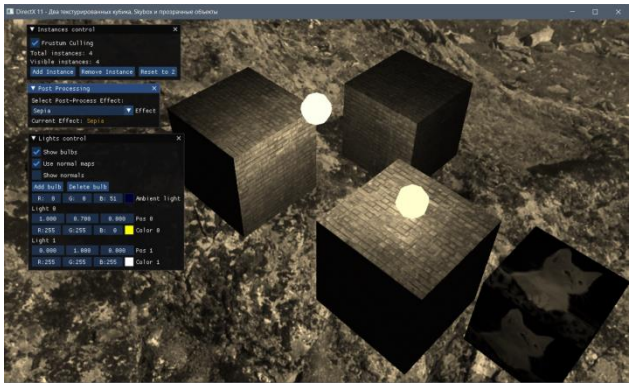


Рис. 14. Результат восьмого этапа – сепия

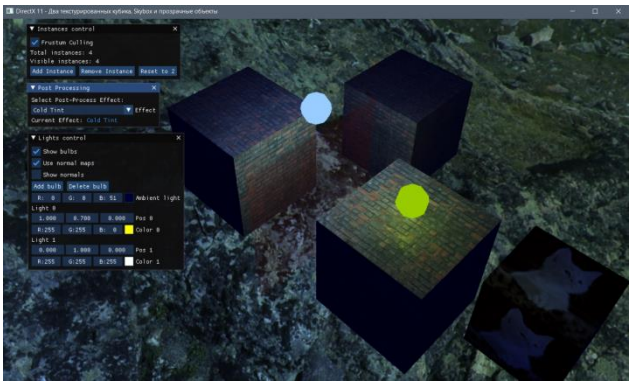


Рис. 15. Результат восьмого этапа – холодный свет

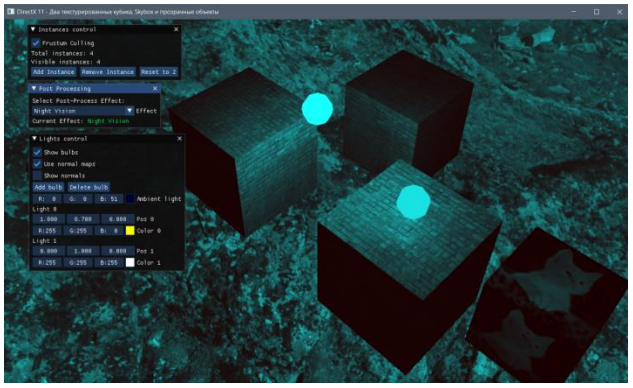


Рис. 16. Результат восьмого этапа – ночное зрение

ЗАКЛЮЧЕНИЕ

Настоящая научно-исследовательская работа была посвящена комплексному исследованию архитектуры, принципов работы и ключевых компонент графического *API DirectX 11* с целью формирования фундаментальной методологической базы для разработки оптимизированных графических приложений.

В данной работе было выполнено следующее:

1) проведён системный анализ архитектуры и базовых механизмов *DirectX 11*, а именно: исследована объектно-ориентированная модель программирования, основанная на разделении устройства (*ID3D11Device*) и контекста (*ID3D11DeviceContext*), детально проанализированы система ресурсов и их представлений (*RTV*, *DSV*, *SRV*, *UAV*), обеспечивающая гибкость конвейера, а также организация вывода через цепочку обмена (*IDXGISwapChain*) и инструменты отладки;

2) исследованы и практически реализованы ключевые техники рендеринга, составляющие основу любого графического приложения, такие как: иерархия систем координат и матричные преобразования для отрисовки трёхмерного объекта с управляемой камерой, система текстурирования с загрузкой *DDS*-текстур, включая создание реалистичного фона (*skybox*) на основе кубических текстур, механизмы управления видимостью объектов, в том числе алгоритм *Z*-буферизации, решение проблемы *z-fighting* через формат *D32_FLOAT* и технику обратной глубины, а также корректный рендеринг прозрачных объектов с альфа-смешиванием и сортировкой, динамическая модель освещения по Фонгу с поддержкой множества точечных источников, использование карт нормалей для имитации поверхностного рельефа, интеграция библиотеки *ImGui* для настройки параметров сцены в реальном времени;

3) проанализированы и успешно применены на практике основные методы оптимизации рендеринга, что позволило продемонстрировать пути

повышения производительности графических приложений, а именно: была реализована техника инстансинга (*instancing*) для эффективного рендеринга множества геометрически идентичных объектов за один вызов GPU, что значительно снизило нагрузку на центральный процессор, был внедрён алгоритм отсечения невидимой геометрии (*Frustum Culling*) на *CPU,* позволивший исключить из конвейера объекты, находящиеся вне поля зрения камеры, была создана система постобработки на основе многоэтапного рендеринга с использованием промежуточных целей отрисовки (*Render Targets*), в рамках которой реализованы и продемонстрированы различные экранные эффекты (сепия, холодный тон, ночное видение);

4) разработано полнофункциональное графическое приложение, служащее практическим подтверждением проведённого исследования: в ходе работы была применена чёткая методология – поэтапное усложнение функционала, модульная архитектура, активное использование отладочного слоя *DirectX 11* для валидации и поиска ошибок, что обеспечило создание устойчивой, расширяемой и надёжной кодовой базы.

В результате работы, поставленные задачи были решены в полной мере, и цель была успешно выполнена. Разработанное приложение наглядно демонстрирует работу ключевых компонент *API* и эффективность применённых оптимизационных методик.

В заключение работы можно сделать следующие выводы:

1) графический *API DirectX 11*, благодаря своей сбалансированной архитектуре, сочетающей относительную высокоуровневость с близостью к аппаратному обеспечению, является эффективной платформой для системного изучения фундаментальных принципов компьютерной графики, таких как организация графического конвейера, управление ресурсами и шейдерное программирование;

2) последовательная практическая реализация техник рендеринга – от работы с геометрией и текстурами до сложного освещения и управления

прозрачностью – подтвердила теоретические положения и позволила сформировать целостное понимание процесса формирования конечного изображения;

3) применённые методы оптимизации, в частности инстансинг и отсечение по пирамиде видимости, показали свою высокую эффективность для рендеринга сцен с большим количеством объектов, что является критически важным для современных графических приложений;

4) разработанное в ходе исследования приложение и использованная методология разработки (поэтапное усложнение, модульность, активная отладка) представляют собой готовую основу для дальнейшего углублённого изучения более сложных графических технологий и алгоритмов – полученный опыт и кодовая база напрямую применимы для решения задач, требующих понимания внутренних механизмов работы графического конвейера.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Shakaev V., Shabalina O., Kamaev V. Interactive Graphics Applications Development: An Effect Framework for Directx 11 //World Applied Sciences Journal. – 2013. – Т. 24. – №. 24. – С. 165-170.

2. Боресков А.В. Разработка и отладка шейдеров. – СПб.: БХВ-Петербург, 2006. – 496 с.

3. Pharr M., Randima F. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. – Addison-Wesley, 2005. – 814 pp.

4. Яблонская Ю.С., Черносвитов А.В. Программирование трехмерной графики с помощью шейдеров в рамках библиотеки OpenGL., 2016.

5. Luna F. Introduction to 3D game programming with DirectX 11. – Walter de Gruyter GmbH & Co KG, 2012.

6. Cubemaps: the salt of computer graphics): [сайт]. – URL: https://shawnhargreaves.com/blog/cubemaps-the-salt-of-computer-graphics.html (дата обращения: 13.01.2026).

7. Varcholik P. Real-time 3D rendering with DirectX and HLSL: A practical guide to graphics programming. – Addison-Wesley Professional, 2014.

8. Гитхаб с полным кодом для каждого из этапов: [сайт]. – URL: https://github.com/AnastasiaHudina/DirectX11 (дата обращения: 13.01.2026).