Муниципальное бюджетное общеобразовательное учреждение

Средняя общеобразовательная школа №70

**ИНДИВИДУАЛЬНЫЙ ПРОЕКТ**

Компьютерная графика. Разработка игрового движка на основе графической библиотеки OpenGL.

Руководитель:

Циперман Владимир Евсеевич,

учитель информатики

Работу выполнил:

Петренко Артемий Евгеньевич,

ученик 10 «А» класса

Работа допущена к защите:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

г. Воронеж, 2025

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ. 3](#_Toc195489948)

[ГЛАВА 1. Компьютерная графика, методы построения и технологии 5](#_Toc195489949)

[1.1. Понятие компьютерной графики 5](#_Toc195489950)

[1.2. Представление цвета в компьютерной графике 6](#_Toc195489951)

[1.2.1. Цветовая модель RGB 7](#_Toc195489952)

[1.3. Основные этапы рендеринга 3D-графики 8](#_Toc195489953)

[1.3.1. Обработка вершин 8](#_Toc195489954)

[1.3.2. Растеризация 10](#_Toc195489955)

[1.3.3. Обработка фрагментов 13](#_Toc195489956)

[1.4. Конвейер рендеринга в OpenGL 18](#_Toc195489957)

[1.4.1. Реализация конвейера рендеринга в OpenGL 19](#_Toc195489958)

[1.4.2. Отрисовка треугольника в OpenGL 21](#_Toc195489959)

[ГЛАВА 2. Создание игрового движка 25](#_Toc195489960)

[2.1. Определение целей работы 25](#_Toc195489961)

[2.2. Выбор инструментов и среды разработки 26](#_Toc195489962)

[2.3. Архитектура игрового движка 27](#_Toc195489963)

[2.4. Основные компоненты движка 29](#_Toc195489964)

[2.5. Реализация ключевых функций движка 30](#_Toc195489965)

[2.5.1. Отображение 3D-объектов и взаимодействие с ними 30](#_Toc195489966)

[2.5.2. Управление камерой 32](#_Toc195489967)

[2.5.3. Обработка освещения 32](#_Toc195489968)

[2.5.4. Демонстрационный уровень 33](#_Toc195489969)

[ЗАКЛЮЧЕНИЕ 35](#_Toc195489970)

[СПИСОК ЛИТЕРАТУРЫ 36](#_Toc195489971)

# **ВВЕДЕНИЕ.**

**Актуальность.** Современный мир невозможно представить без компьютерных технологий. Растущий интерес к производству и потреблению различного рода видео-контента поднял планку сложности и качества компьютерной графики, что в свою очередь, требует от разработчиков создания эффективных и гибких инструментов для её охвата. Разработка собственного игрового движка позволяет на практике изучить основы компьютерной графики и получить опыт в построении архитектуры приложения, который будет полезен для дальнейшей профессиональной деятельности.

**Цель.** Создать прототип игрового движка на основе графического API – OpenGL, который сможет служить основой для разработки 2D и 3D игр.

**Задачи.**

1. Изучить основы компьютерной графики и методы её построения на экране устройства.
2. Познакомиться и научиться работать с графическим API – OpenGL.
3. Разработать архитектуру игрового движка.
4. Реализовать базовые функции движка:

* Отображение 3D-объектов и базовые действия над ними (перемещение, масштабирование, вращение);
* Управление камерой;
* Обработка освещения;

1. Создать демонстрационный уровень с использованием движка.

**Назначение проекта.** Разработка игрового движка имеет учебное и практическое значение. С точки зрения обучения – проект позволяет понять и закрепить основные концепции компьютерной графики; освоить принципы построения архитектуры приложения; получить новый опыт работы с языками программирования и современными инструментами для создания ПО. С практической стороны – разработанный движок может быть использован, как основа, упрощающая процесс разработки современных 2D и 3D проектов.

# **ГЛАВА 1. Компьютерная графика, методы построения и технологии**

## **Понятие компьютерной графики**

Компьютерная графика – область информатики, изучающая создание, способы хранения и обработки изображений с помощью компьютера.

В компьютерной графике выделяют разделы, изучающие методы работы с изображением на плоскости, 2D графику, и трехмерную (3D) графику.

Трехмерное изображение отличается от двухмерного, тем, что строится исходя из математического описания некоторой трехмерной сцены. Математическое описание сцены чаще всего является моделью физических объектов в трехмерном пространстве. Таким образом, для получения трехмерного изображения требуется построить математическую модель сцены и объектов на ней, а далее визуализировать путем получения проекции с учетом освещения материалов и пр. В результате визуализации мы получим изображение на плоскости экрана.

Кроме этого, по способу представления изображения в памяти ЭВМ, компьютерную графику разделяют на 3 основных вида:

* **растровая графика –** изображение представляется в виде массива пикселей (точек), каждый из которых имеет свой цвет. Обладает высокой детализацией и подходит для работы со сложными изображениями, но зависит от разрешения (при увеличении/уменьшении разрешения изменяется объем изображения и его качество);
* **векторная графика** – изображение описывается с помощью математических формул линий, кривых и др. геометрических примитивов, поэтому объекты, построенные этим видом графики масштабируются без потери качества и занимают меньший объем в памяти по сравнению с растровой графикой, но графика не подходит для работы с фотографиями и сложными изображениями из-за ограниченного реализма;
* **фрактальная графика –** изображение строится с помощью математических формул, которые генерируют фракталы (сложные геометрические структуры с самоподобием). Фракталы создаются с помощью рекурсивных алгоритмов. Обладают бесконечной детализацией, подходят для создания абстрактных изображений, но применение очень ограниченно, не подходят для создания фотореалистичных изображений.

Дальше в главе речь будет про работу с растровой графикой, т.к. она наиболее подходит под вышеописанные задачи проекта.

## **Представление цвета в компьютерной графике**

Понятие цвета тесно связано с тем, как человек воспринимает свет. Можно сказать, что ощущение цвета формируется человеческим мозгом в результате анализа электромагнитного излучения (света), попадающего на сетчатку глаз. Считается, что в глазе человека существует три группы цветовых рецепторов, каждая из которых чувствительна к определенному диапазону длин волны. Каждая группа формирует один из трех основных цветов: красный, зеленый, синий.

В связи с необходимостью описания различных физических процессов воспроизведения цвета были разработаны различные цветовые модели - математические системы, которые описывают цвет как комбинацию базовых компонентов. Считается, что в среднем человек способен воспринимать около 256 оттенков одного цвета. Таким образом, любой цвет можно разложить на оттенки основных цветов и обозначить его набором цифр – цветовых координат. Таким образом, при выборе цветовой модели можно определять трехмерное цветовое координатное пространство, внутри которого каждый цвет представляется точкой. Выделяют несколько наиболее распространенных цветовых моделей:

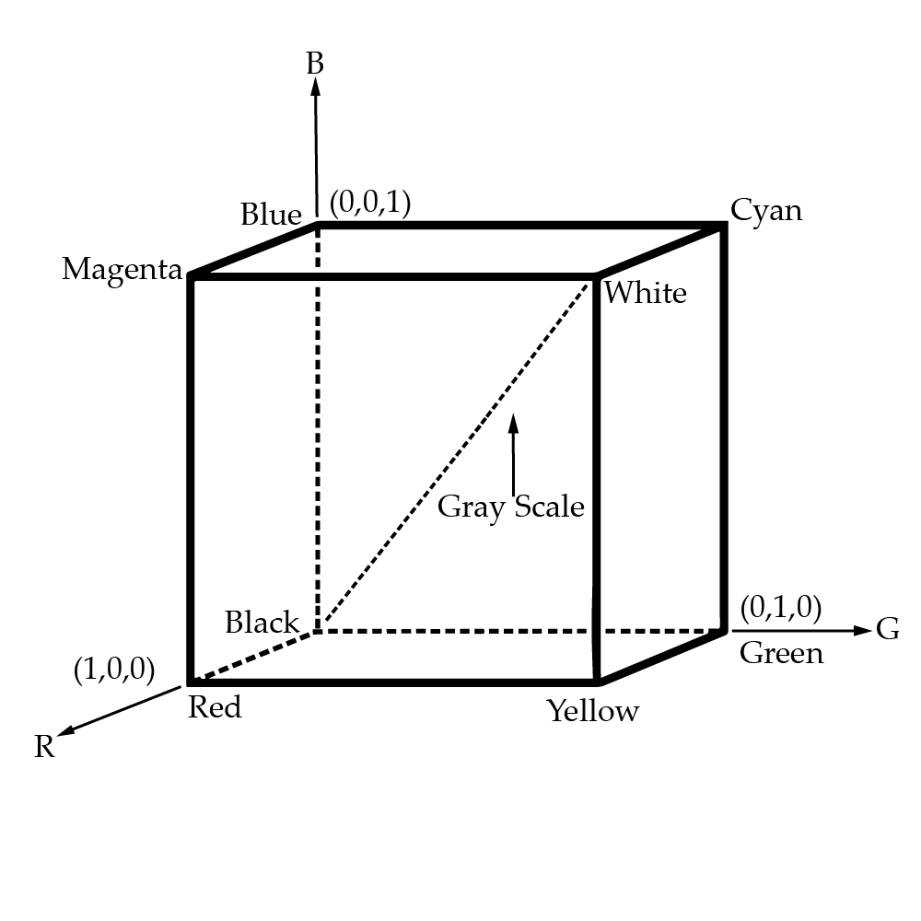
* **RGB –** используется для отображения цвета на экранах;
* **CMYK –** используется для печати;
* **HSV/HSL –** удобна для работы с цветом в графических редакторах.

Под задачи проекта лучше всего подходит цветовая модель RGB, поэтому дальше речь пойдет о ней.

### Цветовая модель RGB

Модель RGB основана на принципе аддитивного сложения цветов, когда различные оттенки получаются путем комбинации трех основных цветов: красного (Red), зеленого (Green) и синего (Blue). Каждый канал – R, G, B имеется свой отдельный параметр, указывающий на количество соответствующей компоненты в конечном цвете. Например: (255, 64, 23) – цвет, содержащий сильный красный компонент, немного зеленого и совсем немного синего.

Значения координат R, G и B можно считать принадлежащими отрезку [0,1], что представляет пространство RGB в виде единичного куба (рис. 1.1).



*Цветовое пространство RGB в виде куба (рис. 1.1.)*

Информация из предыдущих пунктов даёт базовое понимание о представлении изображения на экранах мониторов. Теперь следует разобрать процесс построения графики до её вывода на экран и технологии, использующиеся для этого. Т.к. в основе проекта преимущественно будет лежать 3D-графика, дальнейшая информация будет именно про неё (хотя практически те же методы используются и для работы с 2D-графикой).

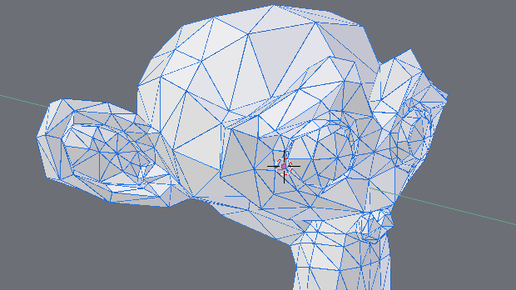
## **Основные этапы рендеринга 3D-графики**

Рендеринг 3D-графики — это сложный процесс, который включает множество этапов, но для базового понимания достаточно знать только основные: **обработка вершин, растеризация, обработка фрагментов.** Далее каждый этап будет разбираться отдельно.

### Обработка вершин

Основная идея этого шага в том, чтобы взять всю геометрию и все сетки объектов в трёхмерном пространстве и с помощью поля зрения камеры рассчитать, где каждый объект окажется в двухмерном окне, называемом экраном просмотра.

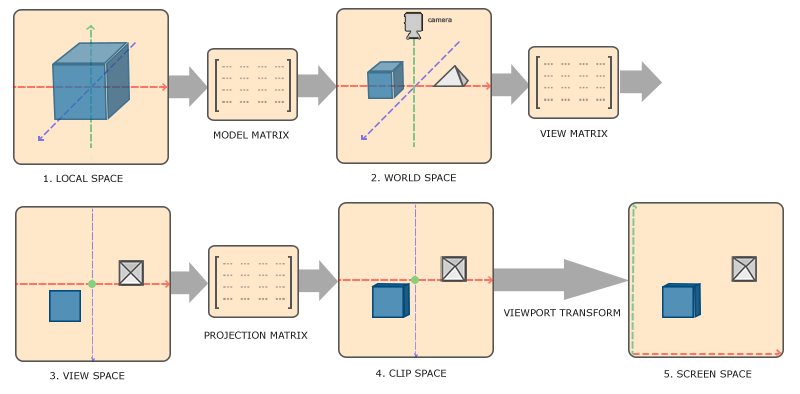
Каждый объект на сцене представляет из себя набор плоских треугольников, состоящих из 3-х вершин и различных материалов и цветов, нанесенных на поверхность треугольников (рис 1.2.).



*Пример представления модели как набор плоских треугольников (рис. 1.2.)*

Концептуально каждый объект перемещается на экран просмотра, как единое целое. Но на самом деле каждая из вершин объекта перемещается строго по одной. Процесс перемещения каждой вершины, а вместе с ней и связанных треугольников, из трехмерного мира на двухмерный экран просмотра выполняется с помощью трех преобразований.

Сначала вершина перемещается из пространства модели (Local space) в мировое пространство (World space), затем из мирового пространства в пространство камеры (View space), её отсечения (Clip space) и, наконец, на экран просмотра (Screen space) (рис. 1.3).



*Процесс перевода 3D координат объекта в координаты дисплея (рис. 1.3.)*

Для выполнения этих преобразований используются координаты X, Y и Z вершины в пространстве модели. Затем учитываются её позиция, масштаб и поворот в мировом пространстве и в конце — координаты и поворот камеры вместе с полем зрения. Далее все эти числа подставляются в различные матрицы преобразований (Model matrix, View matrix, Projection matrix) и перемножаются, получая значения x и y для вершины на экране просмотра, а также значения z или глубины, которое будет использоваться для определения перекрытия объектов.

И весь этот процесс выполняется каждый кадр для каждой вершины модели, которых может быть тысячи или миллионы. Поэтому современные графические процессоры заточены для обработки треугольных сеток и за десятилетия эволюционировали настолько, что способны обрабатывать миллионы треугольников каждые несколько миллисекунд.

### Растеризация

После того, как все вершины перемещены на двухмерную плоскость, следующий шаг — использовать три вершины каждого отдельного треугольника и определить, какие конкретные пиксели на дисплее покрываются этим треугольником. Этот процесс называется растеризацией.

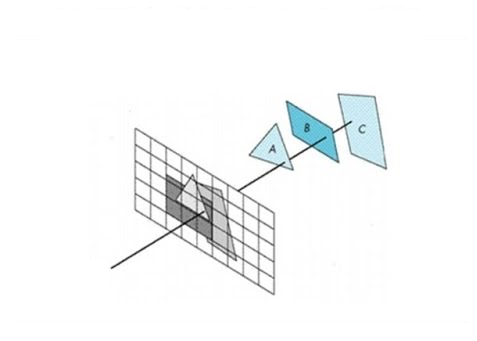
Используя координаты вершин данного треугольника, графический процессор вычисляет, где фигура располагается и какие именно пиксели покрываются ей. Затем пиксели закрашиваются с использованием текстуры или цвета, назначенного этому треугольнику.

Ещё для правильности рендеринга важно учитывать, какие треугольники находятся спереди, а какие сзади. Данная задача решается с помощью Z-буфера или буфера глубины. Этот буфер добавляет дополнительное значение к каждому пикселю на экране, которое точно соответствует расстоянию этого пикселя от виртуальной камеры.

Когда треугольник растеризуется, значение глубины (Z) этого треугольника сравнивается со значениями, хранящимися в Z-буфере. Если значения глубины треугольника меньше, чем те, что находятся в Z-буфере, то треугольник расположен ближе к камере, пиксели закрашиваются цветом треугольника и заменяются значения в Z-буфере на значения глубины этого треугольника.

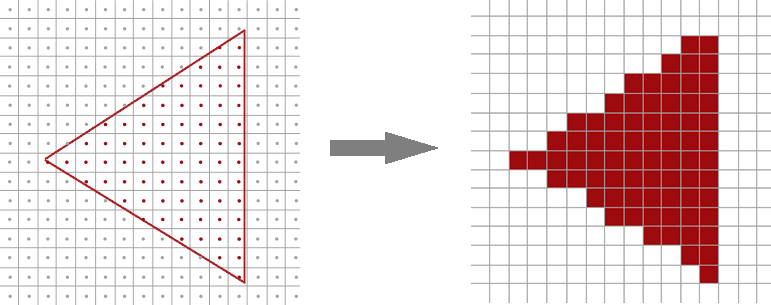
Однако, если появляется второй треугольник со значениями Z больше, чем те, что находятся в Z-буфере, то треугольник расположен дальше. Тогда он отбрасывается и сохраняются пиксели от ранее нарисованного треугольника с меньшими значениями Z.

Используя этот метод, на экране будут отображаться только ближайшие к камере треугольники с наименьшими значениями Z (рис. 1.4.).



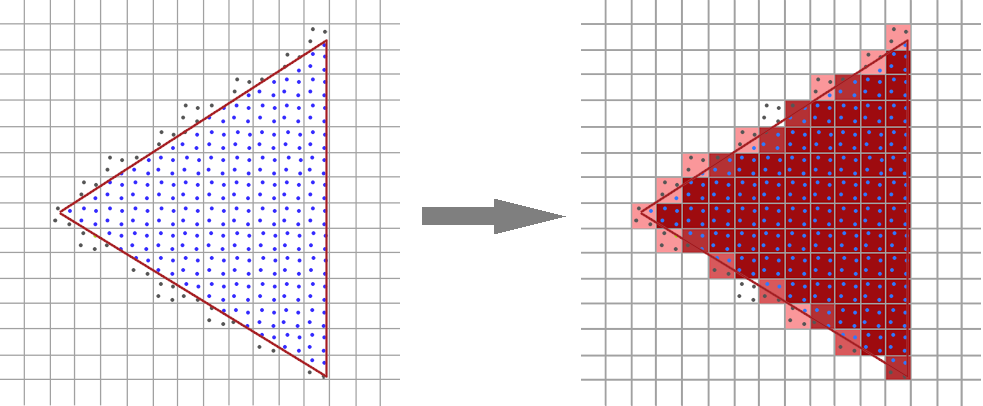
*Алгоритм Z-буфера (рис. 1.4.)*

Ещё одна проблема растеризации в пикселе заключается в том, что если треугольник проходит под углом через центр пикселя, то весь пиксель закрашивается цветом этого треугольника, что приводит к зазубренным и пикселизованным краям, такое явление ещё называют алиасинг (рис 1.5.).

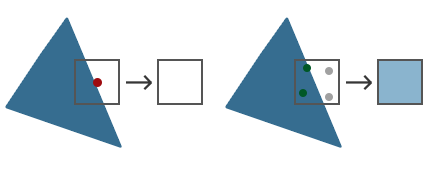


*Пример алиасинга (рис. 1.5.)*

Чтобы уменьшить появление этих зазубренных краёв, графические процессоры используют технику сглаживания методом мультисэмплинга. При использовании этого сглаживания в каждый пиксель как бы помещается 4 точки выборки (может быть другое кол-во, зависит от реализации) вместо 1. И когда треугольник пересекает пиксель, в зависимости от того, сколько из 4 точек он покрывает, пикселю присваивается соответствующий промежуточный оттенок этого цвета (рис. 1.6.). Это создает плавные края на изображении и значительно уменьшает заметную пикселизацию (рис. 1.7.).



*Пример изображения, к которому был применен алгоритм мультисэмплинга (рис. 1.7.)*



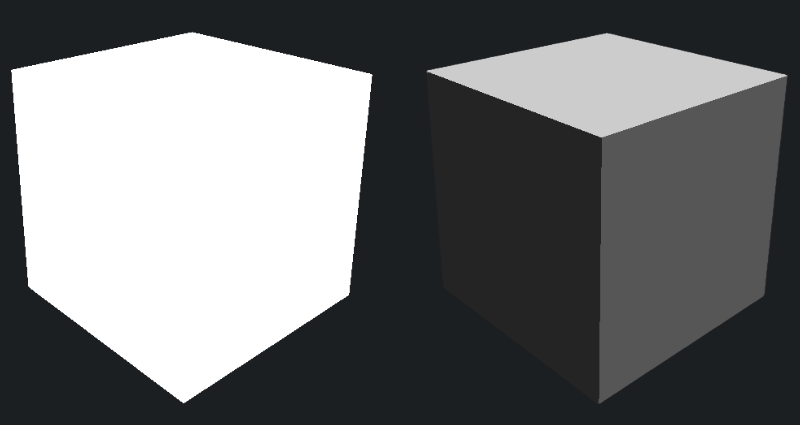
*Принцип алгоритма мультисэмплинга (рис. 1.6.)*

Таким образом, при растеризации треугольники превращаются во фрагменты, которые представляют собой группы пикселей, имеющих одинаковую текстуру или цвет. Затем по циклу то же самое происходит со следующим треугольником и т.д., до тех пор, пока не будет отрисована вся модель целиком.

### Обработка фрагментов

Теперь, когда получен набор пикселей для каждого треугольника, недостаточно просто закрасить их. Чтобы сделать сцену реалистичной, нужно учитывать направление и силу света или освещения, положение камеры, а также отражение и тени от других объектов. Обработка фрагментов используется для точного расчёта освещения каждого пикселя, делая сцену реалистичной.

Без правильного освещения даже простые объекты выглядят плоскими и неестественными. Например, если равномерно закрасить все грани куба одним цветом, он будет восприниматься как непонятный многоугольник. Но если применить расчёт освещения, учитывающий направление света и нормали поверхностей, куб обретает объём (рис. 1.8.).



*Пример обычной заливки цвета и заливки с учётом затенения (рис. 1.8.)*

В реальном мире свет ведёт себя сложно: он отражается, рассеивается, поглощается, преломляется. Полное физическое моделирование этого процесса требует огромных вычислений. Поэтому в компьютерной графике используют модели освещения **–** упрощённые, но эффективные математические способы описания поведения света.

Существует множество подходов, отличающихся сложностью и реализмом, вот несколько из них:

* **Плоское (Flat) освещение –** объект заливается 1 цветом без учета света;
* **Модель Ламберта (Diffuse-only) –** учитывает только рассеянный свет (Diffuse);
* **Модель Фонга (Phong) -** Добавляет к Diffuse зеркальные блики (Specular), тем самым может создавать эффект глянца;
* **Физически корректные модели (PBR) –** используют сложные законы оптики, требуют больше ресурсов, но дают фотореализм.

**Модель Фонга –**это «золотая середина» между простотой и реализмом, поэтому она и будет лежать в основе освещения в проекте, и поэтому дальше речь пойдет о ней.

#### **Освещение по Фонгу**

Модель освещения Фонга — это классический подход к расчету освещения в компьютерной графике, который позволяет создавать правдоподобные тени и блики на поверхностях объектов. Она была разработана Буй Туонг Фонгом в 1975 году и до сих пор остается одной из базовых техник в 3D-рендеринге.

Освещение по Фонгу складывается из **трех ключевых компонентов**:

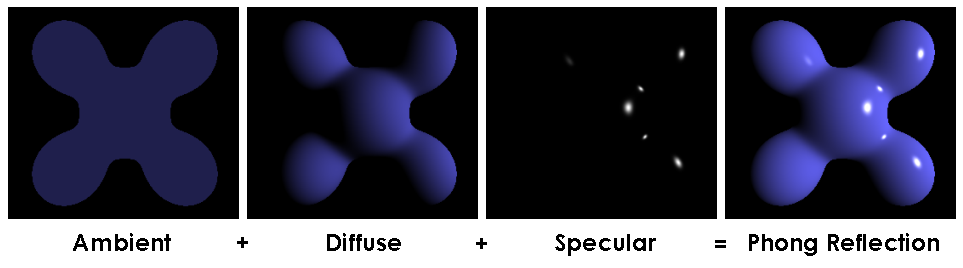
* **Фоновое освещение (Ambient) –** минимальная подсветка, которая присутствует даже в тенях. В реальном мире свет многократно отражается от окружающих объектов, поэтому абсолютно темных участков почти не бывает, поэтому компонент добавляет небольшой постоянный цвет ко всем участкам объекта.
* **Рассеянное освещение (Diffuse) -**основной свет, который падает на объект и равномерно рассеивается. Зависит от угла между нормалью поверхности и направления источника света. Так, если свет падает перпендикулярно – поверхность максимально яркая, если под острым углом – освещение слабее, если параллельно – поверхность не освещается.
* **Зеркальное отражение (Specular) -** яркие блики, которые появляются на глянцевых поверхностях. Зависит от направления света, направления камеры и гладкости материала. Блик появляется там, где отраженный свет попадает прямо в камеру. Чем более гладкая поверхность, тем ярче блик.

В формуле итоговый цвет точки вычисляется как сумма 3-х компонентов (рис. 1.9.):

***I color* = *I ambient + I diffuse + I specular***

* ***I ambient =ka \* La***
* ***ka*** *– коэффициент фонового материала*
* ***La*** *– интенсивность фонового света*
* ***I diffuse = kd \* (L \* N) \* Ld***
* ***kd​ —*** *коэффициент рассеянного отражения*
* ***L*** *– направление света*
* ***N*** *– нормаль поверхности*
* ***Ld*** *– интенсивность рассеянного света*
* ***I specular = ks \* (R \* V)­­a\* Ls***
* ***ks*** *– коэффициент зеркального отражения*
* ***R****– отраженный луч света*
* ***V****– направление к камере*
* ***a –*** *степень блеска (чем больше, тем уже блик)*
* ***Ls​****– интенсивность зеркального света*

*Формула получения цвета пикселя по модели освещения Фонга (рис. 1.9.1.)*



*Визуальное суммирование компонентов в модели освещения Фонга (рис. 1.9.2.)*

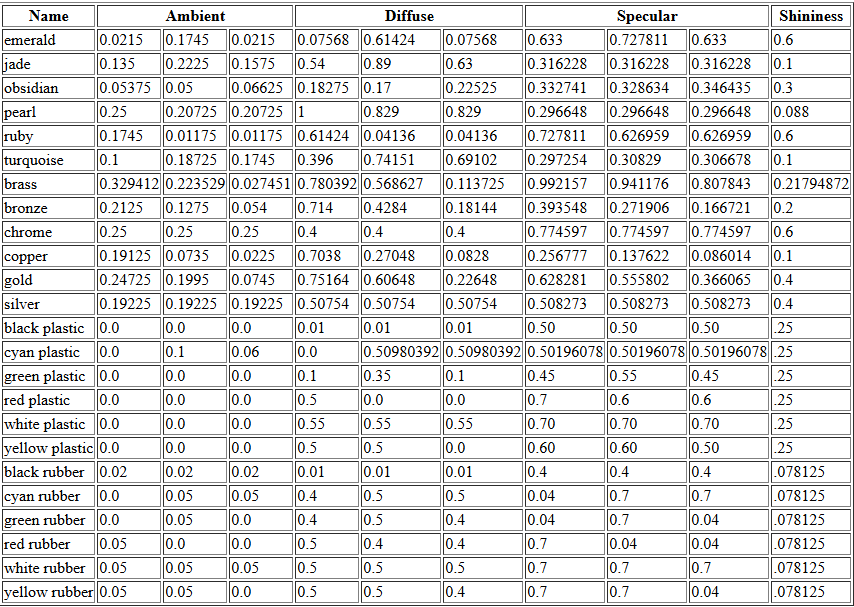
Особенность модели Фонга заключается в возможности настройки характеристик поверхности объекта через четыре основных параметра: **фоновое отражение (ambient)**, **рассеянное отражение (diffuse)**, **зеркальное отражение (specular)** и **степень блеска (shininess)**. Эти параметры работают совместно, создавая целостное восприятие материала.

**Фоновое отражение** определяет, как объект выглядит в условиях рассеянного окружающего освещения. Например, матовые и пористые материалы, такие как необработанное дерево или кирпич, обычно имеют слабое фоновое отражение, тогда как светлые гладкие поверхности, такие как мрамор или снег, характеризуются более интенсивным фоновым отражением.

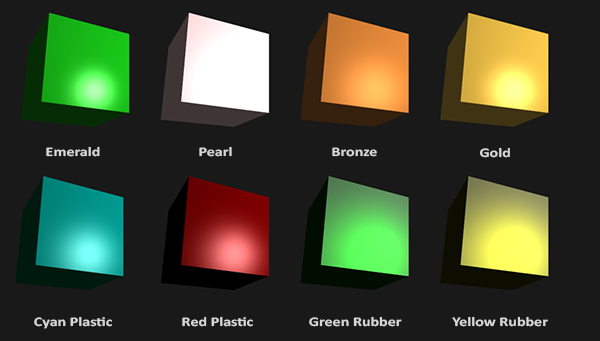
**Рассеянное отражение** отвечает за основной цвет объекта при прямом освещении. Этот компонент наиболее заметен и определяет, насколько ярко выглядит поверхность при попадании на нее света. Матовые поверхности демонстрируют равномерное рассеянное отражение по всей площади, в то время как у глянцевых материалов могут наблюдаться более выраженные переходы между освещенными и затененными участками.

**Зеркальное отражение** и **степень блеска** работают в паре, создавая эффекты бликов на поверхности. **Зеркальная составляющая** определяет интенсивность отраженного света, а **степень блеска** влияет на размер и резкость бликов. Например, полированные металлы имеют высокие значения обоих параметров, что приводит к появлению небольших, но очень ярких бликов. А матовые поверхности характеризуются слабым зеркальным отражением и низкой степенью блеска, что делает их внешний вид более мягким и равномерным.

Для создания наиболее реалистичных результатов можно использовать готовые значения для каждого параметра из различных таблиц материалов (рис 1.10.).



*Пример таблицы материалов (рис. 1.10.1.)*



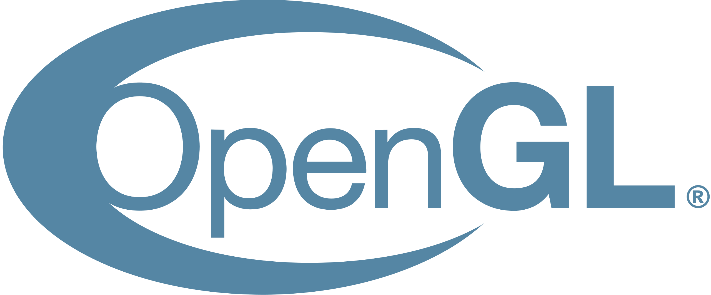
*Пример внешнего вида различных материалов (рис. 1.10.2.)*

Таким образом, рассмотренный конвейер рендеринга представляет собой строгую последовательность преобразований, где каждый этап выполняет критически важную функцию. Далее речь пойдёт о практической реализации этих принципов. Будет уделено внимание графическому API OpenGL и как описанные выше принципы воплощаются в алгоритмах и технологиях рендеринга.

## **Конвейер рендеринга в OpenGL**

Исходя из предыдущего пункта, можно сделать вывод, что рендеринг графики – это сложный процесс, требующий тесного взаимодействия между программным обеспечением и видеокартой. В начале существования компьютерной графики разработчикам приходилось писать код отдельно для каждого оборудования, что делало программы несовместимыми с разными устройствами. Каждый производитель использовал свои методы вывода изображения, из-за чего даже простой 3D-рендеринг превращался в рутину управления видеопамятью и регистрами.

Ситуация изменилась в 1992 году, когда компания Silicon Graphics (SGI) представила **OpenGL** (рис. 1.11.) — открытый и кроссплатформенный графический **API** – Интерфейс Программирования Приложений (англ. Application Programming Interface). Он стал универсальным стандартом, позволяющим писать код, который запускался бы на разных видеокартах без переписывания. Это была настоящая революция: разработчики получили единый интерфейс, абстрагирующий аппаратные особенности, а видеокарты стали поддерживать общий набор функций.



*Графический API OpenGL (рис. 1.11.)*

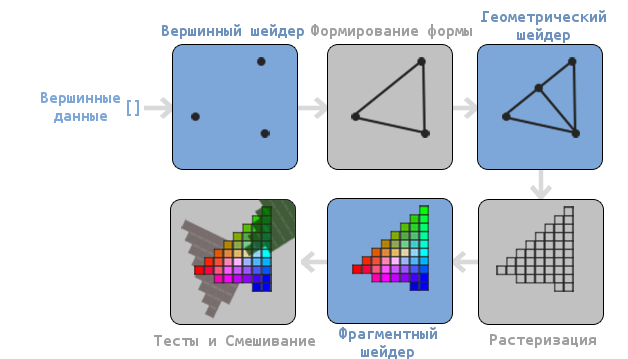
Сейчас разработчики всё чаще прибегают к более современным API вроде Vulkan или DirectX, однако для учебного проекта OpenGL – оптимальный выбор, т.к. он позволяет сосредоточиться на основах компьютерной графики, не углубляясь в сложности низкоуровневых API. Теперь речь пойдёт непосредственно о реализации конвейера рендеринга в OpenGL.

### Реализация конвейера рендеринга в OpenGL

Суть рендеринга в OpenGL ничем не отличается от всего того, что было описано выше. В OpenGL всё находится в 3D пространстве, но при этом экран и окно — это 2D матрица из пикселей. Поэтому большая часть работы OpenGL — это преобразование 3D координат в 2D пространство для отрисовки на экране. Весь процесс можно разделить на 2 большие части: первая часть преобразовывает 3D координаты в 2D координаты, а вторая часть преобразовывает 2D координаты в цветные пиксели.

Все этапы конвейера крайне специализированы и могут с легкостью исполняться параллельно. По причине их параллельной природы большинство современных GPU (графических процессоров) имеют возможности для быстрой обработки данных графического конвейера с помощью запуска большого количества маленьких программ на каждом этапе рендеринга. Эти программы называются **шейдерами**.

Некоторые из этих шейдеров могут настраиваться разработчиком, что позволяет писать собственные шейдеры для замены стандартных. Это дает гораздо больше возможностей тонкой настройки специфичных мест конвейера. На изображении ниже можно увидеть примерное представление всех этапов графического конвейера (рис. 1.12.). Синие части описывают этапы, для которых разработчик может описывать свои собственные шейдеры.



*Представление графического конвейера в OpenGL (рис. 1.12.)*

На вход конвейера передается массив 3D координат, из которых можно сформировать треугольники. Вершина — это набор данных поверх 3D координаты. Эти данные представляются, используя атрибуты вершины, которые могут содержать любые данные, например позиция, цвет и др.

Первый этап конвейера — это вершинный шейдер, который принимает на вход одну вершину. Основная задача вершинного шейдера — это преобразования 3D координат в другие 3D координаты и тот факт, что есть возможность изменения этого шейдера, позволяет выполнять некоторые основные преобразования над значениями вершины.

Следующий этап – сборка примитивов. Это этап, принимающий на вход все вершины из вершинного шейдера, которые формируют примитив, и собирает из них сам примитив. (Какой именно примитив нужно собрать, указывается на этапе вызовы команд отрисовки, например точки, треугольники, линии).

Результат этапа сборки примитивов передается геометрическому шейдеру. Он же в свою очередь на вход принимает набор вершин, формирующих примитивы и может генерировать другие фигуры с помощью генерации новых вершин для формирования новых (или других) примитивов. К примеру, в на (рис 1.12.) он сгенерирует второй треугольник.

Результат работы геометрического шейдера передается на этап растеризации, где результирующие примитивы будут соотноситься с пикселями на экране, формируя фрагмент для фрагментного шейдера. Перед тем как запустится фрагментный шейдер, выполняется вырезка. Она отбрасывает все фрагменты, которые находятся вне поля зрения, повышая таким образом производительность.

Основная цель фрагментного шейдера — это вычисление конечного цвета пикселя, а также это, чаще всего, этап, когда выполняются все дополнительные эффекты OpenGL. Зачастую фрагментный шейдер содержит всю информацию о 3D сцене, которую можно использовать для модификации финального цвета.

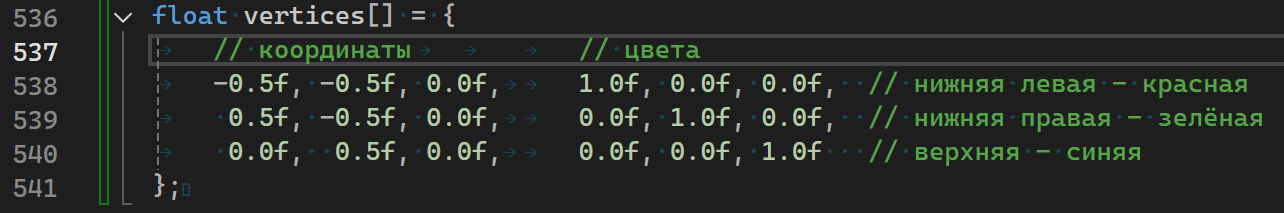
После того как определение всех соответствующих цветовых значений будет закончено, результат пройдет еще один этап, который называется альфа тестирование и смешивание. Этот этап проверяет соответствующие значения глубины фрагмента и использует их для проверки местоположения фрагмента относительно других объектов. Этот этап также проверяет значения прозрачности и смешивает цвета, если это необходимо. Таким образом во время отрисовки множественных примитивов, результирующий цвет пикселя может отличаться от цвета, вычисленного фрагментным шейдером.

Несмотря на сложность всего процесса, для базовых задач достаточно работать с вершинным и фрагментным шейдерами. Геометрический шейдер не обязателен и зачастую оставляется стандартным. Перед завершением 1 главы будет разобран упрощённый пример отрисовки обычного треугольника в OpenGL.

### Отрисовка треугольника в OpenGL

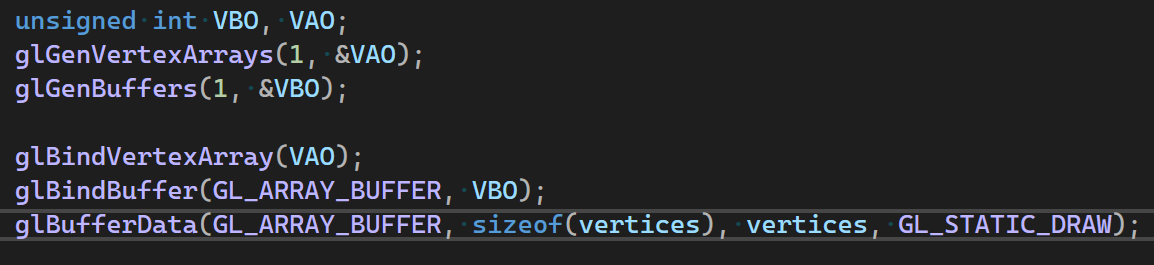
OpenGL — это 3D библиотека и поэтому все координаты, которые сообщаются ей, находятся в трехмерном пространстве (x, y и z). OpenGL обрабатывает только 3D координаты в определенном промежутке между **-1.0** и **1.0** по всем трем осям. Все такие координаты называются нормализованными.

Всё начинается с создания массива вершин. Треугольник можно представить в виде трёх точек на плоскости, где каждая вершина имеет не только координаты (x, y, z), но и свой цвет (r, g, b) (рис 1.13.)



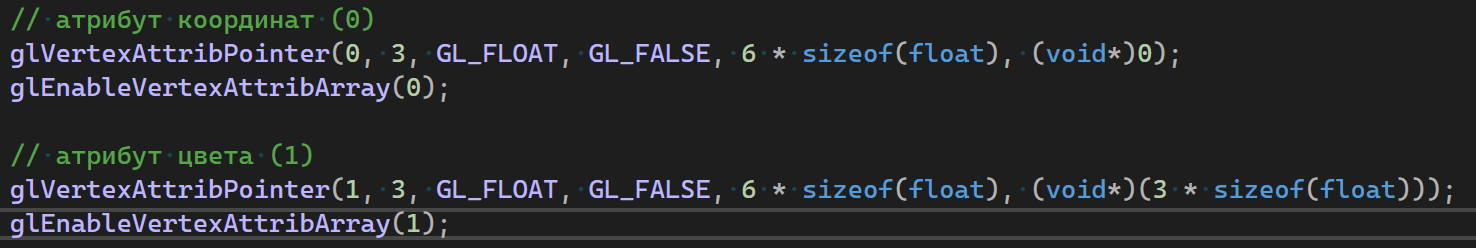
*Создание массива вершин треугольника (рис. 1.13.)*

Эти данные нужно передать в видеопамять GPU. Для этого создаётся Vertex Buffer Object (VBO) и Vertex Array Object (VAO) (рис 1.14.). Здесь VAO выступает как контейнер для настроек атрибутов вершин, а VBO хранит сами данные.



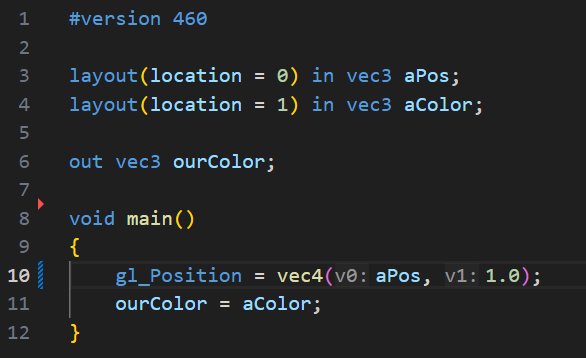
*Передача данных вершин в память GPU (рис. 1.14.)*

Теперь нужно объяснить OpenGL, как интерпретировать наши данные (рис. 1.15.). Здесь задаются атрибут 0 – 3 числа типа float (координаты), идущие через каждые 6 чисел (шаг) и атрибут 1 – цвета, начинающиеся с 3-го числа в каждом блоке.



*Создание атрибутов вершин (рис. 1.15.)*

Далее нужно написать шейдеры. Вершинный шейдер преобразует координаты и передаёт цвет дальше (рис 1.16.), а фрагментный – берёт переданный цвет и выводит его (рис. 1.17.).

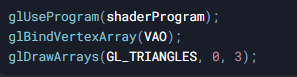


*Создание вершинного шейдера (рис. 1.16.)*



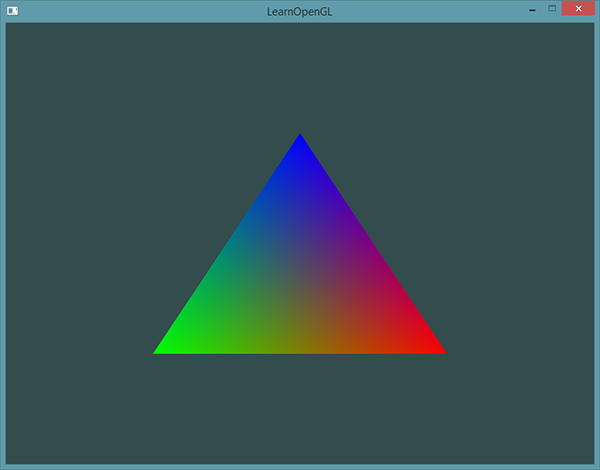
*Создание фрагментного шейдера (рис. 1.17.)*

И наконец в основном цикле рендеринга нужно активировать шейдерную программу (связка из ранее написанных шейдеров) и отрисовать получившийся примитив (рис. 1.18.). Здесь GLDrawArraysактивирует конвейер рендеринга, в параметры функции передается название примитива, который нужно составлять из полученных вершин, и кол-во этих вершин. Далее для каждой из 3-х вершин вызывается вершинный шейдер, OpenGL автоматически интерполирует цвета между вершинами и для каждого фрагмента вызывается фрагментный шейдер.



*Отрисовка треугольника (рис. 1.18.)*

В результате, на экране появляется треугольник с плавным цветовым градиентом (рис 1.19.).  Этот градиент — результат интерполяции цветов в процессе растеризации, когда фрагментный шейдер получает промежуточные значения между вершинами.



*Пример получившегося треугольника (рис. 1.19.)*

Такой простой пример закладывает основу для более сложной графики. Понимание конвейера графики даёт разработчику возможность создавать уникальные продукты, оптимизируя каждый этап производства. В следующей главе речь пойдет о непосредственной разработке игрового движка, опирающейся на полученной информации.

# **ГЛАВА 2. Создание игрового движка**

## **Определение целей работы**

Перед началом работы нужно чётко понимать, как должен выглядеть желаемый результат, поэтому начальным этапом в разработке было ознакомление с существующим программным обеспечением на рынке и формирование общего представления о продукте.

Первым делом было определено, что игровой движок – это программный каркас, предназначенный для создания не только видеоигр, но и любого медиаконтента, в зависимости от поставленных задач. Он выступает в роли основы, объединяющей системы рендеринга, обработки физических взаимодействий, воспроизведения звуков, искусственного интеллекта и других компонентов. По сути, движок представляет собой промежуточный слой между "железом" компьютера и медиаконтентом, избавляя разработчиков от необходимости каждый раз реализовывать базовые механизмы с нуля. Современные игровые движки различаются по специализации и сложности: от универсальных решений вроде Unreal Engine и Unity до узкоспециализированных платформ для конкретных задач.

В рамках данного проекта было решено создать минимально необходимую систему, способную обеспечить базовый цикл работы приложения:

* инициализацию;
* обработку ввода;
* обновление игрового состояния;
* отрисовку графики через OpenGL,

а также предоставить базовые возможности для редактирования игровой сцены: добавление и взаимодействие с примитивами, работа с освещением и управление состояниям объектов.

Для использования всего функционала ещё нужно создать интерфейс, который будет удобен и понятен пользователю.

## **Выбор инструментов и среды разработки**

Основным языком разработки стал C++. Данный язык считается одним из базовых инструментов в игровой индустрии. Он обеспечивает низкоуровневый контроль над памятью и аппаратными ресурсами, что критически важно для подобный проектов, где кадровая частота и эффективность рендеринга играют решающую роль.

Одной из сложностей работы с C++ — это сборка проектов на разных платформах и IDE. Этим можно заниматься вручную, но данный процесс сложный и долгий. Программа «**CMake**» решает это: пишется конфигурационный файл, где указывается, какие исходники компилировать, какие библиотеки подключать и как собирать проект. Далее **CMake** сам генерирует нужные файлы сборки под нужную платформу.

Во время создания крупных приложений постоянно появляются изменения в коде: добавляются новые функции, исправляются баги, экспериментируется подходы. Без системы контроля версий легко потерять рабочую версию или запутаться в изменениях. **Git** решает эту проблему, фиксируя каждое изменение с комментариями, предоставляя возможность возвращаться к любой предыдущей версии и др. (в один из этапов работы на рабочем компьютере вышел из строя диск с проектом, и именно благодаря данной системе всё было восстановлено и продолжена работа).

В качестве среды разработки была выбрана **Visual Studio** из-за глубокой интеграции с экосистемой C++ и наличием множества удобных инструментов для разработки, такие, как отладка кода, встроенная поддержка **CMake** и **Git**, и многое другое.

Так как OpenGL сильно абстрагирован от операционных систем, разработчику нужно самостоятельно реализовывать весь функционал взаимодействия с ОС. Чтобы не заниматься этим, была выбрана библиотека GLFW, которая обеспечивает базовую инфраструктуру для работы с окнами и вводом, не предоставляя лишних функций (например, для 2D-рендеринга, как у аналога - SFML), что делает её оптимальным выбором для чистого OpenGL. Также будет использоваться генератор загрузчика функций OpenGL – GLAD, это чисто техническая часть, занимающаяся автоматической загрузкой и взаимодействием с функциями OpenGL.

Все вышеперечисленные инструменты являются основой большинства проектов на OpenGL. Но для некоторого упрощения разработки в проект дополнительно включены несколько библиотек:

* **GLM** – математическая библиотека, предоставляющая готовые инструменты для векторных, матричных и др. сложных вычислений;
* **spdlog** – применяется для вывода отладочной информации, ошибок и т.д.;
* **ImGui** – инструмент для создания интерактивных интерфейсов (кнопки, окна, графики и т.д.);
* **stb\_image** – библиотека для загрузки изображений, упрощает работу с текстурами без лишних зависимостей;
* **ExprTk** – движок для парсинга и вычисления математических выражений из строк, лежит в основе некоторого функционала проекта.

Это все основные инструменты, которые применялись в разработке. Они обеспечивали функциональность и надёжность проекта, а также не заставляли с нуля создавать второстепенные системы, освобождая время для основной работы.

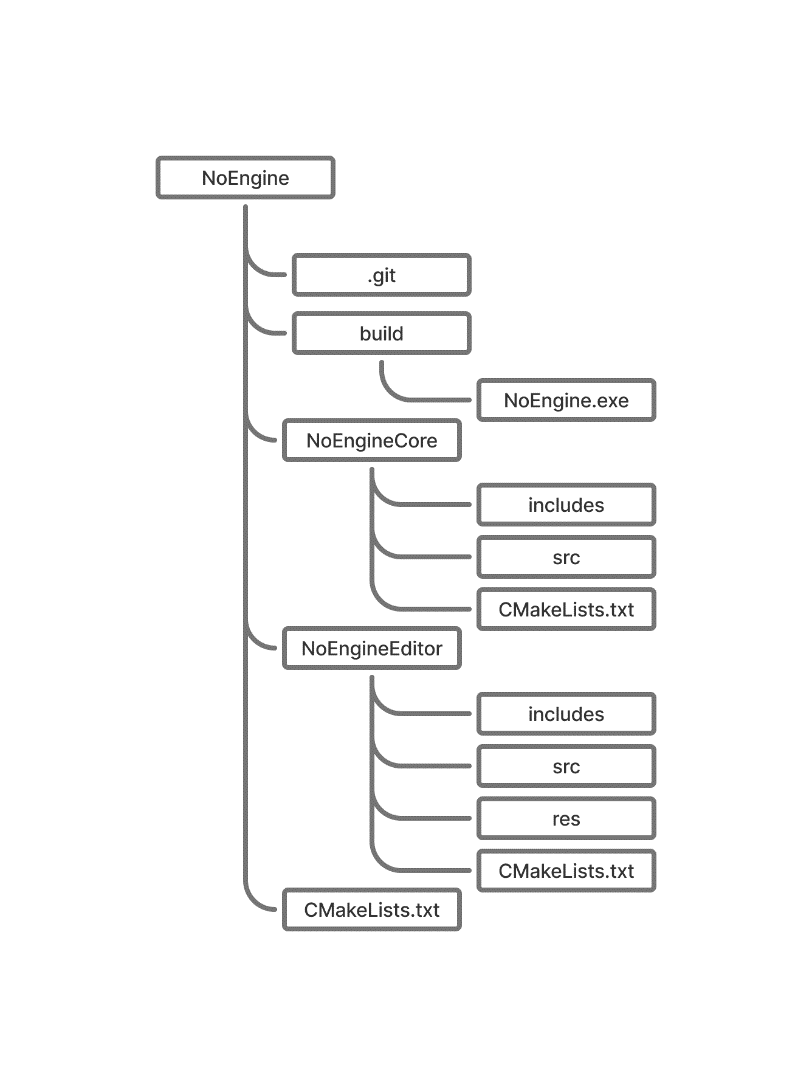
## **Архитектура игрового движка**

Структура проекта организована в виде модульной системы, разделённой на логические компоненты. Обобщённая структура проекта показана на рисунке (рис. 2.1.) («NoEngine» – это фактическое название продукта, будет часто присутствовать в названиях файлов или каталогов).

* **NoEngine/ –** корневая директория;
* **.git/ -** служебная папка системы контроля версий Git;
* **build/ –** специально созданный каталог, в котором хранятся все сгенерированные файлы CMake, в том числе и исполняемый файл **NoEngine.exe**;

Перед тем как продолжить, нужно уточнить ключевую структурную особенность проекта. В движке для большей гибкости и дополнительного запаса для нововведений было принято решение на разделение кода на две части: **NoEngineCore** и **NoEngineEditor**. NoEngineCore выступает в роли фреймворка проекта – основы, которая предоставляет стандартные решения для типовых задач. Основное отличие от библиотек, в том, что библиотеки дают только набор функций, а фреймворк задает дальнейшую архитектуру проекта, предоставляя готовые универсальные компоненты и вызывая написанные скрипты в нужный момент. Таким образом, в NoEngineCore был реализован весь основной цикл работы приложения, а в NoEngineEditor – всё пользовательское окружение.

* **includes/ –** каталог с заголовочными файлами проекта (.hpp);
* **src/ –** каталог с исходным кодом проекта (.cpp);
* **res/ -** каталог с ресурсами проекта (шрифты, текстуры, шейдеры);
* **CMakeLists.txt -** конфигурационный файл для сборки проекта



*Общая иерархия проекта (рис. 2.1.)*

## **Основные компоненты движка**

В движке реализованы компоненты, образующие каркас, на котором строится всё приложение. Эти компоненты тесно связаны друг с другом и не могут существовать по отдельности:

* **Окно –** базовый элемент, без которого невозможен вывод изображения на экран;
* **Обработка событий –** движок реагирует на системные события: изменение размера окна, паузу, закрытие приложения. Каждое событие можно по-своему обрабатывать для достижения нужных результатов;
* **Управление ресурсами –** любые графические приложения оперируют множеством данных: шейдерами, текстурами, 3D-моделями, звуками. Чтобы избежать дублирования и утечек памяти, нужна система загрузки, хранения и выгрузки ресурсов. Например, шейдеры компилируются при старте, текстуры загружаются в видеопамять, а модели хранятся в оптимизированном для рендеринга формате. Хорошая система управления ресурсами позволяет избежать падений производительности;
* **Система ввода -** движок должен обрабатывать действия пользователя: нажатия клавиш, движение мыши. Ввод обычно абстрагируется в виде событий (например, "клавиша W нажата") или состояний ("удерживается Пробел"). Это позволяет отделить логику управления от платформо-зависимого кода и обеспечить независимость от конкретного устройства.
* **Цикл приложения –** важнейший компонент, который обновляет состояние сцены и рисует кадры. Перед каждой отрисовкой кадра вызываются методы **Update –** обработка физики, ввода, игровой логики и **Render –** отрисовка сцены с учётом новых изменений.

Перечисленные выше компоненты характеры практически для любого графического приложения. В текущем проекте были реализованы еще несколько важных компонентов:

* **Система логирования –** механизм записи служебной информации во внешние файлы или консоль. В движке он решает такие задачи, как **отслеживание ошибок** (например, падение шейдера или загрузки текстуры) и **упрощение отладки** (вывод состояния переменных в реальном времени).
* **Камера –** определяет, какую часть сцены видит пользователь, задаёт **проекцию** изображения (перспективное/ортографическое), **перемещение по сцене**, определяет «правила» для пользователя, характерные для конкретной ситуации, что делает взаимодействие с интерфейсом наиболее удобным и приятным;
* **Менеджер объектов –** промежуточный слой между приложением и объектами, который управляет сущностями без прямого доступа к классам из вне. Такой подход гарантирует гибкость приложения.

## **Реализация ключевых функций движка**

Сейчас речь пойдет о том, как были реализованы заявленные в задачах функции.

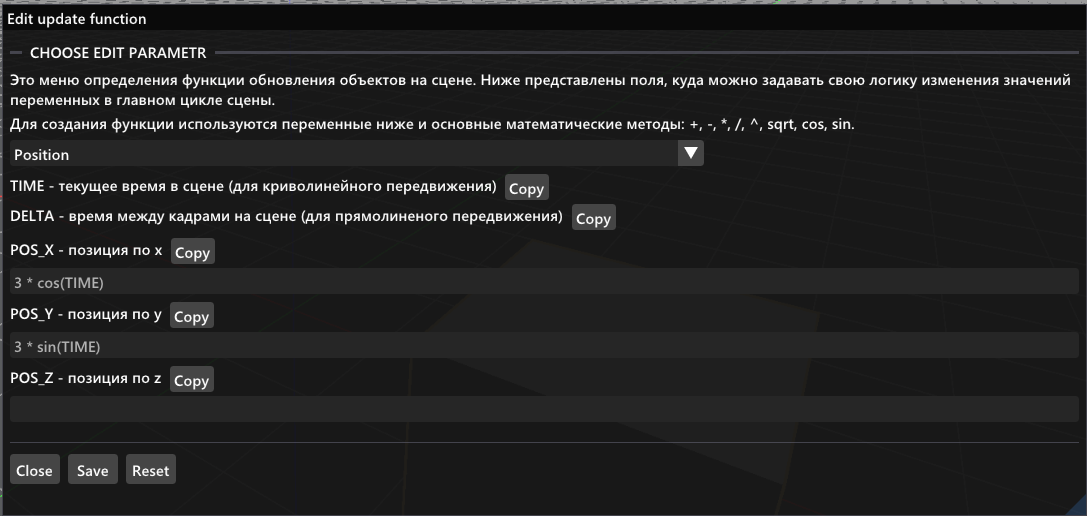
### Отображение 3D-объектов и взаимодействие с ними

В движке существует **4** основных вида объектов, с которыми пользователь может как-либо взаимодействовать (добавлять, удалять, перемещать, масштабировать и т.д.) – это **куб**, **сфера**, **плоскость** и **точечный источник света**. Обращение к ним производится через объявление их базового класса, в котором определены основные методы (например, **set\_position**, **get\_position** и т.д.), которые одинаковы для каждого объекта и не требует перегрузки, и виртуальные методы (**draw**, **update** и т.д.), реализация которых отличается у каждого объекта и определяется непосредственно в классе самого объекта. Ещё в базовом классе определены базовые переменные объектов (**position**, **rotation** и т.д.), которые так же присущи всем объектам. В дочерних классах определены индивидуальные поля и методы (например, **массив вершин куба** или **генерация вершин для сферы**).

Это простая реализация основных парадигм ООП, которая обеспечивает гибкость и масштабируемость проекта. Существуют и другие подходы, например (ECS – Entity-Component-System, где появляется еще большая гибкость из-за отделения компонентов объекта от самого объекта), но они более сложные и требуют большего времени на разработку.

Взаимодействие с объектами осуществляется путём считывания значений переменной, введенных пользователем в интерфейсе движка, получения новых значений с помощью матричных и векторных преобразований ([**пункт 1.3.1.**](#_Обработка_вершин)) и замены текущих значений переменной в объекте на новые.

Дополнительно в движке реализована система динамического обновлений переменных объекта по функциям, заданным пользователем. В отдельном меню представлены доступные переменные и функции, которыми пользователь может оперировать. Например, так будет выглядеть задание движения объекта окружности с центром в начале координат и радиусом 3 (рис. 2.2.).



*Пример задания движения объекта по окружности (рис. 2.2.)*

### Управление камерой

Движок поддерживает два вида камер. Каждая по-разному взаимодействует с пользователем и влияет на итоговое изображение.

* **Перспективная камера –** это FPS-камера (First Person View – камера от первого лица) с управлением WASD + мышь. В редакторе используется чаще всего, т.к. выходная картинка наиболее реалистична за счет перспективной проекции, где дальние объекты выглядят меньше, чем ближние. У камеры можно настроить угол обзора (FOV), ближнюю и дальнюю плоскости отсечения и скорость передвижения по сцене.
* **Ортографическая камера –** тоже FPS-камера с управлением WASD + мышь. Используется в основном при моделировании сцены, в случаях, когда, например, необходимо точно разместить объект на сцене, т.к. объекты через эту камеру сохраняют свои истинные размеры без зависимости от расстояний.

В движке реализован функционал динамического переключения между камерами во время работы программы. Это даёт пользователю удобный интерфейс взаимодействия с объектами.

### Обработка освещения

Реализованная система освещения в движке включает в себя несколько типов освещения:

* **Directional light (Направленный свет) –** имитирует удаленный мощный источник освещения, такой как солнце или луна. Этот светильник характеризуются параллельными лучами постоянной интенсивности, не меняющимися в зависимости от положения объектов в сцене. Он особенно важен для создания базового освещения открытых пространств. В движке представлен в единичном экземпляре с интерфейсом для настройки основных характеристик;
* **Point light (Точечный источник) –** представляет из себя локальный светильник, излучающий свет во всех направлениях с определенной интенсивностью, которая уменьшается с расстоянием согласно законам затухания. Такой источник идеально подходят для моделирования ламп, фонарей, факелов и других локальных световых эффектов. В движке есть возможность создания нескольких таких источников и управления ими.
* **Spotlight (Прожектор) –** сочетает свойства направленного и точечного освещения, создавая конический луч света с четко выраженной направленностью. Этот тип источников применяется для фар автомобилей, уличных прожекторов, фонариков и других ситуаций, где требуется создать направленный световой поток. В движке реализован так же в единичном экземпляре в виде «фонарика», редактирование параметров происходит в отдельном меню.

В основе системы освещения стала модифицированная модель Фонга ([**пункт 1.3.3.1.**](#_Освещение_по_Фонгу)), которая, несмотря на свою относительную простоту, обеспечивает достаточно реалистичные результаты.

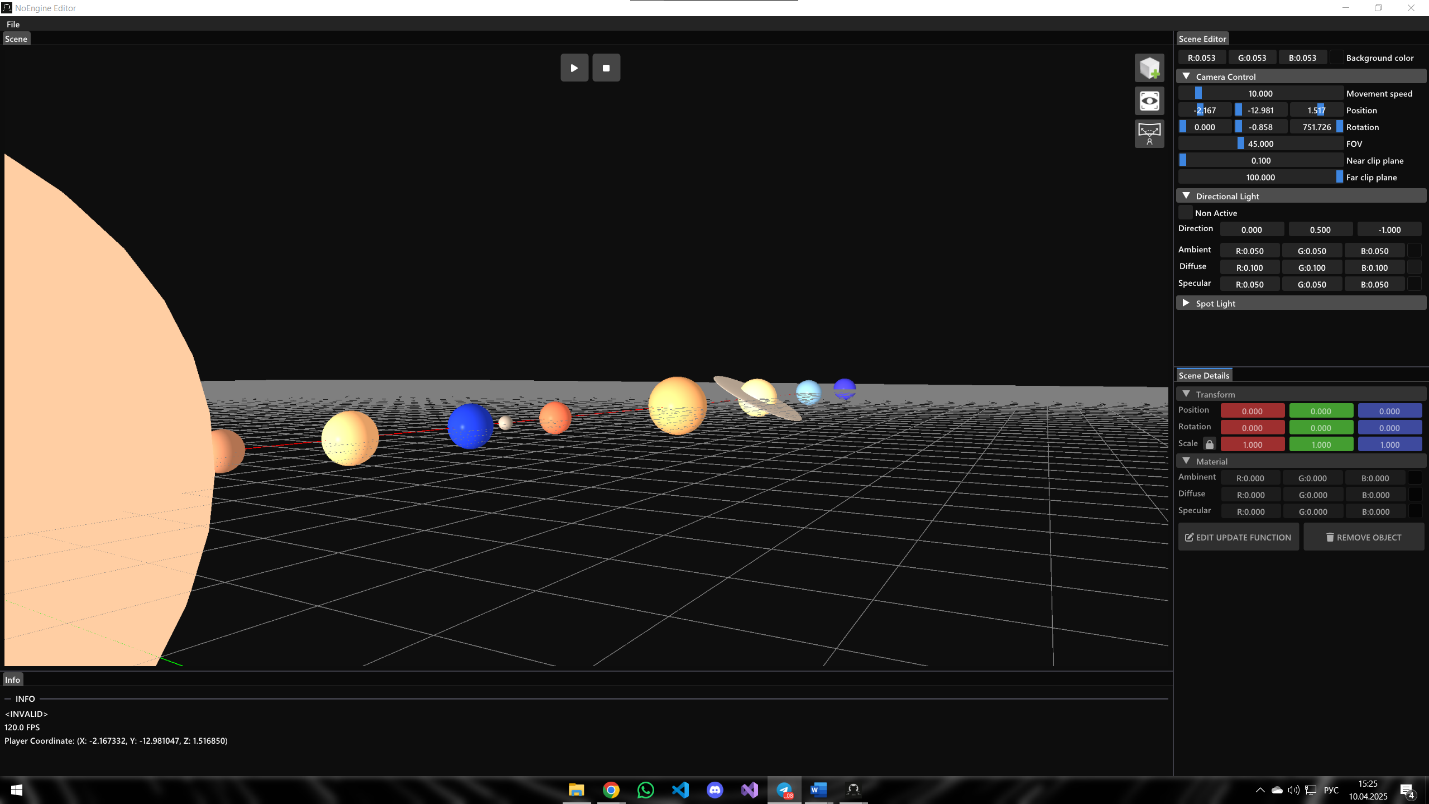
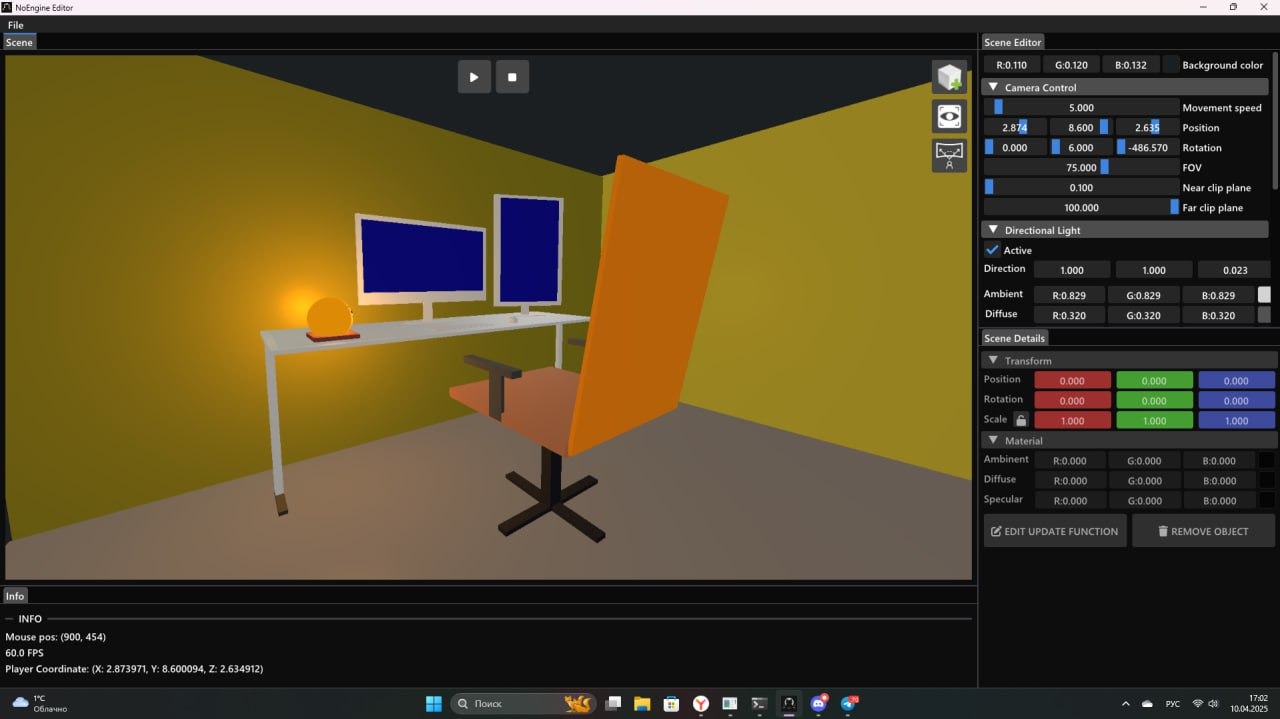
### Демонстрационный уровень

Для тестирования возможностей движка были созданы две демонстрационные сцены.

Первая сцена (рис. 2.3.) демонстрирует возможности движка в гибкой настройке объектов и их материалов для достижения поставленных целей.

Во второй сцене (рис. 2.4.) упор идёт на динамическую составляющую. К каждому объекту применен свой материал и функция обновления координаты. Эти две разные по задачам и реализации сцены говорят об универсальности движка и его широком спектре сфер применения.

*Первая демонстрационная сцена – «Комната с компьютером» (рис. 2.3.)*



*Вторая демонстрационная сцена – «Солнечная система» (рис. 2.4.)*

*Сцена в движении –* [*ссылка*](https://rutube.ru/video/private/43e362a3aa597fd115dd18115a698b41/?r=wd&p=z0HPN3Zm2BjLQ3zuTmBEVA)*.*

# **ЗАКЛЮЧЕНИЕ**

Подытожив, могу сказать, что поставленной во введении цели я полностью добился. В ходе выполнения проекта был разработан функциональный прототип игрового движка на базе графической библиотеки OpenGL, реализующий все поставленные задачи. Были изучены принципы работы 3D-графики, освоен стек технологий, необходимых для создания продукта, реализован весь запланированный функционал движка, построена модульная архитектура проекта и созданы две тестовые сцены, подтверждающие работоспособность и функциональность движка.

Эту работу, я считаю, можно использовать, как теоретическую основу для изучения современных технологий рендеринга. Она наглядно демонстрирует, как теоретические знания о компьютерной графике могут быть применены на практике. Разработанный движок может стать отличной стартовой площадкой для людей, желающих углубиться в изучение графических технологий.

С практической точки зрения, проект доказал свою эффективность как инструмент для быстрого прототипирования 3D-сцен, визуализации алгоритмов и математических моделей, создания интерактивных демонстраций.

Для меня данная тема всё ещё не исчерпана, и поэтому проект будет дальше поддерживаться и развиваться, пока не приобретёт вид полноценного игрового движка с гибкой архитектурой и объёмным функционалом.

В заключение хочу подчеркнуть, что данный проект стал для меня ценным опытом, который не только позволил закрепить теоретические знания, но и развил важные практические навыки в области программирования и разработки программного обеспечения. Полученные результаты убедительно доказывают, что даже в рамках учебного проекта можно создать работоспособную систему, сочетающую в себе образовательную ценность и практическую пользу.

# **СПИСОК ЛИТЕРАТУРЫ**

1. Дёмин А.Ю. Основы компьютерной графики: учебное пособие / А.Ю. Дёмин. - Томск: Томский политехнический университет, 2011. - 191 с.
2. Джейсон Г. Игровой движок. Программирование и внутреннее устройство. - 3-е изд. - СПб.: Питер, 2024. - 1136 с.
3. 3D Game Engine [Электронный ресурс]. - URL: <https://www.youtube.com/playlist?list=PL6x9Hnsyqn2XfzD5GK-xjsOplPWP-ByYW> (дата обращения: 05.11.2024).
4. Computer Graphics Stack Exchange: вопросы и ответы по компьютерной графике [Электронный ресурс]. - URL: <https://computergraphics.stackexchange.com/> (дата обращения: 17.02.2025).
5. Learn OpenGL: официальный учебный ресурс по OpenGL [Электронный ресурс]. - URL: <https://learnopengl.com/> (дата обращения: 10.12.2024).