МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

((НИЯУ МИФИ))

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Научная работа № 1
по теме:

«РАЗРАБОТКА ГРАФИЧЕСКОГО 3D-ДВИЖКА»

Работу выполнил студент Б25-507 _____ Буцких В.В.

Преподаватель _____ Седых И.В.

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

1 Цель работы

Цель работы — сделать упрощённую версию графического 3d-движка, чтобы подробнее изучить модель Ламберта и её практическое примение.

2 Задачи

Для выполнения работы необходимо:

1. Изучить принцип работы модели Ламберта, понять смысл формул, и упрощённо отразить это в разрабатываемом ПО.
2. Определиться с используемым языком программирования, библиотеками и фреймворками.
3. Реализовать и протестировать ПО на практике.

3 Используемое ПО и библиотеки

В качестве языка программирования был выбран Python, ввиду его простоты, удобного дебаггинга и его глубокого знания выполняющим работу. Поскольку работа сугубо научная и не планируется выпускаться в прод на данном этапе, вопрос оптимизации кода пока сугубо второстепенный.

Также для отрисовки объектов на экран использовался Tkinter — встроенная в Python библиотека для рисования на экране 2-мерных изображений. [6]

Для хранения координат вершин и прочих подобных данных была применена библиотека numpy — ввиду её большей простоты и оптимизации. [5]

4 Краткий справочник:

В программе есть три геометрических примитива для отображения: куб, сфера, тор. Их можно выбрать с помощью интерфейса, также как их цвет, а случае с шаром и тором — ещё и степень детализации (число используемых полигонов). [2]

5 Оптимизация программы

В ходе разработки было выяснено, что без алгоритмов оптимизации, программа «лагает» при большой детализации шара и тора. Это связано с большим количеством отображаемых полигонов. Для этого стандартная интерпретация кода. В качестве альтернативы можно было бы переписать программу на C++, но это значительно увеличило бы время разработки ПО. Поэтому были приняты менее радикальные улучшения:

Backface Culling

1. JIT-компиляция с Numba [3]

Основные JIT-функции:

```
@njit(parallel=True, fastmath=True)
def rotate_vertices_jit(vertices, rotation_matrix)
@njit(parallel=True, fastmath=True)
def project_vertices_jit(vertices, camera_distance, fov, width, height)
@njit(parallel=True, fastmath=True)
def calculate_face_depths_jit(rotated_vertices, faces)
```

Оптимизации:

1. parallel=True: автоматическое распараллеливание циклов
2. fastmath=True: использование быстрой математики (меньшая точность, выше скорость)
3. Векторизованные операции: работа с массивами NumPy вместо отдельных элементов
4. Предварительное выделение памяти: np.empty() вместо динамического расширения

2. Кэширование вычислений

LRU-кэш для генерации геометрии:

```
@lru_cache(maxsize=10)
def generate_sphere_cached(self, lat_segments, lon_segments)
@lru_cache(maxsize=10)
def generate_torus_cached(self, R, r, u_segments, v_segments)
```

Преимущества:

1. Избегает повторной генерации одинаковых геометрий
2. maxsize=10 ограничивает использование памяти
3. Ключи кэша основаны на параметрах детализации

Словарные кэши:

```
self.sphere_cache = {}
self.torus_cache = {}
```

1.Хранят готовые геометрии для быстрого доступа

2. Автоматическая очистка при изменении параметров

3. Алгоритмические оптимизации

Сортировка граней (Painter's Algorithm): [4]

```
face_depths = calculate_face_depths_jit(rotated_vertices, self.faces)
sorted_indices = np.argsort(-face_depths)
```

Быстрые математические операции:

```
@njit(fastmath=True)
def dot_product_jit(a, b)
@njit(fastmath=True)
def normalize_jit(v)
```

4. Оптимизации памяти

Предварительное выделение массивов:

```
total_vertices = (lat_segments + 1) * (lon_segments + 1)
vertices = np.zeros((total_vertices, 3), dtype=np.float32)
```

Эффективные структуры данных:

1. Использование np.float32 вместо np.float64
2. Статические массивы NumPy вместо списков Python
3. Минимизация копирования данных (copy() только при необходимости)

5. Оптимизации рендеринга

Пакетная обработка вершин:

Все вершины обрабатываются одним JIT-вызовом

Минимизация переходов между Python и скомпилированным кодом

Эффективное освещение:

```
intensity = self.ambient_light + (1.0 - self.ambient_light) * max(0.0, dot_product)
```

1. Быстрый расчет освещения по Ламберту
2. Оптимизированные нормали для каждой фигуры

7. Специфичные оптимизации для фигур

Для куба:

Предварительно вычисленные нормали

Статическая геометрия

Для сферы:

Параметрическая генерация с кэшированием

Адаптивное количество сегментов

Для тора:

Специальный расчет нормалей для корректного освещения

Параметрическая настройка (R, r)

1 МОДЕЛЬ ЛАМБЕРТА

1.1 Определение [1]

Модель Ламберта для описания отражения от поверхности широко используется в компьютерной графике. Она активно применяется в таких методах визуализации, как радиоузловой метод (radiosity) и трассировка лучей (ray tracing). Однако для многих реальных объектов модель Ламберта может оказаться весьма неточным приближением отражательной способности поверхности. В то время как яркость ламбертовской поверхности не зависит от направления наблюдения, яркость шероховатой поверхности увеличивается по мере приближения направления наблюдения к направлению источника света. В данной статье разработана комплексная модель, которая предсказывает отражательную способность шероховатых поверхностей. Поверхность моделируется как совокупность ламбертовских граней. Показано, что такая поверхность изначально является не-ламбертовской из-за укорочения видимой площади граней поверхности. Кроме того, модель учитывает сложные геометрические и радиометрические явления, такие как маскирование, затенение и взаимные отражения между гранями. Были проведены несколько экспериментов на образцах шероховатых рассеивающих поверхностей, таких как штукатурка, песок, глина и ткань. Все эти поверхности демонстрируют значительное отклонение от ламбертовского поведения. Полученные измерения коэффициента отражения хорошо согласуются с отражательной способностью, предсказанной моделью.

Оригинал (Eng):

Lambert's model for body reflection is widely used in computer graphics. It is used extensively by rendering techniques such as radiosity and ray tracing. For several real-world objects, however, Lambert's model can prove to be a very inaccurate approximation to the body reflectance. While the brightness of a Lambertian surface is independent of viewing direction, that of a rough surface increases as the viewing direction approaches the light source direction. In this paper, a comprehensive model is developed that predicts body reflectance from rough surfaces. The surface is modeled as a collection of Lambertian facets. It is shown that such a surface is inherently non-Lambertian due to the foreshortening of the surface facets. Further, the model accounts for complex geometric and radiometric phenomena such as masking, shadowing, and interreflections between facets. Several experiments have been conducted on samples of rough diffuse surfaces, such as, plaster, sand, clay, and cloth. All these surfaces demonstrate significant deviation from Lambertian behavior. The reflectance measurements obtained are in strong agreement with the reflectance predicted by

the model.

1.2 Основные формулы

1. Основная формула диффузионного отражения

$$I_d = k_d * I_l * (N * L)^+ \tag{1}$$

где:

I_d — диффузная составляющая освещенности

k_d — диффузный коэффициент (базовый цвет)

N — нормализованная нормаль к поверхности

L — нормализованное направление к источнику света

(NL)+=max(0,NL)- положительная часть скалярного произведения

Полная формула освещения:

Itotal=Ia+(1Ia)IdItotal=Ia+(1Ia)Id

где:

I_a — коэффициент рассеянного света (ambient)

I_total — итоговая интенсивность освещения

```
light_dir = self.light_position - np.mean(face_vertices, axis=0)
light_dir_norm = np.linalg.norm(light_dir)
if light_dir_norm > 1e-8:
    light_dir = light_dir / light_dir_norm

normal_norm = np.linalg.norm(normal)
if normal_norm > 1e-8:
    normal = normal / normal_norm

dot_product = np.dot(normal, light_dir)
intensity = self.ambient_light + (1.0 - self.ambient_light) * max(0.0, dot_product)
intensity = min(1.0, max(0.0, intensity))
```

Математический смысл:

Физическая основа: Закон Ламберта описывает рассеянное отражение от матовой поверхности

2. Матрицы вращения (Rotation Matrices)

Формулы:

Вращение вокруг оси X:

$$R_x() = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos() & -sin() \\ 0 & sin() & cos() \end{bmatrix} \tag{2}$$

Вращение вокруг оси Y:

$$R_y() = \begin{bmatrix} cos() & 0 & sin() \\ 0 & 1 & 0 \\ -sin() & 0 & cos() \end{bmatrix} \tag{3}$$

Вращение вокруг оси Z:

$$R_z() = \begin{bmatrix} cos() & -sin() & 0 \\ sin() & cos() & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Комбинированная матрица вращения:

$$R = R_z R_y R_x \tag{5}$$

Реализация в коде:

```
def create_rotation_matrix(self):
    rx = np.array([
        [1, 0, 0],
        [0, np.cos(self.angle_x), -np.sin(self.angle_x)],
        [0, np.sin(self.angle_x), np.cos(self.angle_x)]
    ], dtype=np.float32)

    ry = np.array([
        [np.cos(self.angle_y), 0, np.sin(self.angle_y)],
        [0, 1, 0],
        [-np.sin(self.angle_y), 0, np.cos(self.angle_y)]
    ], dtype=np.float32)

    rz = np.array([
        [np.cos(self.angle_z), -np.sin(self.angle_z), 0],
        [np.sin(self.angle_z), np.cos(self.angle_z), 0],
        [0, 0, 1]
    ], dtype=np.float32)

    return rz @ ry @ rx
```

Математический смысл:

1. Преобразование координат точек при вращении объекта в 3D-пространстве
2. Последовательное применение вращений вокруг трех осей
3. Использование однородных координат для линейных преобразований

3. Перспективная проекция (Perspective Projection)

$$x_{proj} = \frac{fx}{z + d} + \frac{width}{2} \tag{6}$$

$$y_{proj} = \frac{fy}{z+d} + \frac{height}{2} \tag{7}$$

где:

x,y,z — 3D координаты точки

f — фокусное расстояние (параметр проекции)

d — расстояние до камеры

width,height — размеры холста

```
@njit(parallel=True, fastmath=True)
def project_vertices_jit(vertices, camera_distance, fov, width, height):
    n = vertices.shape[0]
    projected = np.empty((n, 2))
    for i in prange(n):
        x = vertices[i, 0]
        y = vertices[i, 1]
        z = vertices[i, 2] + camera_distance

        if abs(z) > 1e-8:
            f = fov / z
            projected[i, 0] = f * x + width / 2
            projected[i, 1] = f * y + height / 2
        else:
            projected[i, 0] = width / 2
            projected[i, 1] = height / 2
    return projected
```

Математический смысл:

1. Имитация перспективы человеческого зрения

2. Точки, находящиеся дальше от камеры, проецируются меньше

3. Деление на zz создает эффект перспективного искажения

4. Сложение с половиной ширины/высоты центрирует изображение

4. Параметрические уравнения для генерации фигур

Сфера:

$$\begin{aligned} x &= rsin(\theta)cos(\phi) \\ y &= rsin(\theta)sin(\phi) \\ z &= rcos(\theta) \end{aligned} \tag{8}$$

где:

r — радиус сферы

— полярный угол [0,][0,] (от северного до южного полюса)

— азимутальный угол [0,2][0,2] (вокруг экватора)

Тор (тороид):

$$
\begin{aligned}
x &= (R + rcos(v))cos(u) \\
y &= (R + rcos(v))sin(u) \\
z &= rsin(v)
\end{aligned} \tag{9}
$$

где:

R — радиус тора (расстояние от центра до центра трубки)

r — радиус трубки

u — угол вокруг тора [0,2]

v — угол вокруг трубки [0,2]

```
# Генерация сферы
for i in range(lat_segments + 1):
    theta = i * np.pi / lat_segments
    sin_theta = np.sin(theta)
    cos_theta = np.cos(theta)

    for j in range(lon_segments + 1):
        phi = j * 2 * np.pi / lon_segments
        sin_phi = np.sin(phi)
        cos_phi = np.cos(phi)

        vertices[idx, 0] = sin_theta * cos_phi
        vertices[idx, 1] = sin_theta * sin_phi
        vertices[idx, 2] = cos_theta
        idx += 1

# Генерация тора
for i in range(u_segments + 1):
    u = i * 2 * np.pi / u_segments
    cos_u = np.cos(u)
    sin_u = np.sin(u)

    for j in range(v_segments + 1):
        v = j * 2 * np.pi / v_segments
        cos_v = np.cos(v)
        sin_v = np.sin(v)

        vertices[idx, 0] = (R + r * cos_v) * cos_u
        vertices[idx, 1] = (R + r * cos_v) * sin_u
        vertices[idx, 2] = r * sin_v
        idx += 1
```

Математический смысл:

1. Сферические координаты для сферы обеспечивают равномерное распределение вершин

2. Тороидальные координаты для тора позволяют точно описать поверхность вращения

3. Параметрическое представление обеспечивает гладкость поверхности при достаточной детализации

5. Нормализация векторов (Vector Normalization)

Формула:

$$V_{norm} = \frac{V}{||V||} = \frac{V}{\sqrt{V_x^2 + V_y^2 + V_z^2}} \tag{10}$$

```
@njit(fastmath=True)
def normalize_jit(v):
    norm = np.sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2])
    if norm < 1e-8:
        return np.array([0.0, 0.0, 1.0])
    return v / norm
```

Математический смысл:

1. Преобразование вектора в единичный вектор (длиной 1)
2. Сохранение направления вектора при изменении длины
3. Необходима для корректного вычисления скалярного произведения в формуле Ламберта
4. Предотвращение деления на ноль при нулевой длине вектора

6. Скалярное произведение (Dot Product)

$$AB = A_x B_x + A_y B_y + A_z B_z = ||A||||B||cos(\theta) \tag{11}$$

```
@njit(fastmath=True)
def dot_product_jit(a, b):
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2]
```

Математический смысл:

1. Вычисление косинуса угла между двумя векторами
2. Проекция одного вектора на другой
3. Ключевая операция в формуле освещения Ламберта
4. Эффективный способ определения ориентации поверхностей относительно источника света

7. Расчет цвета (Color Calculation)

$$\begin{aligned} C_{result} &= C_{base} I_{total} \\ R &= LC_{result}[0]255 \\ G &= LC_{result}[1]255 \\ B &= LC_{result}[2]255 \end{aligned} \tag{12}$$

```
color = self.current_color * intensity
r = int(min(255, max(0, color[0] * 255)))
g = int(min(255, max(0, color[1] * 255)))
b = int(min(255, max(0, color[2] * 255)))
fill_color = f'#{r:02x}{g:02x}{b:02x}'
```

Математический смысл:

1. Модуляция базового цвета интенсивностью освещения

2. Линейное преобразование из диапазона [0,1] в [0,255]

3. Ограничение значений в допустимом диапазоне

4. Преобразование в формат HEX для отображения в tkinter

8. Z-сортировка (Depth Sorting)

Формула расчета глубины грани:

$$depth_{face} = \frac{1}{n} \sum_{i=1}^{N} z_i \tag{13}$$

Применение в коде:

```
@njit(parallel=True, fastmath=True)
def calculate_face_depths_jit(rotated_vertices, faces):
    n_faces = faces.shape[0]
    depths = np.empty(n_faces)
    for i in prange(n_faces):
        face = faces[i]
        z_sum = 0.0
        count = 0
        for j in range(len(face)):
            if face[j] < rotated_vertices.shape[0]:
                z_sum += rotated_vertices[face[j], 2]
                count += 1
        depths[i] = z_sum / count if count > 0 else 0.0
    return depths

# Сортировка граней по глубине
sorted_indices = np.argsort(-face_depths)  # От дальних к ближним
```

Математический смысл:

1. Вычисление средней глубины (Z-координаты) всех вершин грани

2. Сортировка граней по убыванию глубины для правильного наложения

3. Эмуляция Z-буферизации без использования аппаратного ускорения

4. Обеспечение правильной видимости непрозрачных поверхностей

9. Динамические уровни детализации (LOD — Level of Detail)

Формула адаптивной детализации:

$$detail_{optimal} = \begin{cases} max(detail_{min}, & \text{if FPS} < 20 \\ min(detail_{max}, & \text{if FPS} > 40 \\ detail_{current}, & \text{otherwise} \end{cases} \tag{14}$$

```
def get_optimal_detail(self):
    current_time = time.time()
    if current_time - self.last_lod_update > 1.0:
        self.last_lod_update = current_time

        if self.current_primitive == "sphere":
            base_detail = self.sphere_detail_var.get()
        elif self.current_primitive == "torus":
```

```
            base_detail = self.torus_detail_var.get()
        else:
            base_detail = 20

        if hasattr(self, 'current_fps') and self.current_fps < 20:
            return max(self.min_detail, base_detail // 2)
        elif hasattr(self, 'current_fps') and self.current_fps > 40:
            return min(self.max_detail, base_detail + 5)
        else:
            return base_detail
```

Математический смысл:

1. Адаптация сложности геометрии в реальном времени
2. Поддержание стабильной частоты кадров
3. Баланс между визуальным качеством и производительностью
4. Плавное изменение детализации для минимизации визуальных артефактов

10. Адаптивная скорость анимации

Формула:

$$speed_{factor} = min(1.0, \frac{FPS_{current}}{30.0}) \\ \theta = base_{speed} speed_{factor} \tag{15}$$

Применение в коде:

```
if hasattr(self, 'current_fps'):
    speed_factor = min(1.0, self.current_fps / 30.0)
    self.angle_x += 0.005 * speed_factor
    self.angle_y += 0.01 * speed_factor
    if self.current_primitive in ["sphere", "torus"]:
        self.angle_z += 0.003 * speed_factor
```

Математический смысл:

1. Плавное замедление анимации при падении производительности
2. Сохранение визуальной плавности при разных уровнях детализации
3. Адаптация к возможностям аппаратного обеспечения
4. Улучшение пользовательского опыта при работе на слабых устройствах

2 РЕАЛИЗАЦИЯ ОТОБРАЖЕНИЯ ПРИМИТИВОВ

2.1 Куб

Куб задан 8 вершинами и 6 гранями. Для его вращения используется матрица вращения. В отличие от сферы и тора, у куба нормали вращаются отдельно, так как они постоянны для каждой грани и не зависят от позиции вершин.

Также при отображении куба используются плоские грани, что означает: что каждая грань имеет одну постоянную нормаль, что вся грань освещается одинаково, четкие границы между гранями, более высокую производительность по сравнению со сглаженным освещением.

2.2 Сфера

Сфера задаётся при помощи параметрических уравнений в полярной системе координат. Далее при генерации граней каждый прямоугольный сегмент сферы разбивается на 2 полигона (треугольника).

Также важным отличием от куба, является то, что у сферы отсутствуют предзаданные нормали, и они вычисляются динамически.

Освещение сферы проявляет следующие особенности: градиентное освещение (плавный переход от светлых к темным участкам), фоновое освещение (ambient_light = 0.2 обеспечивает минимальную видимость даже на теневых участках), динамическое освещение (Интенсивность зависит от угла между нормалью и источником света), реалистичные блики (Максимальная яркость там, где нормаль направлена точно к источнику света). то есть сфера освещается при помощи (smooth shading).

Исходя из вышесказанного можно понять, что сфера является самым ресурсоёмким примитивом.

2.3 Тор

Тор является наиболее сложным с точки зрения математики для отображения примтивом.

Тор генерируется при помощи параметрических уравнений:

```
vertices[idx, 0] = (R + r * cos_v) * cos_u
vertices[idx, 1] = (R + r * cos_v) * sin_u
vertices[idx, 2] = r * sin_v
```

Где:

R — радиус тора (расстояние от центра тора до центра трубки)

r — радиус трубки (толщина тора)

u — угол вращения вокруг оси Z (0 до 2), определяет положение вдоль «большого круга»

v — угол вращения вокруг центра трубки (0 до 2), определяет положение вдоль «малого круга»

Внутренние и внешние части тора:

Внешние вершины: Когда cos_v > 0, вершины находятся на внешней стороне тора

Внутренние вершины: Когда cos_v < 0, вершины находятся на внутренней стороне (ближе к центральной оси)

Верхние/нижние вершины: Когда sin_v = ±1, вершины находятся в верхней или нижней части трубки

Все грани генерируются как полигоны (треугольники).

Самым сложным в корректном отображении тора — отображение внутренних граней. В отличие от сферы или куба, у тора внутренние и внешние поверхности имеют противоположные направления нормалей.

Для этого для внутренних граней: находится центр трубки (для каждой точки на поверхности тора вычисляется центр соответствующего сечения трубки в плоскости XY), направление нормали (нормаль рассчитывается как вектор от центра трубки к точке на поверхности), для внутренних граней (когда точка находится на внутренней стороне тора (ближе к центральной оси), вектор v0 — tube_center будет направлен внутрь тора, что обеспечивает правильное освещение внутренней поверхности).

Процесс рендеринга внутренних граней:

1. Вращение: Все вершины (включая внутренние) вращаются с помощью матрицы вращения

2. Проекция: Трехмерные координаты проецируются на двумерную плоскость

3. Сортировка по глубине: Грани сортируются по глубине для правильного наложения (что особенно важно для тора, так как внутренние грани могут быть скрыты внешними)

4. Расчет освещения: Для каждой грани (включая внутренние) рассчитывается освещение на основе нормали

5. Особенность внутреннего освещения: Внутренние грани тора получают меньше прямого света, так как их нормали направлены внутрь, и они частично затенены внешними гранями.

Визуальные особенности внутренних граней

1. Затенение: Внутренние грани обычно темнее внешних из-за геометрии нормалей

2. Видимость: При определенных углах обзора внутренние грани могут быть полностью или частично скрыты внешними

3. Глубина: Внутренние грани имеют большую Z-координату (дальше от камеры), что влияет на порядок отрисовки

4. Кривизна: Внутренние грани имеют более резкую кривизну, что требует большей детализации для плавного отображения

Из этого видно, как генерируется тор и в чём заключалась основная сложность его отображения.

ЗАКЛЮЧЕНИЕ

В ходе проведения работы была разработана программа для отображения трёх встроенных примитивов: куба, шара и тора. В то время как куб, ввиду своей структуры, состоял из малого числа геометрических объектов и достаточно быстро обрабатывался процессором даже на стандартном python без использования оптимизаций (за исключением numpy), то поскольку «идеально» отобразить непрямоугольные фигуры в компьютере не представляется возможным из-за его дискретной природы, то для этого использовались полигоны. В зависимости от их числа менялась «угловатость» фигуры, но вместе с ней и число требуемых полигонов, из-за этого при высокой детализации тор и шар подтормаживают работу программы даже при использовании всех оптимизаций. В будущем можно было бы подкрутить поддержку GPU или переписать на C++, но в этой версии автор ограничился этими методами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Модель Ламберта: https://dl.acm.org/doi/10.1145/192161.192213 [1] — дата обращения (29.10.2025) [1]

Ссылка на гит с программой: https://github.com/Sev23456/3D-engine-Lambert — дата обращения (24.11.2025) [2]

Ссылка на документацию numba и jit: https://numba.pydata.org/numba-doc/dev/user/jit.-html — дата обращения (21.11.2025) [3]

Painter's Algorithm: https://cgg.mff.cuni.cz/xtasciitildepepca/lectures/pdf/pg1-28-painter-en-.pdf — дата обращения (22.11.2025) [4]

Документация numpy (недоступна в РФ, пробуйте VPN или Tor): https://numpy.org/ — дата обращения (04.11.2025) [5]

Документация tkinter: https://docs.python.org/3/library/tkinter.html — дата обращения (01.11.2025) [6]