Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет имени
Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему
«Моделирование экранируемых и собственных теней в реальном времени»

Студент Зевахин Михаил Евгеньевич

Руководитель Строганов Ю.В.

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель – разработать программу, визуализирующую сцену, содержащую плоскость, трехмерный объект и один источник света, порождающий тень, падающую на плоскость (*экранируемую*) и на объект, отбрасывающий тень (*собственную*).

Задачи:

1) описать и выбрать алгоритмы построения собственных и экранируемых теней,
2) выбрать алгоритм модели освещения,
3) детерминировать общие входные параметры алгоритмов,
4) реализовать выбранные алгоритмы построения теней,
5) разработать программу, демонстрирующую отображение собственных и экранируемых теней,
6) провести исследование разработанных реализаций алгоритмов.

1 Аналитический раздел

1.1 Модель освещения

В этой части раздела описаны основные идеи Ламбертового освещения [4].

Диффузное освещение

Модель описывает равномерное рассеивание света от поверхности во всех направлениях. Яркость поверхности зависит от угла падения света:

$$I_{\text{diff}} = \max \begin{cases} \vec{L} \cdot \vec{N} \\ 0 \end{cases}, \tag{1.1}$$

где $\vec{L}$ — направление от фрагмента к источнику света, а $\vec{N}$ — нормаль к поверхности.

Уменьшение интенсивности с расстоянием

Интенсивность света уменьшается с увеличением расстояния между источником и фрагментом:

$$A = \max \begin{cases} 1 - \frac{d}{R} \\ 0 \end{cases}, \tag{1.2}$$

где d — расстояние до источника, а R — радиус действия света.

Итоговая интенсивность и цвет освещения

Цвет и интенсивность освещения зависят от мощности источника ($Intensity$), его цвета ($Color$), а также от угла падения I_{diff} – (1.1) и расстояния A – (1.2):

$$\text{LightColor} = \text{Color} \cdot \text{Intensity} \cdot A \cdot I_{\text{diff}}, \tag{1.3}$$

1.2 Типы источников света

1.2.1 Точечный источник света

Точечный источник света «Point Light» испускает свет равномерно во всех направлениях из одной точки.

Такой источник света описывается следующими параматрами:

- **позиция**: $\vec{P}$ – координаты источника света,
- **радиус действия**: R – ограничивает зону влияния света,
- **цвет**: определяет оттенок света,
- **интенсивность**: задаёт силу света.

Расчет освещения для точечного источника света вычисляется по формулам (1.1 – 1.3).

1.2.2 Прожекторный источник света

Прожекторный источник света «Spot Light» испускает свет в определённом направлении, ограниченном конусом.

Такой источник света описывается следующими параматрами:

- **позиция**: $\vec{P}$ — координаты источника света,
- **направление**: $\vec{D}$ — вектор направления света,
- **углы конуса**: θ_{inner} и θ_{outer} — внутренний и внешний углы, задающие форму конуса,
- **радиус действия**: R — зона действия света,
- **цвет**: определяет оттенок света,
- **интенсивность**: сила света внутри конуса.

Расчет освещения для прожекторного источника представлен ниже.

Диффузное освещение

Яркость поверхности, зависящая от угла падения света, вычисляется аналогично алгоритму освещения для точечного источника света, по формуле (1.1).

Уменьшение интенсивности с расстоянием

Однотипно интенсивность света уменьшается с увеличением расстояния между источником и фрагментом по формуле (1.2).

Формирование направления интенсивности света прожектора

Основной шаг алгоритма, ограничивающий большую часть всенаправленного рассеивания точечного источника света и оставляющий только освещенную прожектором область, за счет учета дополнительного коэффициента:

$$\text{spotEffect} = \vec{D} \cdot (-\vec{L}), \tag{1.4}$$

где $\vec{L}$ — направление от фрагмента к источнику света.

Смягчение рассеивания обода прожекторного луча

Для большей реалистичности освещения прожекторным источником света, вводится учет рассеивания обода освещаемой области. Таким образом, интенсивность света уменьшается не только с расстоянием относительно позиции источника света, но и относительно отстояния от направления прожекторного света:

$$\text{spotIntensity} = \min \begin{cases} \max \begin{cases} \frac{\text{spotEffect} - \cos(\theta_{\text{outer}})}{\cos(\theta_{\text{inner}}) - \cos(\theta_{\text{outer}})} \\ 0 \end{cases} \\ 1 \end{cases} \tag{1.5}$$

Итоговое освещение учитывает затухание и ограничение углом:

$$\text{LightColor} = \text{Color} \cdot \text{Intensity} \cdot \text{spotIntensity} \cdot A \cdot I_{\text{diff}}. \tag{1.6}$$

1.3 Алгоритм теневых карт

Эта концепция была представлена Лэнсом Уильямсом в 1978 году в статье под названием «Отбрасывание изогнутых теней на изогнутые поверхности» [1].

Тени создаются путём проверки того, виден ли пиксель из источника света, за счет сравнения пикселя с z-буфером или глубинным изображением источника света, сохранённым в виде текстуры [4].

Его идея заключается в том, чтобы сначала определить, какие области сцены освещены с точки зрения источника света, а затем использовать эту информацию для вычисления теней при рисовании сцены с точки зрения камеры.

Рисование сцены с точки зрения источника света

На этом этапе строится *теневая карта* — текстура глубины, которая хранит расстояния от источника света до ближайших объектов сцены в его проекции.

Проекция сцены выполняется с точки зрения источника света, используя матрицу проекции. Результат сохраняется в текстуру глубины «shadow map».

Преобразование координат фрагмента в пространство источника света

Каждый фрагмент сцены, видимый с точки зрения камеры, должен быть проверен на предмет того, находится ли он в тени. Для этого его мировые координаты преобразуются в пространство источника света:

$$\text{projCoords}_{xy} = \frac{\text{LightSpacePos}_{xyz}}{\text{LightSpacePos}_{w}}. \tag{1.7}$$

Эти координаты нормализуются в диапазон $[0, 1]$:

$$\text{projCoords}_{xy} = \text{projCoords}_{xy} \cdot 0,5 + 0,5. \tag{1.8}$$

Сравнение глубины фрагмента с теневой картой

После преобразования проверяется, является ли текущий фрагмент ближе к источнику света, чем ближайшая точка, записанная в теневой карте:

$$\text{shadow} = \begin{cases} 1, & \text{если } currentDepth < closestDepth + bias \\ 0, & \text{иначе} \end{cases}, \tag{1.9}$$

где:

- closestDepth — глубина, сохранённая в теневой карте,
- currentDepth — глубина текущего фрагмента,
- bias — небольшое смещение для устранения артефактов (погрешностей из-за ошибок вычислений).

closestDepth – вычисляется по формуле:

$$\text{closestDepth} = texture(\text{shadowMap}, \text{projCoords}_{xy}), \tag{1.10}$$

где *texture* - функция чтения из теневой карты по координатам.

Если фрагмент находится дальше, чем значение из теневой карты, то он попадает в тень.

Учет теней при расчёте освещения

Результирующий свет для фрагмента корректируется с учётом теней:

$$\text{lighting} = \text{ambient} + (1 - \text{shadow}) \cdot \text{diffuse}. \tag{1.11}$$

Таким образом, если фрагмент в тени, то только амбиентный свет влияет на его цвет.

1.4 Алгоритм теневых карт с фильтрацией

Алгоритм *Percentage-Closer Filtering* (PCF) – это метод улучшения теней при использовании теневых карт. Он помогает смягчить резкие границы теней, возникающие из-за дискретного характера теневой карты. В основе PCF лежит использование размытия глубины путём усреднения результатов нескольких выборок вокруг текущего фрагмента.

Рисование сцены с точки зрения источника света

Заполнение теневой карты аналогично этапу в стандартном алгоритме.

Преобразование координат фрагмента в пространство источника света

Координаты текущего фрагмента преобразуются в пространство источника света аналогично стандартному алгоритму теневых карт по формулам (1.7) и (1.8).

Несколько выборок из теневой карты

Для применения PCF выполняются несколько выборок глубины вокруг координаты фрагмента в текстуре теневой карты. Берутся выборки в соседних точках:

$$\text{sampleDepth}_i = \text{texture}(\text{shadowMap}, \text{projCoords}.xy + \text{offset}_i). \tag{1.12}$$

Учет каждой выборки

Для каждой выборки проверяется, находится ли фрагмент в тени, аналогично стандартному алгоритму теневых карт по формуле (1.9):

$$\text{shadow}_i = \begin{cases} 1, & \text{если } currentDepth < sampleDepth_i + bias \\ 0, & \text{иначе} \end{cases}. \tag{1.13}$$

Усреднение теневых значений

Результирующее значение тени — это среднее значение всех выборок:

$$\text{shadowFactor} = \frac{1}{N} \sum_{i=1}^{N} \text{shadow}_i, \tag{1.14}$$

где N — количество выборок.

Корректировка освещения с учетом PCF

Окончательное освещение рассчитывается с учётом результата PCF:

$$\text{lighting} = \text{ambient} + (1 - \text{shadowFactor}) \cdot \text{diffuse}. \tag{1.15}$$

1.5 Алгоритм теневых карт с фильтрацией шумом

Шумовая фильтрация «Noise Filtering» используется для смягчения теней и устранения артефактов из-за разрывов или резких переходов на теневой карте. Вместо регулярного фильтра (как «PCF»), в данном подходе добавляются случайные смещения, что создаёт эффект дезориентированных теней и уменьшает эффект лесенки «aliasing». Эти случайные смещения генерируются с помощью шума, часто с использованием случайных или Гауссовых распределений.

Подготовка данных

Рисование сцены с точки зрения источника света и преобразование координат фрагмента в пространство источника света происходят аналогично стандартному алгоритму теневой карты.

Генерация случайного числа

$$f(\vec{v}) = fract(\sin(\vec{v} \cdot \vec{c}) \cdot k), \tag{1.16}$$

где:

- $fract(x)$ – функция, возвращающая дробную часть числа,
- $\vec{v}$ – вектор, для которого вычисляется случайный вектор,
- $\vec{c}$ – вектор-константа, подбирается случайно,
- k – число для нормализации значения, как правило, большое, подбирается случайно.

Генерация Гауссового распределения

Для добавления случайного смещения используется метод Бокса-Мюллера для генерации чисел с Гауссовым распределением.

$$r = \sqrt{-2 \ln u_1} \cdot \cos(2\pi u_2), \tag{1.17}$$

где u_1 и u_2 – случайные числа в диапазоне $[0, 1]$, компоненты вектора, полученного в результате функции (1.16).

Итоговое смещение

$$\text{noiseOffset} = (r_1, r_2) \cdot \sigma, \tag{1.18}$$

где:

- r_1 и r_2 – независимые случайные числа с Гауссовым распределением,
- σ – стандартное отклонение (ширина разброса).

Фильтрация

Этапы алгоритма фильтрации аналогичны шагам в PCF, за исключением того, что вместо $offest$ берется переменая $noiseOffset$ для формулы (1.12).

1.6 Алгоритм мягких теневых карт с фильтрацией

Алгоритм PCSS (Percentage-Closer Soft Shadows) – это улучшенный метод теневого сглаживания, который моделирует мягкие тени. В отличие от стандартного PCF, который использует фиксированный радиус выборок, PCSS динамически определяет радиус на основе расстояния от источника света и близлежащих блокирующих объектов. Это позволяет лучше передавать эффект мягких теней, как в реальной жизни: чем дальше объект от поверхности, тем более размытой становится тень.

Подготовка данных

Рисование сцены с точки зрения источника света и преобразование координат фрагмента в пространство источника света происходят аналогично стандартному алгоритму теневой карты.

Поиск блокирующих объектов

Задача - оценить объекты, которые частично блокируют свет и влияют на размер полутени.

$$avgBlockerDepth = \frac{blockerDepthSum}{numBlockers}, \tag{1.19}$$

где $numBlockers$ – количество выборок, а $blockerDepthSum$ вычисляется по формуле:

$$sum = \sum_{i=1}^{N} \begin{cases} 1, & \text{если } currentDepth_i < sampleDepth_i + bias \\ 0, & \text{иначе} \end{cases}, \tag{1.20}$$

где $sampleDepth_i$ вычисляется аналогично алгоритму PCF по формуле (1.12), за исключением того, что $offset_i$ вычисляется по формуле:

$$offset_i = sampleDisk(i, N) \cdot texelSize \cdot searchRadius, \tag{1.21}$$

где:

- $texelSize$ – размер одного элемента в текстуре теней,

- $searchRadius$ – радиус поиска потенциальных блокирующих объектов,

- $sampleDisk(i, N)$ – генерирация выборки, равномерно распределенной по окружности (1.22).

Формула генерации выборки, равномерно распределенной по окружности:

$$sampleDisk(i, N) = \{\cos(rad); \sin(rad)\}, \tag{1.22}$$

где

$$rad = 2\pi \cdot \frac{i}{N},$$

где

- i – индекс очередной выборки,

- N – общеее количество выборок.

Оценка размера полутени

Задача – определить, насколько велика полутень на основе разницы глубин между фрагментом и блокирующими объектами.

Видовая глубина фрагмента и блокирующего объекта:

$$viewLightFragDepth = \frac{P_y}{2 \cdot currentDepth - P_x - 1}, \tag{1.23}$$

$$viewLightBlockerDepth = \frac{P_y}{2 \cdot avgBlockerDepth - P_x - 1}, \tag{1.24}$$

где

$$P_x = \frac{n + f}{f - n}$$

и

$$P_y = -\frac{2nf}{f - n},$$

где n – ближняя проекционная плоскость отсечения, f – дальняя проекционная плоскость отсечения.

Оценка размера полутени:

$$penumbraSize = \frac{viewLightFragDepth - viewLightBlockerDepth}{viewLightBlockerDepth} \cdot R, \quad (1.25)$$

где R – радиус действия источника света.

Чем больше разница между глубинами, тем шире становится полутень.

Применение PCF с изменяемым радиусом

Задача – использовать Percentage-Closer Filtering (PCF) для размытия границ теней в зависимости от размера полутени.

$$depth_{xy} = texture(shadowMap, projCoords_{xy} + (x, y) \cdot texelSize \cdot fR), \quad (1.26)$$

где $x \in [1, 2, ..halfSize]$ и $y \in [1, 2, ..halfSize]$, где $halfSize = \frac{samleSize}{2}$, где $samleSize$ – количество выборок.

Итоговое значение для тени:

$$shadow = \frac{1}{c} \cdot \sum_{x=1}^{halfSize} \sum_{y=1}^{halfSize} \begin{cases} 1, & currentDepth - bias > depth_{xy} \\ 0, & \text{иначе} \end{cases}, \quad (1.27)$$

где $c = (2 \cdot halfSize + 1)^2$

Итоговый расчет тени

Влияние тени учитывается аналогично формуле (1.15) в алгоритме стандартной фильтрации PCF, за исключением того, что вместо переменной $shadowFactor$ берется значение вычисленное с учетом полутенени – $shadow$.

1.7 Алгоритм мягких теневых карт с фильтрацией шумом

Данный алгоритм является гибридным и сожержит в себе идеи из сразу нескольких: стандартного алгоритма теневых карт, алгоритма теневых карт с фильтрацией шумом - и является расширением алгоритма мягких теневых карт (PCSS).

Все этапы алгоритма будут идентичны шагам в алгоритме мягких теней PCSS, за исколючением того, что в формулах, использующих $offset_i$, вместо вычисления равномерно расположенной позиции на окружноасти будет использоваться вариант Гауссового распределения (формулы: 1.16 - 1.18).

1.8 Параметры алгоритмов теневых карт

В описаных выше алгоритмах теневых карт выделяются общие параметры:

1) параметры в этапе заполнения теневой карты:
   - **view**: матрица вида источника света,
   - **projection**: матрица проекции источника света,
   - **model**: матрица модели объекта,
2) параметры в заключительном этапе рисования сцены:
   - **view**: матрица вида источника света,
   - **projection**: матрица проекции источника света,
   - **view**: матрица вида камеры пользователя,
   - **projection**: матрица проекции камеры пользователя,
   - **model**: матрица модели объекта,
   - **shadowBias**: корректирующее смещение.

Помимо приведенных выше параметров для алгоритмов с фильтрацией используется специфичный параметр **pcfRadius**.

Вывод

В данном разделе была выбрана модель освещения, были теоретически разобраны алгоритмы теневых карт и описаны поддерживаемые источники света в сцене, структурированы параметры источников освещения в Ламбертовой модели, параметры алгоритмов теневых карт.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов.

2.1 Схемы алгоритмов модели освещения

Схемы алгоритмов модели освещения представлены ниже.

1) Схема алгоритма Ламбертового освещения для точечного источника света (рисунок 2.1).
2) Схема алгоритма Ламбертового освещения для прожекторного источника света (рисунок 2.2).

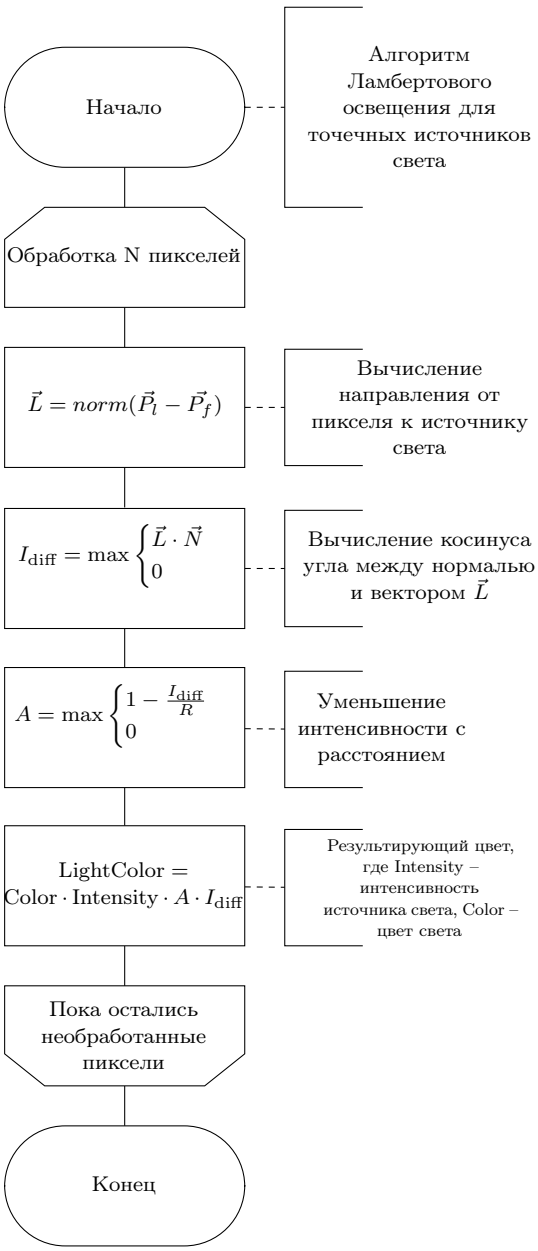

Рисунок 2.1 – Схема алгоритма Ламбертового освещения для точечного источника света

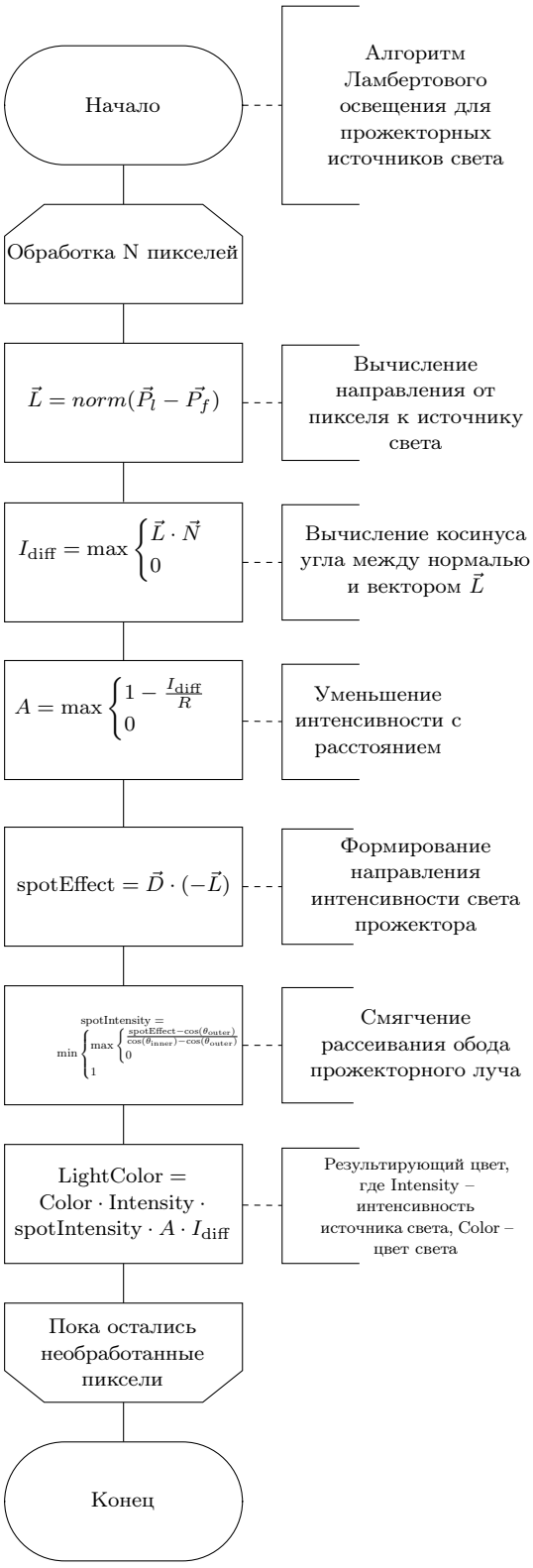

Рисунок 2.2 – Схема алгоритма Ламбертового освещения для прожекторного источника света

2.2 Схемы алгоритмов теневых карт

Схемы алгоритмов теневых карт представлены ниже.

1) Схема алгоритма теневых карт (рисунок 2.3).
2) Схема алгоритма теневых карт с фильтрацией (PCF) (рисунок 2.4).
3) Схема алгоритма теневых карт с фильтрацией шумом (NOISE) (рисунок 2.5).
4) Схема алгоритма мягких теневых карт с фильтрацией (PCSS) (рисунок 2.6).
5) Схема алгоритма мягких теневых карт с фильтрацией шумом (PCSS-NOISE) (рисунок 2.7).

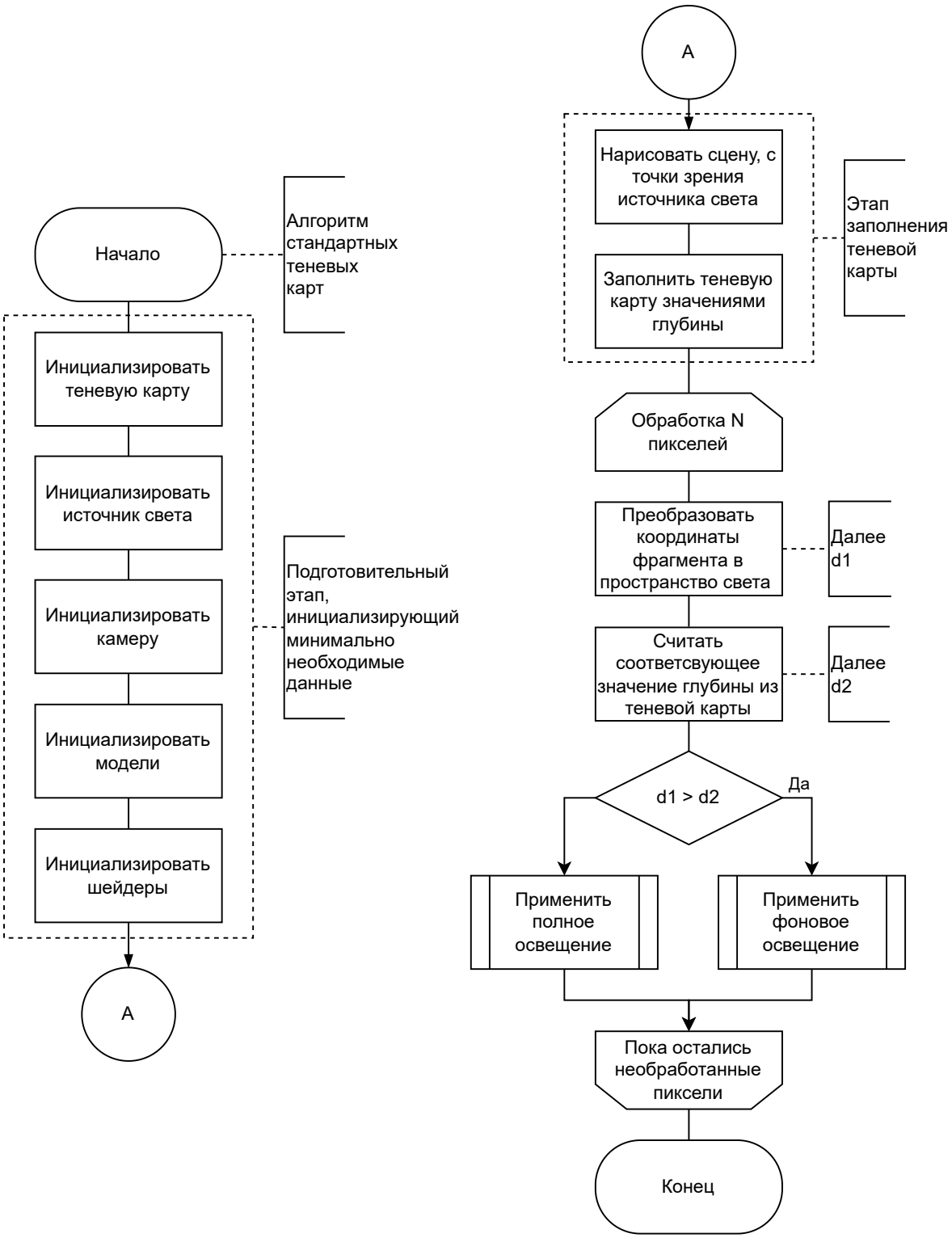

Рисунок 2.3 – Схема алгоритма теневых карт

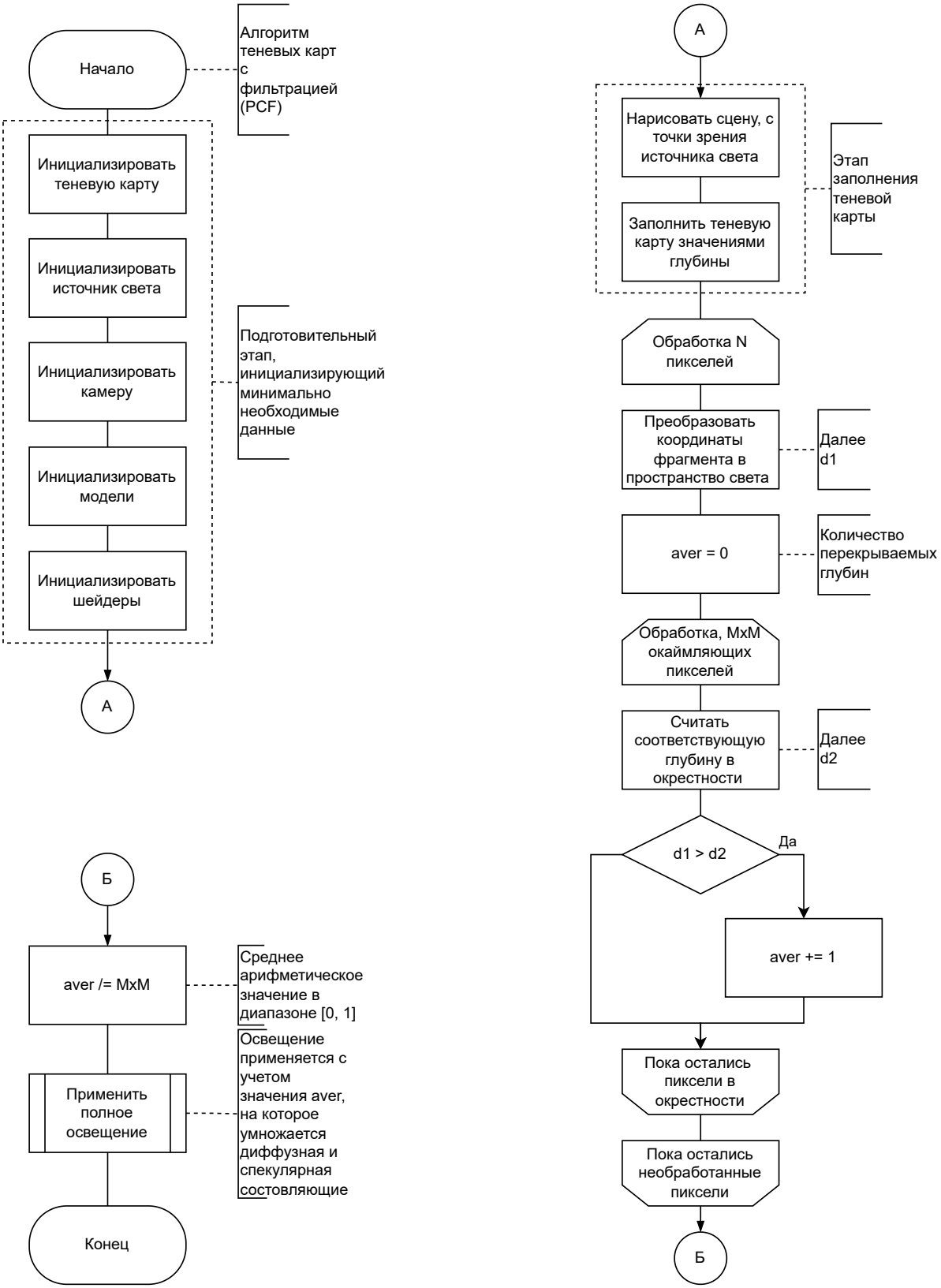

Рисунок 2.4 – Схема алгоритма теневых карт с линейной фильтрацией (PCF)

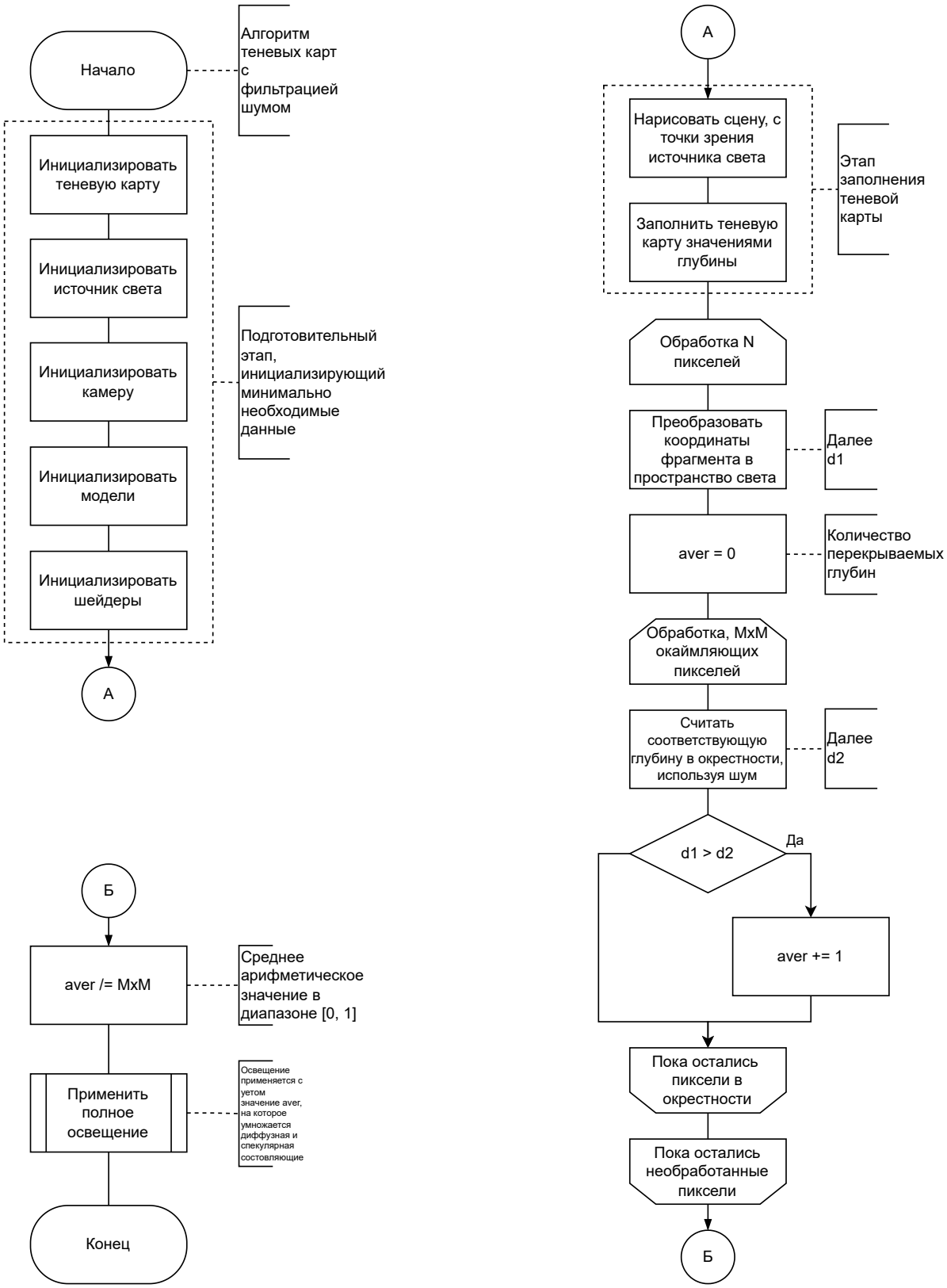

Рисунок 2.5 – Схема алгоритма теневых карт с линейной фильтрацией шумом (NOISE)

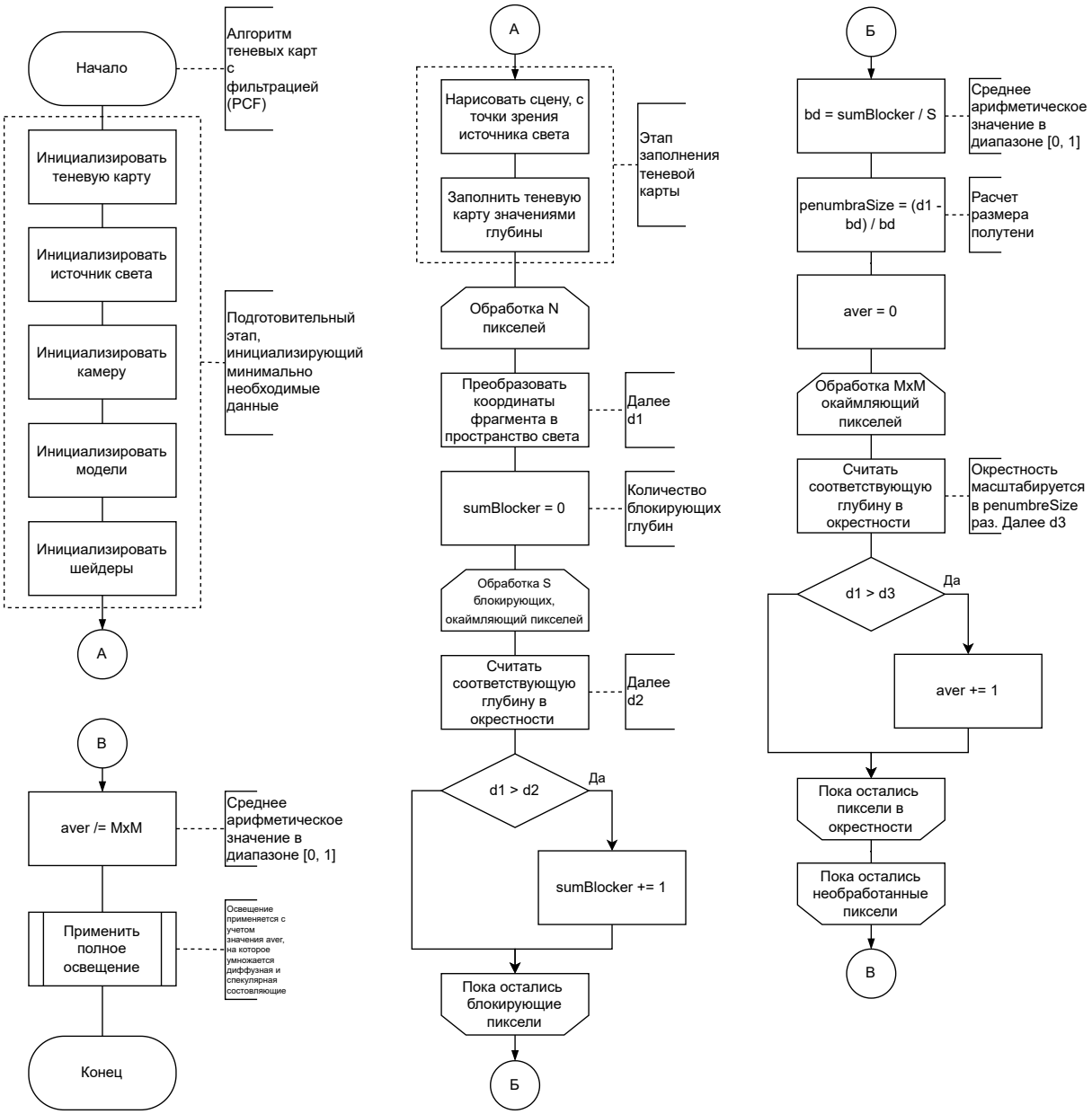

Рисунок 2.6 – Схема алгоритма мягких теневых карт с фильтрацией (PCSS)

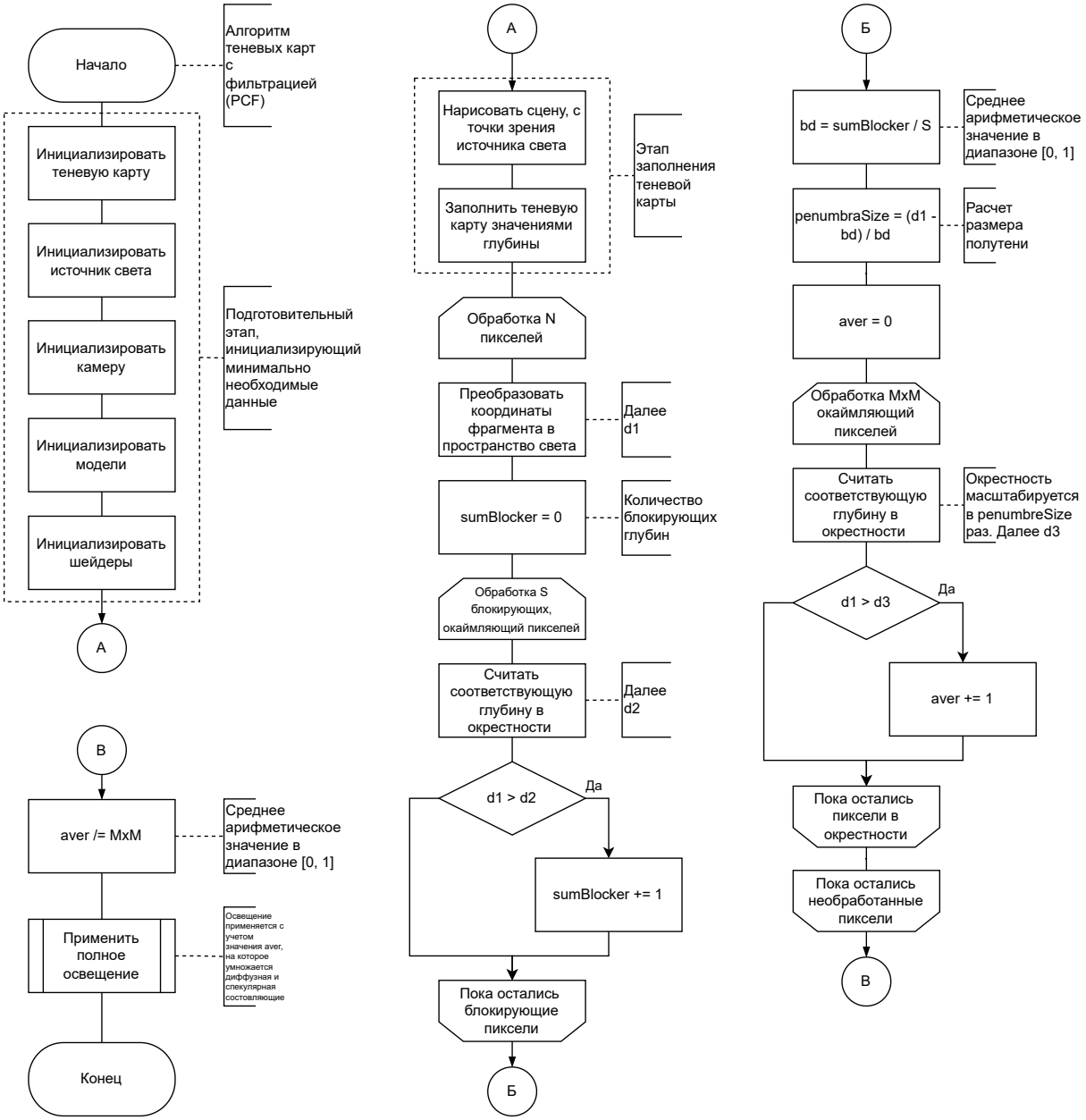

Рисунок 2.7 – Схема алгоритма мягких теневых карт с фильтрацией шумом (PCSS-NOISE)

Вывод

В данном разделе были представлены схемы алгоритмов модели освещения, поддерживающей два источника света: точечный и прожекторный – а также схемы алгоритмов теневых карт.

3 Технологический раздел

В данном разделе описаны используемые языки программирования и среда разработки.

3.1 Выбор языка программирования и среды разработки

Исследование реализаций алгоритмов будет проведено на операционной системе «Windows». Поэтому для создания оконного приложения используется интерфейс прикладного программирования Win32 API. Это набор функций для создания программ, работающих под управлением Microsoft Windows 98, Windows NT или Windows 2000. Все функции этого набора являются 32-битными, что отражено в названии интерфейса [2].

Инициализация, заполнение, обновление и использование теневых карт предполагается в шейдерных программах. Для этого требуется инициализация контекста *«OpenGL»* и загрузка необходимых расширений, позволяющих компилировать такие программы. Для того, чтобы связать контекст и оконное приложение на операционной системе «Windows» требуется непосредственно сама библиотека *«opengl32.lib»* [6].

В ходе разработки реализации алгоритмов теневых карт потребуется использование матричных преобразований. Они должны не только позволять производить непосредственно сами преобразования, но и удовлетворять требованиям использования в шейдерных программах. А именно их представление в памяти должно соответсвовать ожиданиям шейдеров. Для этого отдельно разработан модуль линейной алгебры [9].

Таким образом, в качестве среды программирования выбрана программа Visual Studio Code. Основным языком программирования всего приложения выбран C++. Для разработки модуля матричных преобразований выбран язык C, который наибольшим образом схож с языком программирования шейдерных программ – GLSL.

3.2 Выбор графического API

Для реализации алгоритмов рисования экранируемых и собственных теней в реальном времени был выбран графический API, предоставляющий

работу с графическими ускорителями (рисунок 3.1) [6].

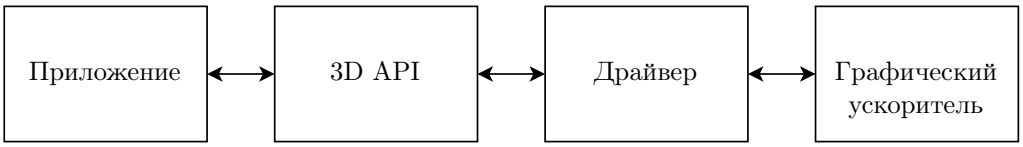

Рисунок 3.1 – Схема взаимодействия приложения с графическим ускорителем через API.

3.2.1 OpenGL

OpenGL, описанный в книге Гинсбурга и Пурномо [5], представляет собой высокоуровневый API, который ориентирован на удобство разработки и поддержку кроссплатформенных приложений.

Основные достоинства OpenGL.

1) **Кроссплатформенность**. OpenGL поддерживается практически на всех операционных системах, включая Windows, macOS и Linux. Эта универсальность делает его предпочтительным выбором для приложений, которые должны работать на различных устройствах.
2) **Простота разработки**. Благодаря высокоуровневой архитектуре, OpenGL позволяет быстро разрабатывать графические приложения, абстрагируя сложные операции взаимодействия с GPU. Как отмечает Боресков в книге «Расширения OpenGL» [6], даже расширенные возможности API остаются интуитивно понятными благодаря наличию хорошо документированных интерфейсов.
3) **Совместимость с современными графическими возможностями**. OpenGL 4.5 поддерживает такие функции, как буферы кадров (FBO), использование шейдеров и текстурные массивы, что делает его подходящим для реализации современных алгоритмов рендеринга.

3.2.2 Vulkan

Вулкан, описанный в книге Селлерса [8], представляет собой низкоуровневый API, который предоставляет разработчику максимальный контроль над процессами работы с графическим оборудованием.

Основные достоинства Vulkan.

1) **Высокая производительность**. Vulkan позволяет эффективно распределять ресурсы GPU и управлять многопоточностью, что делает его подходящим для ресурсоемких приложений.

2) **Гибкость и контроль**. Разработчик полностью контролирует управление памятью, синхронизацию потоков и рендеринг.

3) **Сложность разработки**. Селлерс отмечает, разработка на Vulkan требует значительно большего объема кода и глубокой экспертизы в графических технологиях. Это делает его менее подходящим для задач, где требуется быстрая реализация.

3.2.3 DirectX

DirectX, описанный Горнаковым [7], является API, ориентированным на Windows-платформы.

Основные достоинства DirectX.

1) **Высокая производительность**. DirectX обеспечивает эффективное взаимодействие с GPU, что делает его оптимальным выбором для игр и других графических приложений, работающих на Windows.

2) **Глубокая интеграция с Windows**. DirectX максимально использует возможности операционной системы и оборудования, обеспечивая стабильную производительность.

3) **Ограниченная платформенная поддержка**. DirectX не поддерживается на macOS и Linux.

3.3 Исходные модули программы

Программа для удобства разделена на модули:

1) модуль «LA» - статическая библиотека линейной алгебры, реализующая матричные преобразования, работу с векторами и кватернионами и обеспечивающая выравнивание данных в памяти, ожидаемое в шейдерных программах;

2) модуль «WINAPI» - статическая библиотека, обеспечивающая создание оконного приложения с поддержкой инициализации контекста *«OpenGL»* (версией 4.6) и их совместное связывание;

3) модуль «GLSL» - статическая библиотека, предоставляющая компилирование, линкование, анализ шейдерных программ, оптимизирует рутинные действия при работе с шейдерами;

4) модуль «ShadowMap» - статическая библиотека, в которую собраны исследуемые реализации алгоритмов теневых карт;

5) модуль «app» является основным модулем всего приложения и связывает все модули, упомянутые выше, в единое целое.

3.3.1 Исходные файлы модуля «LA»

В данной библиотеке реализованы такие математические объекты как:

— вектор с 2-мя компонентами,

— вектор с 3-мя компонентами,

— вектор с 4-мя компонентами,

— кватернион,

— матрица размерностью 2 на 2,

— матрица размерностью 3 на 3,

— матрица размерностью 4 на 4.

Файлы модуля представлены ниже:

— **заголовочные** файлы:

1) LA_sup.h – объявление вспомогательных функций для данного модуля;
2) Matrix2D.h – объявление структуры матрицы размерностью 2 на 2 и функций по ее использованию;
3) Matrix3D.h – объявление структуры матрицы размерностью 3 на 3 и функций по ее использованию;
4) Matrix4D.h – объявление структуры матрицы размерностью 4 на 4 и функций по ее использованию;
5) Quaternion.h – объявление структуры кватерниона и функций по его использованию;
6) Vector2D.h – объявление структуры вектора с 2-мя компонентами и функций по его использованию;
7) Vector3D.h – объявление структуры вектора с 3-мя компонентами и функций по его использованию;

8) Vector4D.h – объявление структуры вектора с 4-мя компонентами и функций по его использованию;

— **исходные** файлы:

1) LA_sup.c – реализация вспомогательных функций для данного модуля;
2) Matrix2D.c – реализация матричных функций для размерности 2 на 2;
3) Matrix3D.c – реализация матричных функций для размерности 3 на 3;
4) Matrix4D.c – реализация матричных функций для размерности 4 на 4;
5) Quaternion.c – реализация функций преобразований, задаваемых кватернионом;
6) Vector2D.c – реализация функций взаимодействия с 2-ух компонентными векторами;
7) Vector3D.c – реализация функций взаимодействия с 3-ех компонентными векторами;
8) Vector4D.c – реализация функций взаимодействия с 4-ех компонентными векторами;

3.3.2 Исходные файлы модуля «WINAPI»

Файлы модуля представлены ниже:

— **заголовочные** файлы:

1) winapi_brush_struct.h – объявление структуры кисти;
2) winapi_brash.h – объявление класса, реализующего инициализацию, использование и освобождение кисти;
3) winapi_char_converter.h – объявление функций конвертации строк;
4) winapi_choose_color_dialog.h – объявление функции вызова диалога выбора цвета;
5) winapi_choose_file_dialog.h – объявление функции вызова диалога выбора файла;
6) winapi_common.h – объявление вспомогательных функций, специфичных для данного модуля;
7) winapi_console.h – объявление класса, обеспечивающего создание консоли и перенаправления потоков ввода-вывода;

8) winapi_font_common.h – объявление общих функций работы с шрифтами;
9) winapi_font_struct.h – объявление структуры шрифта;
10) winapi_font.h – объявление класса, обеспечивающего создание, использование и освобождение шрифта;
11) winapi_GLextensions.h – объявление функции загрузки расширений контекста *«OpenGL»*;
12) winapi_GLwindow.h – объявление класса, реализующего создание, использование и освобождение окна, поддерживающего связывание с контекстом *«OpenGL»*;
13) winapi_mat_ext.h – объявление общих расчетных функций, специфичных для данного модуля;
14) winapi_mouse.h – объявление класса мыши, реализующего взаимодействие с вводом мыши;
15) winapi_str_converter.h – объявление расширенных функций конвертации строк;
16) winapi_window.h – объявление класса, реализующего создание, использование и освобождение окна;

— **исходные** файлы:

1) winapi_brash.cpp – реализация класса, обеспечивающего инициализацию, использование и освобождение кисти;
2) winapi_char_converter.cpp – реализация функций конвертации строк;
3) winapi_choose_color_dialog.cpp – реализация функции вызова диалога выбора цвета;
4) winapi_choose_file_dialog.cpp – реализация функции вызова диалога выбора файла;
5) winapi_common.cpp – реализация вспомогательных функций, специфичных для данного модуля;
6) winapi_console.cpp – реализация класса, обеспечивающего создание консоли и перенаправления потоков ввода-вывода;
7) winapi_font_common.cpp – реализация общих функций работы с шрифтами;

8) winapi_font.cpp – реализация класса, обеспечивающего создание, использование и освобождение шрифта;

9) winapi_GLextensions.cpp – реализация функции загрузки расширений контекста *«OpenGL»*;

10) winapi_GLwindow.cpp – реализация класса, обеспечивающего создание, использование и освобождение окна, поддерживающего связывание с контекстом *«OpenGL»*;

11) winapi_mat_ext.cpp – реализация общих расчетных функций, специфичных для данного модуля;

12) winapi_mouse.cpp – реализация класса мыши, обеспечивающего взаимодействие с вводом мыши;

13) winapi_str_converter.cpp – реализация расширенных функций конвертации строк;

14) winapi_window.cpp – реализация класса, обеспечивающего создание, использование и освобождение окна;

3.3.3 Исходные файлы модуля «GLSL»

Файлы модуля представлены ниже:

— **заголовочные** файлы:

1) shader_extensions.h – объявление функций упрощения отправки данных в шейдерные программы;

2) shader.h – объявление класса, обеспечивающего компилирование, линкование и анализ шейдерных программ;

— **исходные** файлы:

1) shader_extensions.cpp – реализация функций упрощения отправки данных в шейдерные программы;

2) shader.cpp – реализация класса, обеспечивающего компилирование, линкование и анализ шейдерных программ;

3.3.4 Исходные файлы модуля «ShadowMap»

Файлы модуля представлены ниже:

— **заголовочные** файлы:

1) DepthBufferGenerator.h – объявление функции создания теневой карты;
2) DepthBufferStruct.h – объявление структуры теневой карты;
3) ShadowMapMainRenderData.h – объявление структуры необходимых данных для стандартного алгоритма теневых карт;
4) ShadowMapPcfRenderData.h – объявление структуры необходимых данных для алгоритма теневых карт с фильтрацией (PCF);
5) ShadowMapNoiseRenderData.h – объявление структуры необходимых данных для алгоритма теневых карт с фильтрацией шумом (NOISE);
6) ShadowMapPcssRenderData.h – объявление структуры необходимых данных для алгоритма мягких теневых карт с фильтрацией (PCSS);
7) ShadowMapPcssNoiseRenderData.h – объявление структуры необходимых данных для алгоритма мягких теневых карт с фильтрацией шумом (PCSS-NOISE);
8) ShadowMap.h – объявление функций стандартного алгоритма теневых карт;
9) ShadowMapPcf.h – объявление функций алгоритма теневых карт с фильтрацией (PCF);
10) ShadowMapNoise.h – объявление функций алгоритма теневых карт с фильтрацией шумом (NOISE);
11) ShadowMapPcss.h – объявление функций алгоритма мягких теневых карт с фильтрацией (PCSS);
12) ShadowMapPcssNosie.h – объявление функций алгоритма мягких теневых карт с фильтрацией шумом (PCSS-NOISE);

— **исходные** файлы:

1) DepthBufferGenerator.cpp – реализация функции создания теневой карты;
2) ShadowMap.cpp – реализация функций стандартного алгоритма теневых карт;
3) ShadowMapPcf.cpp – реализация функций алгоритма теневых карт с фильтрацией (PCF);

4) ShadowMapNoise.cpp – реализация функций алгоритма теневых карт с фильтрацией шумом (NOISE);
5) ShadowMapPcss.cpp – реализация функций алгоритма мягких теневых карт с фильтрацией (PCSS);
6) ShadowMapPcssNosie.cpp – реализация функций алгоритма мягких теневых карт с фильтрацией шумом (PCSS-NOISE);

3.3.5 Исходные файлы модуля «app»

Файлы модуля представлены ниже:

— **заголовочные** файлы:

1) app_args.h – объявление глобальных переменных приложения;
2) user_msgs.h – объявление пользовательских событий;
3) resource.h – объявление индетификаторов элементов оконного приложения;
4) cpu_stop_watch.hpp; – реализация класса таймера;
5) cpu_timing.h – объявление базового класса замера процессорного времени;
6) formater.h – объявление функций конвертации строки в число;
7) FpsSetterDialogProc.h – объявление функции обработки событий диалога установки числа обновления кадров в секунду;
8) general_shadow_options_wnd_proc.h – объявление функции обработки событий окна изменения общих параметров теней;
9) Light.h – объявление функций обновления данных UBO буфера;
10) lighting_wnd_proc.h – объявление функции обработки событий окна изменения параметров источника света;
11) LightStruct.h – объявление структуры, описывающей источник света с выравниванием, требуемым шейдерными программами;
12) main_wnd_proc.h – объявление функции обработки событий родительского окна;
13) ModelLoader.h – объявление функций загрузки трехмерной модели;
14) ModelLoaderDialogProc.h – объявление функции обработки событий диалога выбора загружаемой модели;

15) PlaneMeshGenerator.h – объявление функций создания специфичных примитивных трехмерных объектов;
16) ProjectionEnum.h – объявление перечисления режима проекции;
17) render_wnd_proc.h – объявление функции обработки событий окна отрисовки;
18) ResolutionMapLimit.h – объявление максимальных размеров теневых карт;
19) shadow_wnd_proc.h – объявление функции обработки событий окна держателя вкладок;
20) ShadowAlgEnum.h – объявление перечисления алгоритмов теней;
21) shadowMap_wnd_proc.h – объявление функции обработки событий окна изменения параметров тени;
22) time_definers.h – объявление временных констант;
23) toolbar_wnd_proc.h – объявление функции обработки событий окна инструментов;

— **исходные** файлы:

1) cpu_timing.cpp – реализация базового класса замера процессорного времени;
2) formater.cpp – реализация функций конвертации строки в число;
3) FpsSetterDialogProc.cpp – реализация функции обработки событий диалога установки числа обновления кадров в секунду;
4) general_shadow_options_wnd_proc.cpp – реализация функции обработки событий окна изменения общих параметров теней;
5) Light.cpp – реализация функций обновления данных UBO буфера;
6) lighting_wnd_proc.cpp – реализация функции обработки событий окна изменения параметров источника света;
7) main_wnd_proc.cpp – реализация функции обработки событий родительского окна;
8) main.cpp – точка входы программы;
9) ModelLoader.cpp – реализация функций загрузки трехмерной модели;
10) ModelLoaderDialogProc.cpp – реализация функции обработки событий диалога выбора загружаемой модели;

11) PlaneMeshGenerator.cpp – реализация функций создания специфичных примитивных трехмерных объектов;
12) render_wnd_proc.cpp – реализация функции обработки событий окна отрисовки;
13) shadow_wnd_proc.cpp – реализация функции обработки событий окна держателя вкладок;
14) shadowMap_wnd_proc.cpp – реализация функции обработки событий окна изменения параметров тени;
15) toolbar_wnd_proc.cpp – реализация функции обработки событий окна инструментов;

3.4 Реализация алгоритмов

В данном разделе будут описаны выбранные типы, структуры данных и представлены реализации алгоритмов теней.

3.4.1 Типы и структуры данных

В листинге 3.1 представлена структура, описывающая источник освещения в Ламбертовой модели с учетом выравнивания памяти, ожидаемой в шейдерных программах.

Чтобы передать структуру с ожидаемым расположением полей в структуре в шейдер необходимо следовать следующим правилам.

1) Если элемент представляет собой скаляр, использующий N базовых машинных единиц, базовое выравнивание равно N.
2) Если элемент представляет собой двух- или четырехкомпонентный вектор с компонентами при использовании N базовых машинных единиц базовое выравнивание равно $2N$ или $4N$ соответственно.
3) Если элемент представляет собой трехкомпонентный вектор с компонентами, занимающими N базовых машинных единиц, базовое выравнивание равно $4N$.
4) Если элемент представляет собой массив скаляров или векторов, базовое выравнивание и шаг массива устанавливаются в соответствии с базовым выравниванием отдельного элемента массива в соответствии с правилами

(1), (2) и (3) и округляются в большую сторону к базовому выравниванию четырехкомпонентного вектора. Массив может содержать отступы в начале и конце. Базовое смещение элемента, следующего за массивом, округляется в большую сторону до следующего значения, кратного базовому выравниванию.

5) Если элемент представляет собой матрицу, состоящую из C столбцов и R строк, матрица хранится идентично массиву из C векторов с R компонентами в каждой строке, согласно правилу (4).

6) Если элемент является массивом из S столбцов, то матрица хранится идентично строке векторов-столбцов $S \times C$ с компонентами R в каждом, согласно правилу (4).

7) Если элемент представляет собой матрицу, состоящую из C столбцов и R строк, то матрица сохраняется идентично массиву векторов с R строками с C компонентами, согласно правилу (4).

8) Если элемент представляет собой массив из S матриц, состоящих из R строк и столбцов, то матрица сохраняется идентично строке из $S \times R$ векторов с C компонентами в каждой строке, согласно правилу (4).

9) Если элемент является структурой, базовое выравнивание структуры равно N, где N - наибольшее базовое значение выравнивания для любого из ее элементов, округленное до базового значения выравнивания для четырехкомпонентного вектора. Затем отдельным элементам этой подструктуры присваиваются смещения применяя набор правил рекурсивно, где базовое смещение первого элемента подструктуры равно выровненному смещению структуры. Структура может иметь отступы в конце; базовое смещение элемента, следующего за подструктурой, округляется в большую сторону к следующему кратному базовому выравниванию структуры.

10) Если элемент представляет собой массив из S структур, элементы массива располагаются по порядку в соответствии с правилом (9).

Рациональнее было бы использовать объединение - union, но GLSL его не поддерживает [4].

Листинг 3.1 – Источник освещения

```
1 struct Light
2 {
```

```
    // Кратность 16 ═══════════════════════════════════════════════
    vec3 position;      // Позиция (12 байт)
    float pad0;         // Выравнивание для GLSL vec3 (еще 4 байта)
    // Кратность 16 ═══════════════════════════════════════════════
    vec3 direction;     // Направление (12 байт)
    float pad1;         // Выравнивание для GLSL vec3 (еще 4 байта)
    // Кратность 16 ═══════════════════════════════════════════════
    vec3 color;         // Цвет (12 байт)
    float pad2;         // Выравнивание для GLSL vec3 (еще 4 байта)
    // Кратность 16 ═══════════════════════════════════════════════
    float radius;       // Радиус действия (4 байта)
    float intensity;    // Интенсивность (4 байта)
    float innerCutoff; // Угол (в радианах) внутреннего конуса (4 байта)
    float outerCutoff; // Угол (в радианах) внешнего конуса (4 байта)
    // Кратность 16 ═══════════════════════════════════════════════
    int   type;         // Тип источника света: 0 — PointLight; 1 — SpotLight (4 байта)
    float pad3;         // Дополнительные паддинги,
    float pad4;         // чтобы сделать размер структуры
    float pad5;         // кратным 16 (3 * 4 = 12 байт)
    // Кратность 16 ═══════════════════════════════════════════════
};
```

В листинге 3.2 представлена структура, описывающая необходимые данные для алгоритма стандартных теневых карт.

Листинг 3.2 – Структура параметров для стандартного алгоритма теневых карт

```
struct ShadowMapMainRenderData
{
    DepthBuffer* depthBuffer; // Структура теневой карты
    GLsizei* client_width;    // Ширина клиентской области рисования в пикселях
    GLsizei* client_height;   // Высота клиентской области рисования в пикселях

    Shader* shaderDepthPass;  // Объект класса шейдер
    Shader* shaderRenderPass; // Объект класса шейдер
    Shader* shaderDepthDebug; // Объект класса шейдер

    mat4* view;               // Матрица вида камеры пользователя
    mat4* projection;         // Матрица проекции камеры пользователя

    mat4* lightView;          // Матрица вида источника освещения
    mat4* lightProjection;    // Матрица проекции источника освещения

    GLfloat* shadowBias;      // Корректирующее смещение

    GLuint* quadVAO;          // Идентификатор буфера двумерного объекта, прямоугольник

    GLuint* planeVAO;         // Идентификатор буфера трехмерного объекта, плоскость
    mat4* planeModel;         // Матрица модели этого объекта

    GLuint* modelVAO;         // Идентификатор буфера трехмерного объекта
    GLsizei* modelIndexCount; // Количество индексов в этой модели
    mat4* modelModel;         // Матрица модели этого объекта
};
```

Для всех остальных алгоритмов теневых карт, помимо параметров приведенных выше, требуется еще один параметр - pcfRadius. Таким образом, структура выглядит как в листинге 3.3.

Листинг 3.3 – Структура параметров для алгоритмов теневых карт с фильтрацией

```
struct ShadowMapMainRenderData
{
    DepthBuffer* depthBuffer; // Структура теневой карты
    GLsizei* client_width;    // Ширина клиентской области рисования в пикселях
    GLsizei* client_height;   // Высота клиентской области рисования в пикселях

    Shader* shaderDepthPass;  // Объект класса шейдер
    Shader* shaderRenderPass; // Объект класса шейдер
    Shader* shaderDepthDebug; // Объект класса шейдер

    mat4* view;               // Матрица вида камеры пользователя
    mat4* projection;         // Матрица проекции камеры пользователя

    mat4* lightView;          // Матрица вида источника освещения
    mat4* lightProjection;    // Матрица проекции источника освещения

    GLfloat* shadowBias;      // Корректирующее смещение
    GLfloat* pcfRadius;       // Радиус фильтрации

    GLuint* quadVAO;          // Идентификатор буфера двумерного объекта, прямоугольник

    GLuint* planeVAO;         // Идентификатор буфера трехмерного объекта, плоскость
    mat4* planeModel;         // Матрица модели этого объекта

    GLuint* modelVAO;         // Идентификатор буфера трехмерного объекта
    GLsizei* modelIndexCount; // Количество индексов в этой модели
    mat4* modelModel;         // Матрица модели этого объекта
};
```

В листингах 3.2 – 3.3 использовалась структура теневой карты. Она описана в листинге 3.4.

Листинг 3.4 – Структура теневой карты

```
struct DepthBuffer
{
    GLuint Texture; // Идентификатор карты – текстуры
    GLuint FBO;     // Идентификатор кадрового буфера
    GLsizei width;  // Длина карты в пикселях
    GLsizei height; // Высота карты в пикселях
};
```

3.4.2 Реализация основных этапов алгоритмов теней

Все описанные алгоритмы, как представлено в схемах 2.3 – 2.7, имеют идентичный этап заполнения карты теней (листинг 3.5). В этом участке кода, выполняется заполнение теневой карты, в которую заносится информация глубины объекта.

Листинг 3.5 – Алгоритм заполнения теневой карты

```
void DepthPass(ShadowMapMainRenderData& data)
{
    glViewport(0, 0, data.depthBuffer->width, data.depthBuffer->height);

    glBindFramebuffer(GL_FRAMEBUFFER, data.depthBuffer->FBO);
    glClear(GL_DEPTH_BUFFER_BIT);
```

```
7
8       data.shaderDepthPass->use();
9       uniform_matrix4f(data.shaderDepthPass->get_uniform_location("view"),
            data.lightView);
10      uniform_matrix4f(data.shaderDepthPass->get_uniform_location("projection"),
            data.lightProjection);
11
12      glBindVertexArray(*data.modelVAO);
13      uniform_matrix4f(data.shaderDepthPass->get_uniform_location("model"),
            data.modelModel);
14      glDrawElements(GL_TRIANGLES, *data.modelIndexCount, GL_UNSIGNED_INT, 0);
15      glBindVertexArray(0);
16  }
```

Приведенная выше реализация алгоритма заполнения теневой карты, является универсальной для всех алгоритмов теней, за исключением того, что для каждого алгоритма используется своя структура данных.

Второй этап алгоритмов теневых карт так же является практически одинаковым для всех реализаций алгоритмов за исключением передачи некоторых уникальных данных в соответствующую шейдерную программу. В листинге 3.6 приведена реализация этапа рисования сцены с учетом теней для алгоритма стандартных теневых карт.

Листинг 3.6 – Алгоритм заполнения теневой карты

```
1   void RenderPass(ShadowMapMainRenderData& data)
2   {
3       glViewport(0, 0, *data.client_width, *data.client_height);
4
5       glBindFramebuffer(GL_FRAMEBUFFER, 0);
6       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
7
8       data.shaderRenderPass->use();
9       uniform_matrix4f(data.shaderRenderPass->get_uniform_location("view"), data.view);
10      uniform_matrix4f(data.shaderRenderPass->get_uniform_location("projection"),
            data.projection);
11      uniform_matrix4f(data.shaderRenderPass->get_uniform_location("lightView"),
            data.lightView);
12      uniform_matrix4f(data.shaderRenderPass->get_uniform_location("lightProjection"),
            data.lightProjection);
13      glUniform1f(data.shaderRenderPass->get_uniform_location("shadowBias"),
            *data.shadowBias);
14
15      glActiveTexture(GL_TEXTURE0);
16      glBindTexture(GL_TEXTURE_2D, data.depthBuffer->Texture);
17
18      glBindVertexArray(*data.modelVAO);
19      uniform_matrix4f(data.shaderRenderPass->get_uniform_location("model"),
            data.modelModel);
20      glDrawElements(GL_TRIANGLES, *data.modelIndexCount, GL_UNSIGNED_INT, 0);
21
22      glBindVertexArray(*data.planeVAO);
23      uniform_matrix4f(data.shaderRenderPass->get_uniform_location("model"),
            data.planeModel);
24      glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
25
26      glBindVertexArray(0);
27  }
```

Помимо таких параметров как view, projection, lightView, lightProjection и shadowBias, которые являются общими для всех реализаций, есть дополнительный – pcfRadius, используемый во всех алгоритмах кроме стандартного.

3.4.3 Реализация алгоритма стандартных теневых карт

Подготовительный этап

В подготовительном этапе рисования теней, представленном в листинге 3.5, используется шейдерная программа, контролирующая заполнение теневой карты значениями глубины (листинги 3.7 – 3.8).

Листинг 3.7 – Вершинный шейдер заполнения теневой карты

```
#version 460 core

layout(location = 0) in vec3 aPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Листинг 3.8 – Фрагментный шейдер заполнения теневой карты

```
#version 460 core

void main()
{

}
```

Особенностью реализации фрагментного шейдера (листинг 3.8) является пустая головная функция, так как никакой обработки пикселей не происходит, за исключением записи значений их глубины в теневую карту, предварительно подключенную на запись перед рисованием объектов, отбрасывающих тени.

Важно отметить, что рисовать все объекты подряд не требуется. Если достоверно известно, что экранируемая и собственная тени какого-либо объекта не будут видны с точки зрения пользователя, то такие объекты можно пропустить. При этом экранируемые тени от других объектов все еще будут корректно отображаться на поверхностях чужих объектов.

Именно поэтому в подготовительном этапе отображается только обозреваемый объект, в то время как рисование плоскости игнорируется.

Реализация шейдерной программы, приведенной в листингах 3.7 – 3.8, является общей для всех реализаций подготовительного этапа рисования теней.

Заключительный этап

Используя заполненную теневую карту в первом этапе рисования, заключительная шейдерная программа рисует сцену с учетом теней и освещения (листинги 3.9 – 3.10).

Листинг 3.9 – Вершинный шейдер преобразования координат

```
#version 460 core

layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform mat4 lightView;
uniform mat4 lightProjection;

out vec4 FragPos;
out vec3 Normal;
out vec4 FragPosLightSpace;

void main()
{
    FragPos = model * vec4(aPos, 1.0);
    Normal = mat3(transpose(inverse(model))) * aNormal;
    FragPosLightSpace = lightProjection * lightView * FragPos;
    gl_Position = projection * view * FragPos;
}
```

Листинг 3.10 – Фрагментный шейдер с учетом теней и освещения

```
#version 460 core

// Определение структур, описывающих освещение ...

out vec4 FragColor;

in vec4 FragPos;
in vec3 Normal;
in vec4 FragPosLightSpace;

uniform sampler2D shadowMap;
uniform float shadowBias;

// Объявление и реализация функций вычисляющих освещение ...

float ShadowCalculation(vec4 fragPosLightSpace)
{
    // Конвертация из [-1, 1] в [0, 1]
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;

    // Глубина текущего фрагмента из карты теней
```

```
23      float closestDepth = texture(shadowMap, projCoords.xy).r;
24      float currentDepth = projCoords.z;
25
26      // Проверка на тень
27      float shadow = step(closestDepth + shadowBias, currentDepth);
28      shadow *= step(currentDepth, 1.0);
29
30      return shadow;
31  }
32
33  void main()
34  {
35      vec3 color = vec3(0.5);
36      vec3 ambient = 0.15 * color;
37
38      vec3 diffuse = computeLightColor(FragPos.xyz, Normal) * color;
39
40      float shadow = ShadowCalculation(FragPosLightSpace);
41      vec3 lighting = (ambient + (1.0 - shadow) * diffuse);
42
43      FragColor = vec4(lighting, 1.0);
44  }
```

Вершинный шейдер, представленный в листинге 3.9 является общим для всех алгоритмов рисования теней.

Фрагментный шейдер (листинг 3.10) имеет вариацию в зависимости от вида проецирования: ортогонального или перспективного. В приведенном примере используется перспективное проецирование тени. Отличие ортогонального проецирования от перспективного заключается лишь в конвертации координат, то есть измена только функция *ShadowCalculation()*. Реализация такой вариации приведена в листинге 3.11.

Листинг 3.11 – Фрагментный шейдер с учетом ортогонального проецирония теней

```
1  // ...
2
3  float ShadowCalculation(vec4 fragPosLightSpace)
4  {
5      // Конвертация из [-1, 1] в [0, 1]
6      vec3 projCoords = fragPosLightSpace.xyz;
7      projCoords = projCoords * 0.5 + 0.5;
8
9      // Глубина текущего фрагмента из карты теней
10     float closestDepth = texture(shadowMap, projCoords.xy).r;
11     float currentDepth = projCoords.z;
12
13     // Проверка на тень
14     float shadow = step(closestDepth + shadowBias, currentDepth);
15     shadow *= step(currentDepth, 1.0);
16
17     return shadow;
18 }
19
20 //...
```

По итогу, перспективное деление игнорируется в силу того, что используется матрица ортогонального проецирования вместо перспективного.

3.4.4 Реализация алгоритма теневых карт с фильтрацией

Как было уточнено выше, шейдерная программа (листинги 3.7 – 3.8) в подготовительном этапе эквивалентна во всех реализациях, так же как и вершинный шейдер в заключительном этапе (листинг 3.9). Поэтому здесь и далее для всех реализаций алгоритмов теней будут приведены только фрагметные шейдеры заключительного этапа.

В листинге 3.12 представлена реализация фрагментного шейдера заключительного этапа рисования с учетом перспективной проекции теней с фильтрацией и освещения.

Листинг 3.12 – Фрагментный шейдер с учетом перспективного проецирония теней

```
#version 460 core

// Определение структур, описывающих освещение ...

out vec4 FragColor;

in vec4 FragPos;
in vec3 Normal;
in vec4 FragPosLightSpace;

uniform sampler2D shadowMap;
uniform float shadowBias;
uniform float pcfRadius;

// Объявление и реализация функций вычисляющих освещение ...

float ShadowCalculation(vec4 fragPosLightSpace)
{
    // Конвертация из [-1, 1] в [0, 1]
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;

    // Проверка на тень
    if (projCoords.z > 1.0)
        return 0.0;

    // Тень с использованием PCF
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for (int x = -1; x <= 1; ++x)
    {
        for (int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize
                * pcfRadius).r;
            shadow += projCoords.z - shadowBias > pcfDepth ? 1.0 : 0.0;
        }
    }

    return shadow / 9.0;
}

void main()
{
    vec3 color = vec3(0.5);
    vec3 ambient = 0.15 * color;
```

```
46
47      vec3 diffuse = computeLightColor(FragPos.xyz, Normal) * color;
48
49      float shadow = ShadowCalculation(FragPosLightSpace);
50      vec3 lighting = (ambient + (1.0 - shadow) * diffuse);
51
52      FragColor = vec4(lighting, 1.0);
53  }
```

Вариация реализации фрагментного шейдера рисования теней с фильтрацией с ортогональной проекцией теней представлена в листинге 3.13, конфигурация которой заключается только в функции *ShadowCalculation()*.

Листинг 3.13 – Фрагментный шейдер с учетом ортогонального проецирония теней

```
1  // ...
2
3  float ShadowCalculation(vec4 fragPosLightSpace)
4  {
5      // Конвертация из [-1, 1] в [0, 1]
6      vec3 projCoords = fragPosLightSpace.xyz;
7      projCoords = projCoords * 0.5 + 0.5;
8
9      // Тень с использованием PCF
10     float shadow = 0.0;
11     vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
12     for (int x = -1; x <= 1; ++x)
13     {
14         for (int y = -1; y <= 1; ++y)
15         {
16             float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize
                   * pcfRadius).r;
17             shadow += projCoords.z - shadowBias > pcfDepth ? 1.0 : 0.0;
18         }
19     }
20
21     return shadow / 9.0;
22 }
23
24 // ...
```

3.4.5 Реализация алгоритма теневых карт с фильтрацией шумом

В листинге 3.14 представлена реализация фрагментного шейдера заключительного этапа рисования с учетом перспективной проекции теней с фильтрацией шумом и освещения.

Листинг 3.14 – Фрагментный шейдер с учетом перспективного проецирония теней

```
1  #version 460 core
2
3  // Объявление структур, описывающих освещение ...
4
5  out vec4 FragColor;
6
7  in vec4 FragPos;
8  in vec3 Normal;
9  in vec4 FragPosLightSpace;
10
```

```
11 uniform sampler2D shadowMap;
12 uniform float shadowBias;
13 uniform float pcfRadius;
14
15 #define PI 3.14159265358
16
17 // Объявление и реализация функций вычисляющих освещение ...
18
19 // Случайное число
20 float rand(vec2 uv)
21 {
22     return fract(sin(dot(uv, vec2(12.9898, 4.1414))) * 43758.5453);
23 }
24
25 // Случайное гауссово распределение
26 float randomGaussian(float mean, float stddev, vec2 seed) {
27     float u1 = rand(seed);
28     float u2 = rand(seed + vec2(1.0));
29     float r = sqrt(-2.0 * log(u1)) * cos(2.0 * PI * u2);
30     return mean + stddev * r;
31 }
32
33 // Шум
34 vec2 generateNoise(float mean, float stddev, vec2 seed)
35 {
36     return vec2(randomGaussian(mean, stddev, seed * PI), randomGaussian(mean, stddev,
        seed));
37 }
38
39 float ShadowCalculation(vec4 fragPosLightSpace)
40 {
41     // Конвертация из [-1, 1] в [0, 1]
42     vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
43     projCoords = projCoords * 0.5 + 0.5;
44
45     // Проверка на тень
46     if (projCoords.z > 1.0)
47         return 0.0;
48
49     // Тень с использованием Шума
50     float shadow = 0.0;
51     vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
52     for (int x = -1; x <= 1; ++x)
53     {
54         for (int y = -1; y <= 1; ++y)
55         {
56             vec2 noiseOffset = generateNoise(0.0, 1.0, projCoords.xy + vec2(x, y));
57             float pcfDepth = texture(shadowMap, projCoords.xy + noiseOffset *
                texelSize * pcfRadius).r;
58             shadow += projCoords.z - shadowBias > pcfDepth ? 1.0 : 0.0;
59         }
60     }
61
62     return shadow / 9.0;
63 }
64
65 void main()
66 {
67     vec3 color = vec3(0.5);
68     vec3 ambient = 0.15 * color;
69
70     vec3 diffuse = computeLightColor(FragPos.xyz, Normal) * color;
71
72     float shadow = ShadowCalculation(FragPosLightSpace);
73     vec3 lighting = (ambient + (1.0 - shadow) * diffuse);
74
```

```
75      FragColor = vec4(lighting, 1.0);
76  }
```

Вариация реализации фрагментного шейдера рисования теней с фильтрацией шумом с ортогональной проекцией теней представлена в листинге 3.15, конфигурация которой заключается только в функции *ShadowCalculation()*.

Листинг 3.15 – Фрагментный шейдер с учетом ортогонального проецирония теней

```
1  // ...
2
3  float ShadowCalculation(vec4 fragPosLightSpace)
4  {
5      // Конвертация из [-1, 1] в [0, 1]
6      vec3 projCoords = fragPosLightSpace.xyz;
7      projCoords = projCoords * 0.5 + 0.5;
8
9      // Проверка на тень
10     if (projCoords.z > 1.0)
11         return 0.0;
12
13     // Тень с использованием Шума
14     float shadow = 0.0;
15     vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
16     for (int x = -1; x <= 1; ++x)
17     {
18         for (int y = -1; y <= 1; ++y)
19         {
20             vec2 noiseOffset = generateNoise(0.0, 1.0, projCoords.xy + vec2(x, y));
21             float pcfDepth = texture(shadowMap, projCoords.xy + noiseOffset *
                   texelSize * pcfRadius).r;
22             shadow += projCoords.z - shadowBias > pcfDepth ? 1.0 : 0.0;
23         }
24     }
25
26     return shadow / 9.0;
27 }
28
29 // ...
```

3.4.6 Реализация алгоритма мягких теневых карт с фильтрацией

В листинге 3.16 представлена реализация фрагментного шейдера заключительного этапа рисования с учетом перспективной проекции мягких теней с фильтрацией и освещения.

Листинг 3.16 – Фрагментный шейдер с учетом перспективного проецирония теней

```
1  #version 460 core
2
3  // Объявление структур, описывающих освещение ...
4
5  out vec4 FragColor;
6
7  in vec4 FragPos;
8  in vec3 Normal;
9  in vec4 FragPosLightSpace;
10
11 in vec2 lightProjectionParams;
```

```
12
13 uniform sampler2D shadowMap;
14 uniform float shadowBias;
15 uniform float pcfRadius;
16 uniform float lightRadius = 1.0; // Радиус источника света
17
18 #define PI 3.14159265358
19
20 // Объявление и реализация функций вычисляющих освещение ...
21
22 vec2 sampleDisk(int sampleIndex, int numSamples)
23 {
24     float angle = float(sampleIndex) / float(numSamples) * 2.0 * PI;
25     return vec2(cos(angle), sin(angle));
26 }
27
28 float SearchBlocker(vec3 projCoords, float searchRadius)
29 {
30     int numSamples = 16;  // количество выборок вокруг текущего фрагмента
31     float blockerDepthSum = 0.0;
32     int numBlockers = 0;
33
34     // Вычисление размера текстеля
35     vec2 texelSize = 1.0 / textureSize(shadowMap, 0); // 0 - уровень MIP карты
36
37     // Выборки вокруг текущей точки (в пределах searchRadius)
38     for (int i = 0; i < numSamples; ++i)
39     {
40         // Генерация смещенной позиции выборки (например, случайное смещение или с испо
               льзованием диска)
41         vec2 offset = sampleDisk(i, numSamples) * texelSize * searchRadius;
42
43         // Чтение глубины из карты теней для соседней точки
44         float depthInShadowMap = texture(shadowMap, projCoords.xy + offset).r;
45
46         // Если точка блокирует свет, добавить её глубину
47         if (depthInShadowMap < projCoords.z + shadowBias)
48         {
49             blockerDepthSum += depthInShadowMap;
50             ++numBlockers;
51         }
52     }
53
54     // Если найдены блокирующие объекты, усреднить их глубину
55     if (numBlockers > 0)
56     {
57         return blockerDepthSum / float(numBlockers);
58     }
59
60     // Если блокирующие объекты не найдены, вернуть -1
61     return -1.0;
62 }
63
64 float EstimatePenumbraSize(float fragDepth, float blockerDepth)
65 {
66     // Если нет блокирующего объекта, полутени нет
67     if (blockerDepth < 0.0)
68     {
69         return 0.0;
70     }
71
72     float viewLightFragDepth = lightProjectionParams.y / (fragDepth * 2.0 - 1.0 -
           lightProjectionParams.x);
73     float viewLightBlockerDepth = lightProjectionParams.y / (blockerDepth * 2.0 - 1.0
           - lightProjectionParams.x);
74
```

```
75      // Оценка размера полутени
76      float penumbraSize = (viewLightFragDepth - viewLightBlockerDepth) /
            viewLightBlockerDepth * lightRadius;
77
78      return penumbraSize;
79  }
80
81  float PerformPCF(vec3 projCoords, float penumbraSize)
82  {
83      // Определение ядра для PCF
84      int samleSize = 16; // Предпочтительно нечетное
85      int halfSize = samleSize / 2;
86      float samleCount = pow(halfSize * 2 + 1, 2.0);
87
88      float shadow = 0.0;
89
90      // Применить значение penumbraSize для масштабирования области выборки
91      float filterRadius = penumbraSize;
92
93      // Вычисление размера текстеля
94      vec2 texelSize = 1.0 / textureSize(shadowMap, 0); // 0 - уровень MIP карты
95
96      // Цикл по выборкам вокруг текущей координаты в пространстве света
97      for (int x = -halfSize; x <= halfSize; ++x)
98      {
99          for (int y = -halfSize; y <= halfSize; ++y)
100         {
101             float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize
                    * filterRadius).r;
102             shadow += projCoords.z - shadowBias > pcfDepth ? 1.0 : 0.0;
103         }
104     }
105
106     return shadow / samleCount;
107 }
108
109 float ShadowCalculation(vec4 fragPosLightSpace)
110 {
111     // Преобразование координат в проекционные
112     vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
113     projCoords = projCoords * 0.5 + 0.5;
114
115     if (projCoords.z > 1.0)
116         return 0.0;
117
118     // Шаг 1: Оценка блокирующих объектов
119     float blockerDepth = SearchBlocker(projCoords, pcfRadius);
120
121     if (blockerDepth < 0.0)
122         return 0.0;
123
124     // Шаг 2: Оценка размера полутени
125     float penumbraSize = EstimatePenumbraSize(projCoords.z, blockerDepth);
126
127     // Шаг 3: Применение PCF с изменяемым радиусом
128     return PerformPCF(projCoords, penumbraSize);
129 }
130
131 void main()
132 {
133     vec3 color = vec3(0.5);
134     vec3 ambient = 0.15 * color;
135
136     vec3 diffuse = computeLightColor(FragPos.xyz, Normal) * color;
137
138     float shadow = ShadowCalculation(FragPosLightSpace);
```

```
139     vec3 lighting = (ambient + (1.0 - shadow) * diffuse);
140
141     FragColor = vec4(lighting, 1.0);
142 }
```

Вариация реализации фрагментного шейдера рисования мягких теней с фильтрацией с ортогональной проекцией теней представлена в листинге 3.17, конфигурация которой заключается только в функции *ShadowCalculation()*.

Листинг 3.17 – Фрагментный шейдер с учетом ортогонального проецирония теней

```
1 // ...
2
3
4
5 // ...
```

3.4.7 Реализация алгоритма мягких теневых карт с фильтрацией шумом

В листинге **??** представлена реализация фрагментного шейдера заключительного этапа рисования с учетом перспективной проекции мягких теней с фильтрацией шумом и освещения.

Вариация реализации фрагментного шейдера рисования мягких теней с фильтрацией шумом с ортогональной проекцией теней представлена в листинге **??**, конфигурация которой заключается только в функции *ShadowCalculation()*.

3.4.8 Реализация алгоритма освещения

В каждых из выше

Вывод

4 Исследовательский раздел

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Уильямс, Л. Отбрасывание изогнутых теней на изогнутые поверхности / Лэнс Уильямс // Computer Graphics Lab, New York Institute of Technology. — Old Westbury, New York, 11568.

[2] Щупак, Ю. А. Win32 API. Разработка приложений для Windows. – СПб.: Питер, 2008. – 592 с.: ил. – ISBN 978-5-388-00301-0

[3] Литвиненко, Н. А. Технология программирования на C++. Win32 API-приложения. – СПб.: БХВ-Петербург, 2010. – 288 с.: ил. – (Учебное пособие) ISBN 978-5-9775-0600-7

[4] Вольф, Д. OpenGL 4. Язык шейдеров. Книга рецептов / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2015. – 368 с.: ил. – ISBN 978-5-97060-255-3

[5] Гинсбург Д., Пурномо Б. OpenGL ES 3.0. Руководство разработчика / пер. с англ. А. Борескова. – М.: ДМК Пресс, 2015. – 448 с.: ил. – ISBN 978-5-97060-256-0

[6] Боресков А. Расширения OpenGL / Боресков А. – Санкт-Петербург : БХВ-Петербург, 2005. – 688 с. – ISBN 5-94157-614-5.

[7] Горнаков С. Инструментальные средства программирования и отладки шейдеров в DirectX и OpenGL / Горнаков С. - Санкт-Петербург : БХВ-Петербург, 2005. - 256 с. - ISBN 5-94157-611-0.

[8] Селлерс, Г. Vulkan. Руководство разработчика : руководство / Г. Селлерс ; перевод с английского А. В. Борескова. — Москва : ДМК Пресс, 2017. — 394 с. — ISBN 978-5-97060-486-1.

[9] Потапов, А. П. Линейная алгебра и аналитическая геометрия : учебник и практикум для вузов / А. П. Потапов. — Москва : Издательство Юрайт, 2024. — 309 с. — (Высшее образование). — ISBN 978-5-534-01232-3. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/536935 (дата обращения: 01.11.2024).

[10] all – GLSL 4 – docs.gl [Электронный ресурс]. – URL: https://docs.gl/sl4/all (Дата обращения 18.09.2024)

[11] Функция QueryPerformanceFrequency - Win32 apps | Microsoft Learn [Электронный ресурс]. – URL: https://learn.microsoft.com/ru-ru/windows/

win32/api/profileapi/nf-profileapi-queryperformancefrequency (Дата обращения 18.09.2024)