

**Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»**

**Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика**

**КУРСОВАЯ РАБОТА
Программный проект на тему
Продвинутый 3D renderer**

**Выполнил студент группы БПМИ196, 3 курса,
Смородинов Александр Андреевич**

**Руководитель КР:
Доцент, кандидат физико-математических наук, Трушин
Дмитрий Витальевич**

Москва 2022

Оглавление

| | |
|---|-----------|
| 1 Аннотация | 3 |
| 1.1 Ключевые слова | 4 |
| 2 Введение | 4 |
| 2.1 Постановка задачи | 4 |
| 2.1.1 Цели | 4 |
| 2.1.2 Задачи | 5 |
| 2.2 Актуальность и значимость | 5 |
| 2.3 Полученные результаты | 7 |
| 2.3.1 Репозиторий | 7 |
| 2.3.2 Реализованные продвинутые алгоритмы | 7 |
| 2.3.3 Функциональные требования к результату | 7 |
| 2.3.4 Нефункциональные требования к результату | 7 |
| 2.4 Новизна и полезность полученных результатов | 8 |
| 2.5 Структура работы | 8 |
| 3 Обзор аналогов | 9 |
| 3.1 Mesa | 9 |
| 3.2 Другие аналоги | 9 |
| 4 Краткое описание реализованных алгоритмов | 11 |
| 4.1 HDR | 12 |
| 4.2 Bloom | 13 |
| 4.3 Отрисовка полупрозрачных объектов, blending | 14 |
| 4.4 Shadow mapping | 15 |
| 4.5 Карта нормалей (normal mapping) | 18 |
| 4.6 Parallax mapping | 19 |
| 4.7 Отложенное освещение и затенение (deferred shading) | 20 |
| 4.7.1 Мотивация | 20 |
| 4.7.2 Краткое описание | 21 |
| 4.8 Screen space ambient occlusion (SSAO) | 21 |
| 4.8.1 Ambient occlusion | 21 |
| 4.8.2 SSAO | 22 |
| 5 Детали реализации приложения | 23 |
| 5.1 Изменения в архитектуре | 23 |
| 5.2 Основные классы в реализации | 24 |
| 5.2.1 Application | 24 |

| | |
|--|-----------|
| 5.2.2 UserInterface | 24 |
| 5.2.3 CameraControl | 24 |
| 5.2.4 Scene | 24 |
| 5.2.5 Assets | 25 |
| 5.2.6 SFMLRenderer | 25 |
| 5.2.7 Renderer | 25 |
| 5.2.8 Pipeline | 25 |
| 5.2.9 Mesh<VertexShader, FragmentShader> | 26 |
| 5.2.10 ColorAndDepthBuffer<...> | 26 |
| 5.2.11... | 26 |
| 5.3 Структурная диаграмма классов UML | 26 |
| 5.3.1 Легенда и краткое описание элементов диаграммы | 27 |
| 5.4 Тесты на производительность | 27 |
| 6 Заключение | 28 |
| 6.1 Результат | 28 |
| 6.1.1 Количественные характеристики (просили привести) | 29 |
| 6.2 Направления дальнейших разработок | 30 |

1 Аннотация

На сегодняшний день технологии отрисовки трёхмерных сцен используются во многих сферах нашей жизни: в 3D моделировании и анимации, в компьютерных играх и 3D/VR симуляциях. Первые статьи по данной теме были опубликованы ещё в 60х-70х годах 20 века, и с тех пор было проведено огромное количество исследований и работ в сфере 3D рендеринга.

Цель данной работы - изучить и реализовать ряд важнейших алгоритмов 3D рендеринга.

На втором курсе в рамках программного проекта было реализовано с нуля приложение, позволяющее пользователю взаимодействовать с трёхмерной сценой, задавать различные режимы отрисовки для объектов сцены, менять параметры освещения, источников света, камеры и экрана в реальном времени.

В рамках данной курсовой работы в приложение была добавлена поддержка различных продвинутых алгоритмов отрисовки: normal mapping, parallax mapping, HDR, bloom, shadow mapping,

отрисовка полупрозрачных объектов, deferred shading, SSAO, а также была улучшена производительность.

Abstract

Nowadays 3D rendering algorithms are used in many parts of our lives: 3D modeling and animation, computer games and 3D/VR simulations. First articles on this topic were published in the sixties and seventies of the 20th century, and since then a large number of studies have been carried out on this topic.

The goal of this paper is to study and implement some of the main 3D rendering algorithms.

In the second year as part of a program project, an application was developed from scratch that allows user to interact with 3D scene, set different modes of rendering for scene objects, change parameters of lighting, light sources, camera and screen in real time.

As part of this paper, support for advanced algorithms such as normal mapping, parallax mapping, HDR, bloom, shadow mapping, rendering of transparent objects, deferred shading and SSAO, was added to the application, and performance was also improved.

1.1 Ключевые слова

Программный 3D рендеринг, shadow mapping, normal mapping, parallax mapping, HDR, bloom, deferred shading, SSAO, C++, SFML, glm, imgui, stb_image.

2 Введение

2.1 Постановка задачи

2.1.1 Цели

Основная цель проекта - это изучение и реализация алгоритмов, использующихся в компьютерной графике, на которых основаны большинство программ 3D моделирования, 3D игр, 3D/VR симуляторов и других приложений, имеющих какое-либо отношение к трёхмерной графике.

2.1.2 Задачи

Основные задачи:

- Изучить и реализовать продвинутые алгоритмы 3D рендеринга (в секции с описанием полученных результатов будет конкретный список).
- Интегрировать реализацию продвинутых алгоритмов в существующую базовую версию 3D рендерера (в интерактивное приложение).
- Изложить теоретические основы реализованных алгоритмов, описать детали их реализации, написать сопроводительную документацию к коду и протестировать его.

2.2 Актуальность и значимость

3D рендеринг - это активно развивающаяся и относительно молодая область компьютерной графики, и результаты исследований в данной сфере часто находят применение в 3D анимации, моделировании, 3D/VR симуляторах, компьютерных играх и других приложениях.

Важной особенностью описываемой в данной работе реализации является тот факт, что в проекте не используются такие известные API для работы с компьютерной графикой как OpenGL, DirectX, Vulcan и др. Это означает, что весь рендеринг происходит исключительно на CPU (так называемый software rendering).

Данный подход имеет ряд своих преимуществ и недостатков, по сравнению с GPU rendering-ом:

Преимущества:

- Работает на всех устройствах, независимо от наличия видеокарты (в том числе микроконтроллерах и других встроенных системах).
- Также не нужно требовать от видеокарты пользователя поддержки конкретной версии API (например OpenGL 3.3).
- На всех устройствах одно и то же приложение работает одинаково, нет уязвимости к багам реализации API.

- У разработчика software renderer-а есть полный контроль над реализацией. Добавлять новые функции в рендерер, изменять существующие алгоритмы, исправлять баги в реализации в разы проще и быстрее, чем при использовании сторонних решений.
- С образовательной точки зрения, самостоятельная реализация 3D рендерера с нуля даёт намного более глубокое понимание работы алгоритмов 3D рендеринга, чем просто использование некоторой готовой библиотеки.

Недостатки:

- Главный недостаток - скорость. Видеокарты намного лучше справляются с множеством хорошо распараллелиемых вычислений, чем процессоры. Они специально для этого и проектировались. Разница в скорости очень сильно зависит от конкретной видеокарты и конкретного приложения, но может составлять несколько порядков.
- С точки зрения разработчика - нужно всё писать с нуля самому, а не использовать API (хотя также есть и уже написанные библиотеки).

Реализованный в данной работе 3D рендерер, так как является программным, практически не имеет шансов соперничать с существующими аппаратными рендерами в производительности.

Тем не менее в случаях, когда производительность не так важна, но необходима кроссплатформенность и отсутствие большого количества внешних зависимостей, реализованный 3D рендерер может быть полезен в качестве библиотеки отрисовки, так как он поддерживает не только базовые, но и продвинутые алгоритмы рендеринга.

Также, данный проект будет полезен в качестве примера реализации программного 3D рендерера на C++, поддерживающего продвинутые алгоритмы, для тех, кто интересуется компьютерной графикой и C++.

Работа также значима лично для автора статьи, как возможность изучить алгоритмы 3D рендеринга и реализовать их на практике.

2.3 Полученные результаты

2.3.1 Репозиторий

Репозиторий проекта: [4]

2.3.2 Реализованные продвинутые алгоритмы

1. Отображение нормалей (normal mapping).
2. "Параллакс" отображение (parallax mapping).
3. HDR, tone mapping, gamma correction.
4. Bloom.
5. Рендеринг прозрачных объектов.
6. Shadow mapping.
7. Отложенное освещение и затенение (deferred shading).
8. Screen space ambient occlusion (SSAO).

2.3.3 Функциональные требования к результату

Результат проекта - интерактивное приложение, доступное на ОС windows / linux, в котором пользователь может:

- Просматривать различные 3D сцены, управляя камерой с клавиатуры.
- Настраивать параметры сцены.
- Открывать и добавлять в сцену 3D объекты, сохранённые в различных форматах (на данный момент реализована загрузка простейших 3D моделей в формате waveform obj [12]).

2.3.4 Нефункциональные требования к результату

- Язык программирования - C++ (стандарт C++17).
- Используемые библиотеки (статическая линковка):

- glm 0.9.9 [5].
 - SFML 2.5.1 [13].
 - ImGui v1.83 [6].
 - stb_image 2.26 [21].
- Система контроля версий - git.
 - Линтер/форматтер - clang format, настройки основаны на google codestyle.
 - Для сборки проекта используется IDE Microsoft Visual Studio 2019.

2.4 Новизна и полезность полученных результатов

С теоретической точки зрения все реализованные алгоритмы являются достаточно известными и применяются во многих 3D рендерерах. Например, большая часть из них описана на сайте [8], а также в книгах [27] и [28]. Поэтому теоретической значимости и новизны в данном проекте практически нет.

Но, с практической точки зрения, как далее будет показано, аналогов программного 3D рендерера на C++, поддерживающих все перечисленные выше продвинутые алгоритмы отрисовки, в открытом доступе довольно мало, если они вообще есть (по крайней мере автор прямого аналога не нашёл), и в этом плане работа обладает некоторой новизной. Обоснование практической полезности программных 3D рендереров было приведено выше.

Так как проект программный, такое положение вещей кажется достаточно естественным (малая теоретическая полезность, но существующая практическая полезность).

2.5 Структура работы

В секции 3 приведён обзор аналогов. В секции 4 приведено краткое описание реализованных алгоритмов. В секции 5 описываются особенности реализации приложения. Секция 6 - заключение отчёта по курсовой работе. После секции 6 идёт список источников.

3 Обзор аналогов

Существует множество реализаций программных 3D рендереров. Во многих институтах, где читают лекции по 3D графике реализация 3D рендерера может являться домашним заданием которое выполняют студенты. Также многие разработчики реализуют 3D рендерер для того, чтобы изучить различные алгоритмы 3D рендеринга, в качестве так называемого pet-project-a.

За исключением данных примеров, программные 3D рендереры на данный момент применяются достаточно редко, в основном из-за их низкой производительности по сравнению с аппаратными 3D рендерерами.

3.1 Mesa

Это не совсем аналог данного проекта, но Mesa - [10] содержит программную (software rendering) реализацию OpenGL 3.1. Так что можно с использованием Mesa пользоваться OpenGL (и рядом других API) даже на устройствах, не имеющих GPU (также Mesa реализует набор видеодрайверов для различных GPU).

На устройствах с дистрибутивами GNU/Linux, *BSD и другими ОС Mesa является основой графического стека, так как Mesa - это свободная реализация основных графических API.

Из-за такого широкого распространения было бы немного неправильно вообще не упомянуть данное решение, но как уже было написано выше - это не совсем аналог данного проекта, поэтому сравнивать их между собой довольно сложно.

3.2 Другие аналоги

Так как в интернете в открытом доступе существует большое количество реализаций программного 3D рендеринга, то все их проанализировать не получится, поэтому будет приведён только анализ наиболее популярных по количеству звёзд на гитхабе решений. (метрика популярности довольно произвольная, но нужно же было как-то отсортировать решения). Поиск проектов на гитхабе был по топику software-rendering (полный список - [26]).

1. ssloy/tinyrenderer [20] Отличный проект, весь код занимает примерно 500 строчек. Но реализовано не так много ал-

горитмов 3D рендеринга (например нет клиппинга). Вообще проект в основном носит образовательный характер, и в этом плане там очень хорошие уроки на вики странице.

2. [zauonlok/renderer](#) [24] Тоже интересный проект, написан на C89, реализовано довольно много алгоритмов, но более продвинутые техники также не реализованы.
3. [kosua20/heredragons](#) [7] Не совсем корректно считать этот проект аналогом, так как это не software рендерер, а наоборот, реализация одной сцены с помощью разных графических API. Интересно то, что также есть реализации сцены на платформах, где никаких API для 3D графики нет (например для PICO-8 и Nintendo Game Boy Advance). На таких платформах используются различные способы обойти аппаратные ограничения.
4. [skywind3000/mini3d](#) [16] Небольшой (700 строк) 3D рендерер на с. Не реализованы почти никакие дополнительные / продвинутые алгоритмы.
5. [ssloy/tinyraycaster](#) [19] Как видно из названия, это рейкастер, а не полноценный 3D рендерер, поэтому тоже не очень корректный пример.
6. [skywind3000/RenderHelp](#) [17] Ещё один довольно простой 3D рендерер на C++.
7. [Angelo1211/SoftwareRenderer](#) [1] Один из самых интересных проектов из всего списка. Реализовано довольно много продвинутых алгоритмов. Тем не менее, есть несколько проблем с сглаживанием и "Муаровым эффектом" в некоторых сценах, а также есть нереализованные алгоритмы (из списка реализованных в данном проекте).
8. [martinResearch/DEODR](#) [9] Дифференцируемый 3D рендерер на с. Какая-то классная штука для ML. Не уверен, часто ли данную библиотеку используют в других областях.

Бонусные аналоги (не из списка выше, по крайней мере не из топа по звёздам)

1. bytecode77/fastpix3d [2] Данный 3D рендерер показывает довольно хорошие показатели по производительности. Хотя в примерах используются не очень детализированные модели и текстуры в не очень высоком разрешении, производительность не может не впечатлять, учитывая, что это software рендерер. Каких-то очень продвинутых алгоритмов не реализовано, хотя есть поддержка освещения и теней.
2. Dawoodoz/DFPSR [3] В данном проекте применены очень интересные идеи. Библиотека предназначена для изометрических игр/сцен, и если использовать тот факт, что объекты всегда будут видны только с одного ракурса, то можно "запечь" (pre-render, bake) 3D модель в три текстуры - diffuse, normal, height, а дальше работать с моделью как с одним прямоугольником (причём ориентированным в пространстве камеры вдоль координатных осей). Это очень сильно снижает необходимое количество вычислений на процессоре и позволяет отрисовывать достаточно сложные изометрические сцены в реальном времени.

Как можно видеть, довольно мало проектов из перечисленных выше реализуют какие-то продвинутые алгоритмы 3D рендеринга.

Вывод: реализованный проект имеет практическую значимость и имеет малое количество аналогов по функциональным возможностям.

Также можно дополнительно отметить, что в реализованном приложении есть UI для редактирования сцены и параметров отдельных объектов, тогда как большинство перечисленных аналогов поддерживают только отображение статичных сцен (без возможности их редактирования прямо в приложении).

4 Краткое описание реализованных алгоритмов

Далее будет представлено очень краткое описание реализованных алгоритмов. Более подробную информацию можно найти в книгах [27], [28] и на сайте [8] (здесь описание алгоритмов объясняется наверное наиболее доступно), если читатель захочет бо-

лее глубоко ознакомиться с тематикой проекта. Предполагается, что с базовыми понятиями 3D рендеринга читатель уже знаком. Если это не так, то можно например прочитать отчёт по курсовой работе прошлого года [25].

4.1 HDR

Большинство мониторов способны отображать только цвета, каждый (r, g, b) компонент которых лежит в интервале от 0 до 1 (или от 0 до 255, если компонент задаётся одним байтом, а не числом с плавающей точкой). Поэтому в большинстве графических API, в частности OpenGL, по умолчанию отрисовка производится в буфер, в котором все компоненты также ограничены интервалом $[0, 1]$. В базовой версии программного 3D рендерера буфер также поддерживает только цвета с ограниченными компонентами.

При этом если в 3D сцене при вычислении цвета пикселя получается так, что его компоненты превосходят 1, то по умолчанию величины компонент просто уменьшают до 1 (например $(1.2, 0.3, 1.7) \Rightarrow (1, 0.3, 1)$). Такое преобразование цвета с произвольными компонентами в цвет с ограниченными компонентами называется tone mapping. Изначально цвет у нас задаётся без ограничений, то есть в HDR (High Dynamic Range), и мы преобразуем его в LDR (Low Dynamic Range) цвет (компоненты от 0 до 1).

Наивное clamp преобразование (когда мы просто ограничиваем величины больше 1 единицей) приводит к тому, что мы теряем информацию о цвете ярких фрагментов сцены. Большие участки сцены могут например просто все быть отрисованы одним белым цветом $((1, 1, 1))$, что выглядит не очень красиво.

Есть множество распространенных tone mapping функций, которые обеспечивают различный баланс детализации тёмных и ярких участков сцены.

Например преобразование exposure tone mapping: $1 - \exp(-x \cdot \varepsilon)$, где ε - настраиваемый параметр (выдержка), в зависимости от которого сцена будет отрисована либо светлее, либо темнее.

Часто бывает полезно не сразу делать tone mapping внутри шейдера, а отрисовать сцену в HDR буфер, использовать его (например применить ещё bloom), и только потом преобразовать HDR буфер в LDR, с применением tone mapping.

4.2 Bloom

Данная техника в большинстве случаев применяется вместе с HDR. Основная идея состоит в том, чтобы добавить ярким источникам света эффект "свечения": вокруг данных источников свет "проникает" в другие пиксели (это выражается в том, что пиксели вокруг ярких источников света в небольшом радиусе сами становятся ярче и частично окрашиваются в цвет источника). Если использовать данный эффект в разумных количествах, то это позволяет создавать интересные сцены и значительно улучшить качество освещения сцены.

Реализуется bloom следующим образом:

1. Отрисовываем сцену в HDR буфер.
2. Выделяем из HDR буфера яркие пиксели (например с яркостью > 1) и копируем их во вспомогательный буфер (изначально вспомогательный буфер полностью чёрный).
3. Применяем какой-нибудь алгоритм размытия (например Гауссовское размытие, его можно эффективно реализовать как горизонтальное + вертикальное размытие). Для вычисления Гауссовского размытия нужно для каждого пикселя вычислить среднее взвешенное цветов соседних (например в квадрате $N \times N$) пикселей, при этом в качестве весов используется функция Гаусса. Благодаря свойству, что двумерное гауссовское размытие можно представить как последовательное одномерное горизонтальное и вертикальное размытие (размытие отдельных строк и столбцов), можно значительно снизить количество вычислений (из квадратичного по N до линейного).
4. Прибавляем к оригинальному буферу вспомогательный.
5. Делаем tone mapping.

Рис. 1 - пример сцены, отрисованной без HDR, с HDR и с bloomом. (возможно была выбрана не лучшая сцена для демонстрации данного эффекта)

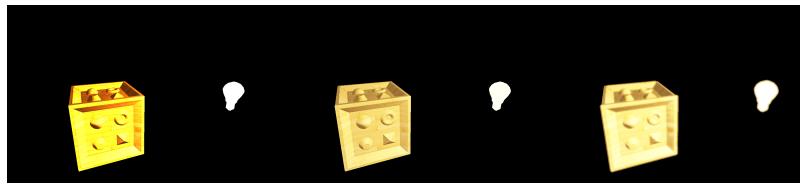


Рис. 1: Сравнение обычной сцены, сцены с HDR и сцены с bloom.

4.3 Отрисовка полупрозрачных объектов, blending

Сквозь полупрозрачные объекты можно видеть объекты за ними, но при этом цвет этих объектов также изменяется в зависимости от цвета полупрозрачного объекта, который расположен перед ними. Этот эффект в 3D рендеринге называется *blending* (смешение), так как цвета различных объектов "смешиваются".

Если поддержать в 3D рендерере *blending*, то это позволит отрисовывать сцены с полупрозрачными объектами. Чтобы реализовать *blending*, нужно просто перед тем, как записывать пиксель в буфер вызывать специальную *blend* функцию, которая принимает цвет текущего пикселя в буфере (*destination*), а также цвет пикселя, который мы хотим записать (*source*). И в буфер мы уже будем записывать не *source*, а $\text{blend}(\text{destination}, \text{source})$ - результат *blend* функции.

Также необходимо каким-то образом упорядочить пиксели при их отрисовке по уменьшению глубины, так как функция *blend* в общем случае не является коммутативной и от порядка попадания пикселей в буфер будет зависеть итоговый результат. При этом "правильный" порядок применения *blend* функции - это именно от дальних пикселей к ближним. Например, самые близкие к камере пиксели полупрозрачных объектов должны быть аргументами *blend* только один раз, и только в качестве *source*, иначе результат будет выглядеть некорректно.

В реализованном 3D рендерере сортировка пикселей производится довольно наивно - сортируются только отдельные треугольные грани объектов, при этом расстояние считается от камеры до центра треугольников. Для многих сцен такой подход работает достаточно неплохо (без артефактов), но существуют случаи, когда он будет производить некорректные результаты (обычно когда два треугольника находятся близко друг к другу).

Одной из возможных альтернатив было бы хранить в буфере цветов для каждого пикселя целый список цветов (то есть не использовать *depth test* для полупрозрачных фрагментов, и сохранять все такие фрагменты). Минусом такого решения было бы

достаточно высокое потребление памяти (в теории неограниченное, в зависимости от сложности сцены и потенциально растущее достаточно быстро), и также скорость отрисовки, из-за сложности такого подхода. Но в теории, такой вариант реализации тоже возможен, хотя на практике он встречается довольно редко (впрочем, то же самое можно сказать и о программных рендерерах), в частности из-за того, что большинство видеокарт не поддерживают такие неограниченные по глубине и по сути трёхмерные буферы.

Проблема отрисовки полупрозрачных объектов является достаточно нетривиальной, какого-то однозначно общепринятого решения кажется нет, поэтому применённый подход, учитывая свою простоту, в целом является неплохим.

Рис. 2 - отрисованная сцена с полупрозрачными объектами.

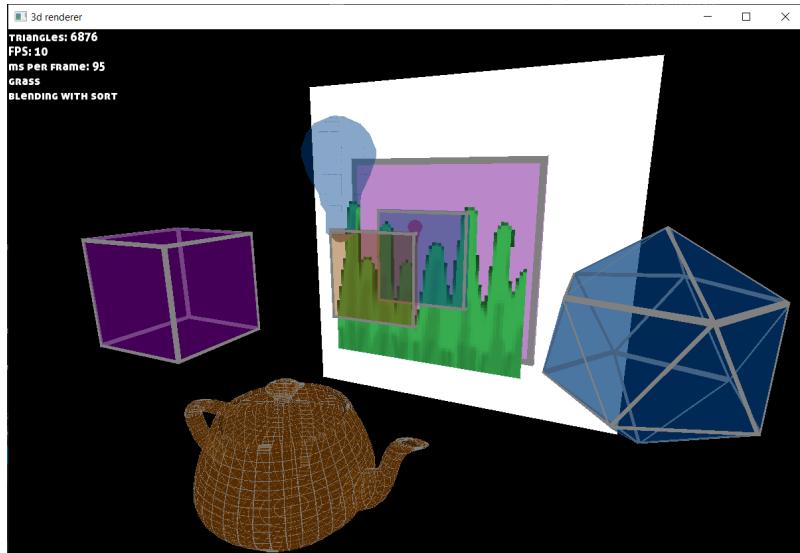


Рис. 2: Сцена с полупрозрачными объектами.

4.4 Shadow mapping

Данный алгоритм применяется для отрисовки теней от направленных источников света (то есть располагающихся настолько далеко, что испускаемые ими лучи можно считать параллельными, например в некоторых случаях можно рассматривать солнце как такой источник).

Основная идея состоит в том, что сначала мы отрисовываем всю сцену с камерой расположенной в некоторой позиции, так чтобы сцена помещалась на экран (этую позицию можно считать источником, хотя и предполагается, что источник расположен

бесконечно далеко), направленной так же, как и источник света. В тени будут находиться все фрагменты, которые не попали в итоговый буфер (то есть провалившие проверку на глубину - depth check), при отрисовке сцены со стороны источника света. Действительно, если фрагмент не попал в буфер, то это означает, что есть какой-то другой фрагмент, находящийся ближе к камере, чем он, то есть загораживающий его от света. Рис. 3 демонстрирует эту идею в двумерном случае.

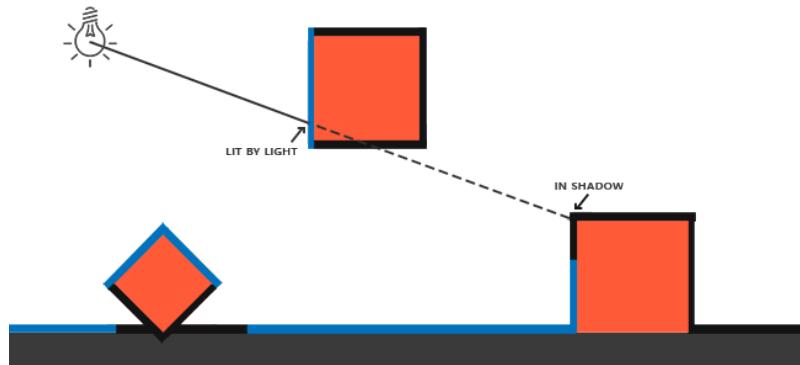


Рис. 3:
Освещённые и
затененные фраг-
менты сцены,
источник - [14].

После того, как мы отрисовали сцену с позиции источника света, у нас есть буфер цветов, который нам не пригодится, и буфер глубины, который дальше будет нужен, назовём его shadow map (или depth map). Заметим (достаточно тривиальный факт), что в буфере глубины для каждого фрагмента (пикселя) хранится минимальное расстояние (по оси z) от камеры до некоторого объекта сцены.

Теперь необходимо отрисовать сцену с исходной позиции камеры, и при этом уметь определять по фрагменту, находится он в тени, или нет.

Позиция фрагмента задаётся в пространстве мира (world space - глобальные координаты), сначала её необходимо преобразовать в пространство источника света - то есть умножить на view матрицу из прошлого шага, когда сцена отрисовывалась с позиции источника света.

Затем нужно из пространства источника света перевести координаты в экранные (то есть применить матрицу проекции, использующуюся на прошлом шаге).

Теперь мы получили нормализованные координаты x, y , которые можно использовать как текстурные координаты при обращении к буферу глубины (shadow map) и с помощью него получить глубину z_0 фрагмента, находящегося ближе всего к источ-

нику света. Также у нас есть z координата - это глубина текущего фрагмента, относительно источника света.

Если $z > z_0$, то есть объект ближе к источнику света, чем текущий фрагмент, а значит текущий фрагмент находится в тени (утверждение верно в обе стороны).

Если мы знаем, что фрагмент находится в тени, то можно в шейдере умножить посчитанное значение цвета фрагмента на константу < 1 , или визуализировать это как-то по другому.

Рис. 4 показывает, как лучи света соотносятся с значениями глубины, сохранёнными в depth map (shadow map), и как производится преобразование позиции фрагмента в пространство источника света.

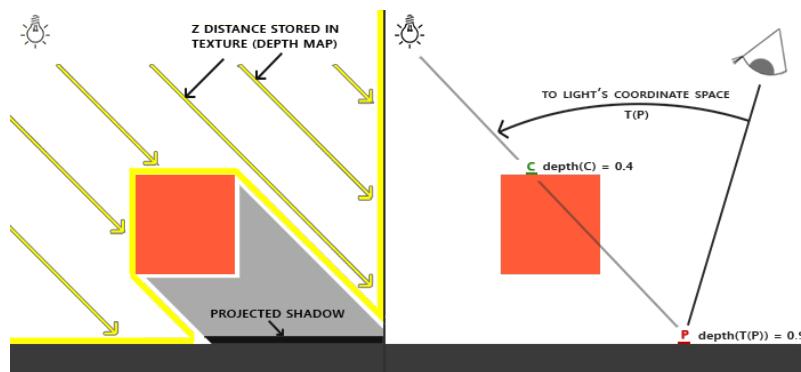


Рис. 4: Буфер глубины, преобразование в пространство источника света, источник - [15].

Рис. 5 - пример сцены, отрисованной с применением shadow mapping-a.

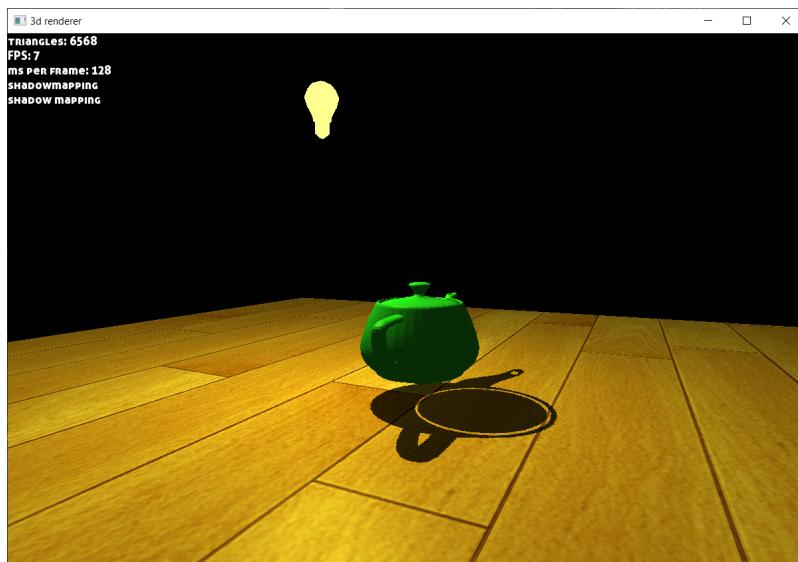


Рис. 5: Shadow mapping.

4.5 Карта нормалей (normal mapping)

Для обеспечения более реалистичного и детализированного рендеринга модели на экране часто применяются карты нормалей.

Большинство поверхностей в реальном мире содержат множество неровностей и никогда не являются плоскими на 100 процентов. В 3D сценах же наоборот, все объекты состоят из плоских треугольников, которые можно достаточно легко заметить, если они довольно большие и их не очень большое число.

В большинстве моделей освещения участвуют векторы нормалей, и для всей плоскости вектор нормали один и тот же. Но если для каждого фрагмента плоскости использовать вектор нормали, заданный в отдельной текстуре (карте нормалей), то с точки зрения освещения каждый фрагмент уже не будет иметь один и тот же вектор нормали, и это создаст иллюзию, что поверхность уже не является плоскостью и содержит множество неровностей (так как она освещается не так, как освещалась бы плоскость).

Такой приём позволяет значительно увеличить реалистичность моделей, добавить им больше деталей и создать иллюзию "глубины" для плоскостей.

После получения вектора нормали из карты нормалей его нужно перевести в пространство мира из так называемого касательного пространства (tangent space) треугольника, который мы отрисовываем (карта нормалей строится для поверхности, ориентированной вдоль плоскости ХОY, перпендикулярно оси Z, но отрисовываемый треугольник в сцене может иметь произвольную ориентацию). Это делается с помощью TBN матрицы (*tangent, bitangent, normal* - это базис касательного пространства, записанный в пространстве мира, см. рис. 6), которая вычисляется в вершинном шейдере.

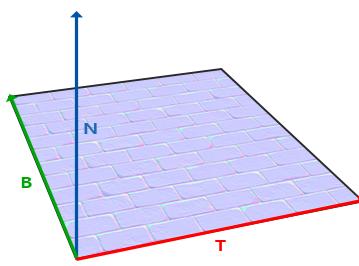


Рис. 6:
TBN вектора,
источник
- [11].

Если умножить вектор в касательном пространстве на матрицу TBN слева, то мы получим координаты данного вектора в про-

пространстве мира (вектора $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ перейдут в вектора *tangent*, *bitangent*, *normal*, при этом можно считать, что плоскость проходит через начало координат в пространстве мира, так как нас интересует только вектор нормали в пространстве мира, а он не зависит от положения плоскости, а только от её ориентации).

Рис. 7 - отрисованный куб (хотя и видна только одна грань) без и с normal mapping-ом.



Рис. 7: Blinn-Phong шейдер без и с normal mapping-ом.

4.6 Parallax mapping

Идея похожа на normal mapping, но здесь вместо карты нормалей используется карта смещения (displacement map). В этой текстуре задаётся смещение каждого фрагмента треугольника по глубине. С помощью parallax mapping-а по карте смещения, текущим текстурным (*uv*) координатам, и направлении вдоль которого мы смотрим на фрагмент (*viewPos – fragPos*) мы определяем новые текстурные (*uv*) координаты (намного более детально этот процесс описывается в [8]), и дальше уже используем их, когда обращаемся к диффузной карте (diffuse map) и карте нормалей (normal map). Вычисления производятся в касательном пространстве. Данный алгоритм позволяет добиться реалистично выглядящих и детализированных рендерингов моделей, но при просмотре под очень острыми углами может иметь небольшие артефакты.

Рис. 8 - отрисованный куб с normal mapping-ом, с parallax mapping-ом и без них (просто с обычным шейдером).

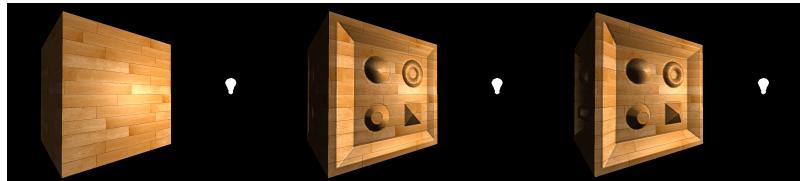


Рис. 8: Сравнение стандартного Blinn-Phong шейдера, шейдера с normal mapping-ом и шейдера с parallax mapping-ом.

4.7 Отложенное освещение и затенение (deferred shading)

4.7.1 Мотивация

Во всех перечисленных выше алгоритмах для каждого объекта его освещение рассчитывалось индивидуально, в момент отрисовки (сначала отрабатывал вершинный шейдер, а затем фрагментный - отвечающий за расчёт освещения), такой подход называется forward shading (прямое освещение и затенение).

В каждом фрагментном шейдере при этом все источники света обрабатываются последовательно, поэтому если в сцене M объектов и N источников света, то общая сложность расчёта освещения - $O(MN)$.

Также, в случае, когда пиксель ближе к камере перезаписывает сохраненный в буфере, вычисления, которые сделал предыдущий фрагментный шейдер по сути пропадают в пустую (в случае, когда нет blending-a). Для данного пикселя в буфере может быть вызван фрагментный шейдер много раз, но по итогу важен лишь самый последний записанный цвет.

Отложенное освещение и затенение решает данные проблемы: если использовать light volumes, deferred lighting, или tile-based deferred shading, то сложность расчёта освещения можно уменьшить до $O(M + N)$, вместо $O(MN)$. Также, при использовании отложенного освещения и затенения, для каждого пикселя в буфере фрагментный шейдер вызывается всего один раз в любых сценах.

На базе данного подхода также могут быть реализованы и другие продвинутые алгоритмы отрисовки - например SSAO.

4.7.2 Краткое описание

Основная идея состоит в том, чтобы "отложить" тяжелые вычисления (например расчёт освещения) на более позднюю стадию отрисовки.

Отложенное освещение и затенение состоит из двух этапов:

1. Геометрический этап (geometry pass). Сцена отрисовывается один раз, при этом в буфер для каждого фрагмента (пикселя) сохраняется информация о диффузной компоненте цвета, позиции, векторе нормали, и возможно других необходимых для последующего этапа освещения параметрах. Буфер называется G-buffer (от слова geometry).
2. Этап освещения (lighting pass). На данном этапе для каждого фрагмента (пикселя) в буфере с использованием информации, хранящейся в G-buffer-е, рассчитывается освещение (с помощью фрагментного шейдера).

4.8 Screen space ambient occlusion (SSAO)

4.8.1 Ambient occlusion

Из-за рассеивания лучей света в сценах очень редко бывают полностью тёмные участки, не освещаемые ничем. Свет отражается от различных поверхностей с разной интенсивностью, и из-за этого поверхности, которые освещены не напрямую, а отразившимися от других поверхностей лучами света, должны иметь некоторые различные ненулевые уровни освещенности. Для того, чтобы точно рассчитать уровни освещенности для всех поверхностей, можно запустить много лучей света, и промоделировать их рассеивание, но такой подход будет вычислительно достаточно сложным.

Ambient occlusion - один из способов аппроксимировать данный эффект непрямого освещения. Идея состоит в том, чтобы затенять поверхности, которые находятся близко друг к другу, а также внутренности углов, дыр и т.д., так как данные области другие объекты закрывают от света, и поэтому они выглядят темнее.

Несмотря на то, что ambient occlusion добавляет в сцену достаточно незаметные детали, в целом сцена становится более реалистичной.

4.8.2 SSAO

SSAO - это достаточно эффективный алгоритм, аппроксимирующий ambient occlusion, при этом выдающий довольно реалистичные результаты, поэтому он используется во многих 3D приложениях.

SSAO использует буфер глубины для того, чтобы определять затенение поверхности (идея чем-то похожа на shadow mapping). При этом все вычисления производятся в пространстве камеры (view space - координаты до применения перспективного преобразования) и в пространстве экрана (clip space - координаты после применения перспективного преобразование), отсюда и название - screen space.

SSAO построено на основе отложенного рендеринга, G-buffer состоит из позиций, нормалей и диффузной компоненты цвета фрагментов.

Для каждого фрагмента SSAO шейдер (фрагментный) вычисляет число от 0 до 1 - уровень "окклюзии" (occlusion - непроходимость), зависящее от значений буфера глубины точек (sample-ов), расположенных вокруг (в полусфере, ориентированной вдоль вектора нормали) данного фрагмента. Чем больше sample-ов имеют глубину больше, чем значение в буфере глубины, тем больше уровень окклюзии. Например, на рис. 9 слева фрагмент будет иметь окклюзию 0 (так как все сэмплы имеют минимальную глубину относительно камеры), а справа - 0.3, так как 3 сэмпла из 10 имеют не минимальную глубину, то есть есть объекты ближе к камере, загораживающие их.

Сравнение глубин фрагментов производится примерно также, как и в алгоритме shadow mapping-а, но нужно переводить позиции фрагментов в пространство экрана относительно камеры, а не относительно некоторого источника света, и можно использовать те же view и projection матрицы, что и обычно.

После того, как уровень окклюзии вычислен для всех фрагментов, его можно использовать в финальной стадии расчёта освещения.

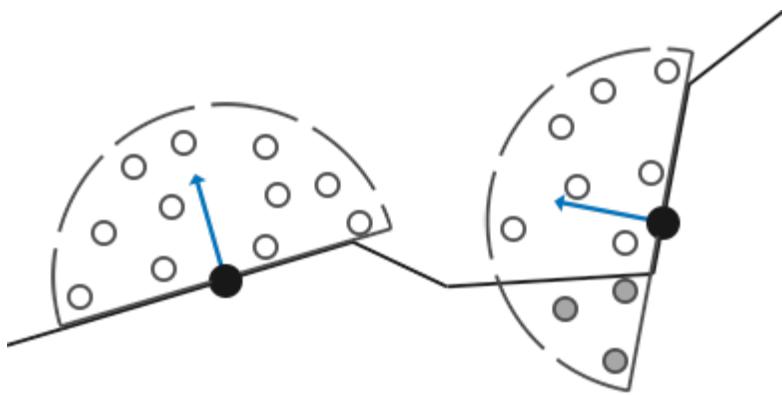


Рис. 9: Сэмплы вокруг фрагмента, источник - [18].

На рис. 10 - показано сравнение сцен с применением SSAO и без, а также текстура со значениями окклюзии.



Рис. 10: Сцена без SSAO, сцена с SSAO, текстура с вычисленным уровнем окклюзии.

5 Детали реализации приложения

5.1 Изменения в архитектуре

В основном архитектура приложения осталась такой же, как и в базовом рендерере (см. [25]), хотя и подверглась некоторым изменениям.

Класс `eng::Screen` распался на `eng::ColorAndDepthBuffer` и `eng::ProjectionInfo`.

`eng::ColorAndDepthBuffer` отвечает за хранение буфера цвета и глубины, при этом это шаблонный класс, и хранимый в нём цвет может быть произвольным.

`eng::ProjectionInfo` - хранит в себе матрицу проекции, а также различные параметры типа `nearPlaneDistance`, `farPlaneDistance`, `width`, `height` и т.д.

void `rasterizeTriangle(...)` теперь поддерживает `blending`, `discard` определенных цветов (например с нулевой *alpha* компонентой), а также преобразование цветов (например если shader выдаёт `glm::vec4` - 4 **float** значения, а буфер может хранить в

себе только `glm::u8vec4` - 4 **`unsigned char`** значения, то будет вызвано преобразование - clamp, hdr, и т.д.)

`eng::Renderer` не сам отрисовывает сцену, а делегирует эту функцию некоторому пайплайну, при этом можно в приложении выбирать для каждой сцены, какой пайплайн использовать.

Засчёт данного изменения можно удобно реализовать продвинутые алгоритмы отрисовки типа bloom, shadow mapping, отрисовку полупрозрачных объектов и другие.

Таким образом, у нас есть две возможности выбора режима отрисовки: для каждого объекта отдельно можно выбирать его тип (`FlatMesh`, `TextureMesh`, `PhongMesh`, `CubemapMesh` и т.д.), и глобально для сцены можно выбирать пайплайн отрисовки.

5.2 Основные классы в реализации

5.2.1 Application

`void Application::run()` - это точка входа в основной цикл работы приложения.

`Application` содержит в себе (через композицию) другие главные компоненты - `SFMLRenderer`, `UserInterface`, `CameraControl`, `std::vector<Scene>` (вектор сцен) и координирует их взаимодействие.

5.2.2 UserInterface

Отвечает за вывод отладочной информации на экран (SFML текст, SFML - [13]), а также за логику пользовательского интерфейса, реализованную с помощью ImGui - [6].

5.2.3 CameraControl

Умеет обрабатывать ввод пользователя с клавиатуры и мыши и соответствующим образом обновлять позицию и направление камеры.

5.2.4 Scene

Содержит в себе камеру (`Camera`), список источников света (вектор `PointLight`), список объектов сцены (вектор `std::variant`-ов

различных видов `Mesh<VertexShader, FragmentShader>`: `FlatMesh`, `PhongMesh`, и т.д.), своё имя и текущий пайплайн отрисовки (просто `std::string`). По сути это просто набор различных данных, самостоятельной функциональности в сцене нет.

5.2.5 Assets

Класс синглтон, он отвечает за хранение в одном месте текстур и 3D моделей, и позволяет получать ссылки на текстуры и модели по их имени файла (но при этом загружает их лениво, по запросу, и сохраняет в памяти на случай последующих обращений к ним).

5.2.6 SFMLRenderer

Относительно небольшая обёртка над классом `Renderer` - он позволяет отрисовывать сцену на экране SFML окна, а также сохранять скриншот сцены в файл. При отрисовке он сначала вызывает соответствующий метод класса `Renderer`, а затем вызывает набор SFML функций, которые принимают буфер пикселей в качестве аргумента.

5.2.7 Renderer

Содержит в себе набор различных pipeline классов (`DefaultPipeline`, `BlendingPipeline`, `BloomPipeline`, `SSAOPipeline` и т.д.). Когда вызывают функцию `PipelineResult Renderer::RenderScene(Scene& scene)`, `Renderer` смотрит на `std::string pipeline` в переданной сцене, и в зависимости от неё вызывает какой-то определенный пайплайн и возвращает результат его работы.

5.2.8 Pipeline

Есть много различных классов вида `[Name]Pipeline`, и они все имеют функцию `renderScene`, которая отрисовывает сцену. Самый простой `DefaultPipeline` просто обходит все объекты в сцене и через `std::visit` вызывает их методы отрисовки. Все пайплайны содержат один, или несколько `ColorAndDepthBuffer<...>`, а также могут содержать несколько произвольных буферов (`Buffer<T>`), где `Buffer<T>` - это просто type alias на `Vector2d<T>` - двумерный вектор.

5.2.9 Mesh<VertexShader, FragmentShader>

Класс, отвечающий за хранение параметров 3D модели, а также за свою отрисовку в заданном буфере. Параметризуется типами FragmentShader и VertexShader - классами, отвечающими за вычисление цвета пикселя (фрагмента) и за (в основном) применение к вершинам объекта различных преобразований (например некоторой модели, трансформирующей объект).

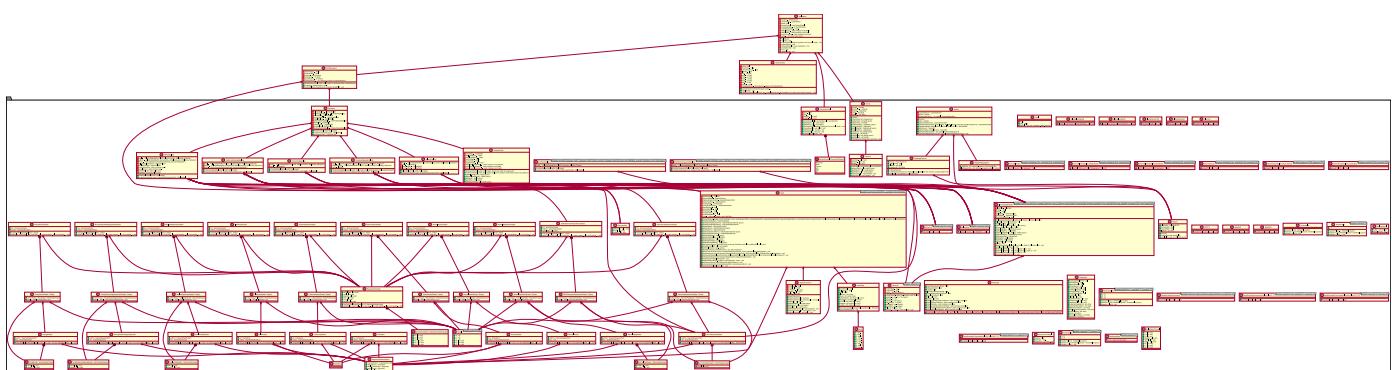
5.2.10 ColorAndDepthBuffer<...>

Хранит в себе два буфера - буфер цветов и глубины, имеет методы, которые выполняют depth check при записи пикселя в буфер. Также имеет ряд вспомогательных методов.

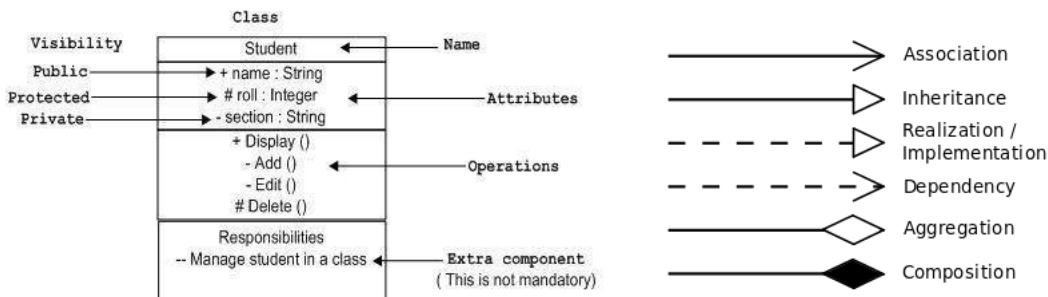
5.2.11 ...

Есть ещё много других классов, но основные были перечислены выше. Более подробно можно изучить структуру классов в проекте либо с помощью UML диаграммы ниже, либо заглянув в исходный код, который размещён в открытом доступе на гитхабе - [4].

5.3 Структурная диаграмма классов UML



5.3.1 Легенда и краткое описание элементов диаграммы



Картинки взяты отсюда [22] и отсюда [23]. В источнике [23] достаточно подробно описывается формат UML, если читатель хочет узнать больше про данный формат.

5.4 Тесты на производительность

Время отрисовки кадра сильно зависит от разрешения окна, количества объектов в сцене, площади экрана, покрываемой треугольниками, применяемых шейдеров и выбранного пайплайна отрисовки (стандартный пайплайн, bloom, shadow mapping и т.д.). Ниже приведены времена отрисовки для некоторых сцен. Проверить производительность можно, если запустить сцены с именами "bench*.scn". Тестирование производилось на ноутбуке с процессором intel pentium 4415U, 2.3 GHz, 2 физических (4 виртуальных) ядра, окно в разрешении 1200x800.

| Имя сцены | Время отрисовки кадра, ms |
|---------------------------|---------------------------|
| bench_teapot1_white | 38 |
| bench_teapot1_texture | 44 |
| bench_teapot1_phong | 72 |
| bench_teapot1_phong2 | 85 |
| bench_teapot1_normal | 261 |
| bench_teapot1_disp | 376 |
| bench_teapot1_bloom | 574 |
| bench_teapot1_transparent | 85 |
| bench_teapot1_ssao | 1325 |

Описание сцен:

- white - Просто белый чайник, шейдер возвращает константный цвет.

- texture - Чайник с текстурой (uv координаты для данной модели генерируются автоматически и имеют мало смысла, эта сцена, как и остальные, нужна просто для проверки производительности).
- phong - Blinn-Phong шейдер.
- phong2 - То же самое, только 2 источника света.
- normal - Normal mapping.
- disp - Parallax mapping.
- bloom - phong2 + bloom пайплайн (нужно выбрать bloom вместо default пайплайн-а).
- transparent - Полупрозрачный чайник (нужно выбрать blending with sort пайплайн).
- ssao - чайник с белой текстурой, Blinn-Phong шейдер, источников света нет, SSAO пайплайн (нужно выбрать SSAO вместо default пайплайн-а).

Чайник состоит из 6320 треугольников, во всех сценах кроме disp и normal расположен одинаково и занимает примерно треть окна по площади. В сценах disp и normal чайник расположен близко к камере и занимает практически всё пространство окна.

Скриншоты сцен - рис. 11 (сцены идут по порядку слева направо, сверху вниз):

6 Заключение

6.1 Результат

Главный результат работы - это интерактивное приложение, позволяющее пользователю взаимодействовать с трёхмерной сценой, задавать различные режимы отрисовки для объектов сцены, менять параметры освещения, источников света, камеры и экрана в реальном времени.

В 3D рендерере, на базе которого построено приложение, реализованы продвинутые алгоритмы 3D рендеринга: normal mapping,

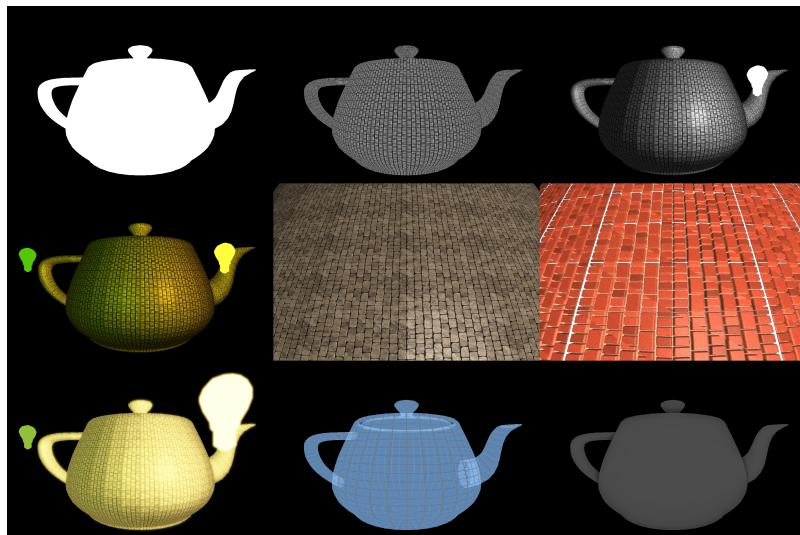


Рис. 11:
Скриншоты сцен,
на которых те-
стировалась
производительность.

parallax mapping, HDR, bloom, blending (поддержка полупрозрачных объектов), shadow mapping, deferred shading, SSAO. Результаты работы данных алгоритмов, и то, как их применение / неприменение визуально влияет на отрисованную сцену, можно наглядно увидеть с помощью использования приложения.

6.1.1 Качественные характеристики (просили привести)

- Число строк кода: 2475 в ".h" файлах, 2168 в ".cpp" файлах. Итого: 4643 строчки кода. Для подсчёта использовалась команда:

```
find . -name '*.php' | sed 's/.*/"/&/' | xargs wc -l
```

(вместо php cpp и h)

- Объём кода в килобайтах: 147.
- Но необходимо учитывать, что данный проект - это продолжение проекта прошлого года, поэтому часть строк уже существовала на момент начала работы над проектом, и частично строки переписывались, рефакторились и т.д. Также здесь в количестве строк учитываются комментарии, пустые строки, строки с фигурными скобками. Поэтому не уверен, что по данной статистике можно сделать какие-то определённые выводы.

6.2 Направления дальнейших разработок

Как было написано в аннотации, в сфере 3D рендеринга существует множество статей и различных алгоритмов, поэтому дальнейшую разработку можно посвятить реализации ещё не затронутых методов 3D рендеринга. Также всегда можно заняться оптимизацией производительности (хотя аппаратные рендереры всё равно будут значительно эффективнее), улучшению качества кода, и т.д.

Библиографический список

- [1] Angelo1211/softwarerenderer. URL: <https://github.com/Angelo1211/SoftwareRenderer>.
- [2] bytecode77/fastpix3d. URL: <https://github.com/bytecode77/fastpix3d>.
- [3] Dawoodoz/dfpsr. URL: <https://github.com/Dawoodoz/DFPSR>.
- [4] github репозиторий проекта. URL: <https://github.com/asmorodinov/3d-renderer-from-scratch/tree/dev>.
- [5] glm. URL: <https://github.com/g-truc/glm>.
- [6] ImGui. URL: <https://github.com/ocornut/imgui>.
- [7] kosua20/heredragons. URL: <https://github.com/kosua20/heredragons>.
- [8] learnopengl.com. URL: <https://learnopengl.com/>.
- [9] martinresearch/deodr. URL: <https://github.com/martinResearch/DE0DR>.
- [10] Mesa. URL: <https://www.mesa3d.org/>.
- [11] normal mapping tbn vectors image source. URL: https://learnopengl.com/img/advanced-lighting/normal_mapping_tbn_vectors.png.
- [12] obj. URL: https://en.wikipedia.org/wiki/Wavefront_.obj_file.

- [13] Sfml. URL: <https://www.sfml-dev.org/>.
- [14] shadow mapping image 1 source. URL: https://learnopengl.com/img/advanced-lighting/shadow_mapping_theory.png.
- [15] shadow mapping image 2 source. URL: https://learnopengl.com/img/advanced-lighting/shadow_mapping_theory_spaces.png.
- [16] skywind3000/mini3d. URL: <https://github.com/skywind3000/mini3d>.
- [17] skywind3000/renderhelp. URL: <https://github.com/skywind3000/RenderHelp>.
- [18] Ssao image source. URL: https://learnopengl.com/img/advanced-lighting/ssao_hemisphere.png.
- [19] ssloy/tinyraycaster. URL: <https://github.com/ssloy/tinyraycaster>.
- [20] ssloy/tinyparser. URL: <https://github.com/ssloy/tinyparser>.
- [21] stb_image. URL: https://github.com/nothings/stb/blob/master/stb_image.h.
- [22] uml image 1 source. URL: <https://coderlessons.com/tutorials/akademicheskii/uchit-uml/uml-osnovnye-notatsii>.
- [23] uml image 2 source. URL: https://www.wikiwand.com/en/Class_diagram.
- [24] zauonlok/renderer. URL: <https://github.com/zauonlok/renderer>.
- [25] Отчёт по КР за 2 курс. URL: https://github.com/asmorodinov/3d-renderer-from-scratch/blob/dev/term_papers/2%20course/paper.pdf.
- [26] Список software 3d renderer-ов на github. URL: <https://github.com/topics/software-rendering>.

- [27] Samuel R. Buss. *A Mathematical Introduction with OpenGL*. Cambridge University Press, 2003.
- [28] Eric Lengyel. *Mathmatics for 3d game programming and computer graphics*. Course Technology PTR, 2012.