

Продвинутый 3D renderer

Программный проект

Смородинов Александр, БПМИ196

Научный руководитель:
доцент, кандидат физико-математических наук,
Трушин Дмитрий Витальевич

Факультет Компьютерных Наук
НИУ ВШЭ (Москва)

Июнь 2022

Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

- **3D рендеринг** - преобразование 3D моделей в 2D изображения.
- Для выполнения рендеринга может использоваться:
 - CPU (программный рендеринг).
 - CPU + GPU (аппаратный рендеринг).

Программный vs аппаратный рендеринг

- Программный рендеринг:
 - + Аппаратная независимость
 - Производительность

Программный vs аппаратный рендеринг

- Программный рендеринг:
 - + Аппаратная независимость
 - Производительность
- Аппаратный рендеринг:
 - + Производительность
 - Аппаратная независимость

Содержание

- 1 Предметная область
- 2 Актуальность задачи**
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

① Востребованность:

- 3D рендеринг широко распространен, используется в 3D моделировании и анимации, в 3D и VR симуляциях, в компьютерных играх. Данная область относительно молодая и постоянно развивается.
- Программный рендеринг применяется в случаях, когда важна аппаратная независимость и производительность не является критическим требованием.

Актуальность задачи

1 Востребованность:

- 3D рендеринг широко распространен, используется в 3D моделировании и анимации, в 3D и VR симуляциях, в компьютерных играх. Данная область относительно молодая и постоянно развивается.
- Программный рендеринг применяется в случаях, когда важна аппаратная независимость и производительность не является критическим требованием.

2 Нерешённость:

- Как будет показано далее, в открытом доступе программных рендереров, реализующих **продвинутые** алгоритмы отрисовки, достаточно мало.

Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

Цель и задачи

1 Цель:

- Изучить и реализовать продвинутые алгоритмы отрисовки и таким образом дополнить и улучшить базовый 3D рендерер, реализованный в прошлом году.

Цель и задачи

1 Цель:

- Изучить и реализовать продвинутые алгоритмы отрисовки и таким образом дополнить и улучшить базовый 3D рендерер, реализованный в прошлом году.

2 Основные задачи:

- Изучить различные продвинутые алгоритмы отрисовки.
- Реализовать данные алгоритмы и интегрировать их в существующую реализацию 3D рендерера.
- Протестировать реализованные алгоритмы.

Цель и задачи

1 Цель:

- Изучить и реализовать продвинутые алгоритмы отрисовки и таким образом дополнить и улучшить базовый 3D рендерер, реализованный в прошлом году.

2 Основные задачи:

- Изучить различные продвинутые алгоритмы отрисовки.
- Реализовать данные алгоритмы и интегрировать их в существующую реализацию 3D рендерера.
- Протестировать реализованные алгоритмы.

3 Дополнительные задачи:

- Рефакторинг и улучшение качества кода 3D рендерера.
- Оптимизация, улучшение производительности.
- Написание сопроводительной документации, описание деталей реализации, изложение теоретических основ алгоритмов.

Формальная постановка задачи

- 1 Задача - реализовать следующие алгоритмы:
 - Normal mapping, parallax mapping, HDR, tone mapping, gamma correction, bloom, рендеринг полупрозрачных объектов (blending), shadow mapping, deferred shading, SSAO.

Формальная постановка задачи

- ① Задача - реализовать следующие алгоритмы:
 - Normal mapping, parallax mapping, HDR, tone mapping, gamma correction, bloom, рендеринг полупрозрачных объектов (blending), shadow mapping, deferred shading, SSAO.
- ② 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 (поддерживаются 3D модели в формате waveform obj).

Нефункциональные требования

- Язык программирования - C++ (стандарт C++17).
- Используемые библиотеки (статическая линковка):
 - glm 0.9.9
 - SFML 2.5.1
 - ImGui v1.83
- Система контроля версий - git. Репозиторий:
<https://github.com/asmorodinov/3d-renderer-from-scratch/tree/dev>.
- Линтер/форматтер - clang format, настройки основаны на google codestyle.
- Для сборки проекта используется IDE Microsoft Visual Studio 2019.

Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

Обзор существующих решений

- Mesa - <https://www.mesa3d.org/>.
 - Реализация набора видеодрайверов для основных графических API.
 - Программная реализация OpenGL 3.1.
 - Широкое распространение на устройствах с GNU/Linux, *BSD и др., в качестве основы графического стека.

Обзор существующих решений

- Mesa - <https://www.mesa3d.org/>.
 - Реализация набора видеодрайверов для основных графических API.
 - Программная реализация OpenGL 3.1.
 - Широкое распространение на устройствах с GNU/Linux, *BSD и др., в качестве основы графического стека.
- Другие аналоги - самые популярные github репозитории (по количеству звёзд) с темой software rendering:
 - ssloy/tinyrenderer, zauonlok/renderer, kosua20/heredragons, skywind3000/mini3d, ssloy/tinyraycaster, skywind3000/RenderHelp, Angelo1211/SoftwareRenderer, martinResearch/DEODR
 - bytecode77/fastpix3d, Dawoodoz/DFPSR
 - Ни один из перечисленных выше аналогов не реализует все продвинутые алгоритмы, указанные на прошлых слайдах. Хотя все аналоги достаточно интересны (особенно последние два), и немного более подробно я их описывал в отчёте.

Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

- **LDR** - все компоненты цвета лежат в ограниченном диапазоне ($[0, 1]$, или $0, \dots, 255$).
- **HDR** - компоненты цвета ограничены по величине только максимальным значением типа (например `float32`).
- Из HDR в LDR цвет преобразуется с помощью `tone mapping`.

- Отрисовываем сцену в HDR буфер.
- Выделяем из HDR буфера "яркие" пиксели.
- Размыываем яркие пиксели.
- Прибавляем полученный буфер к исходному.
- Делаем tone mapping.

Скриншот



Скриншот



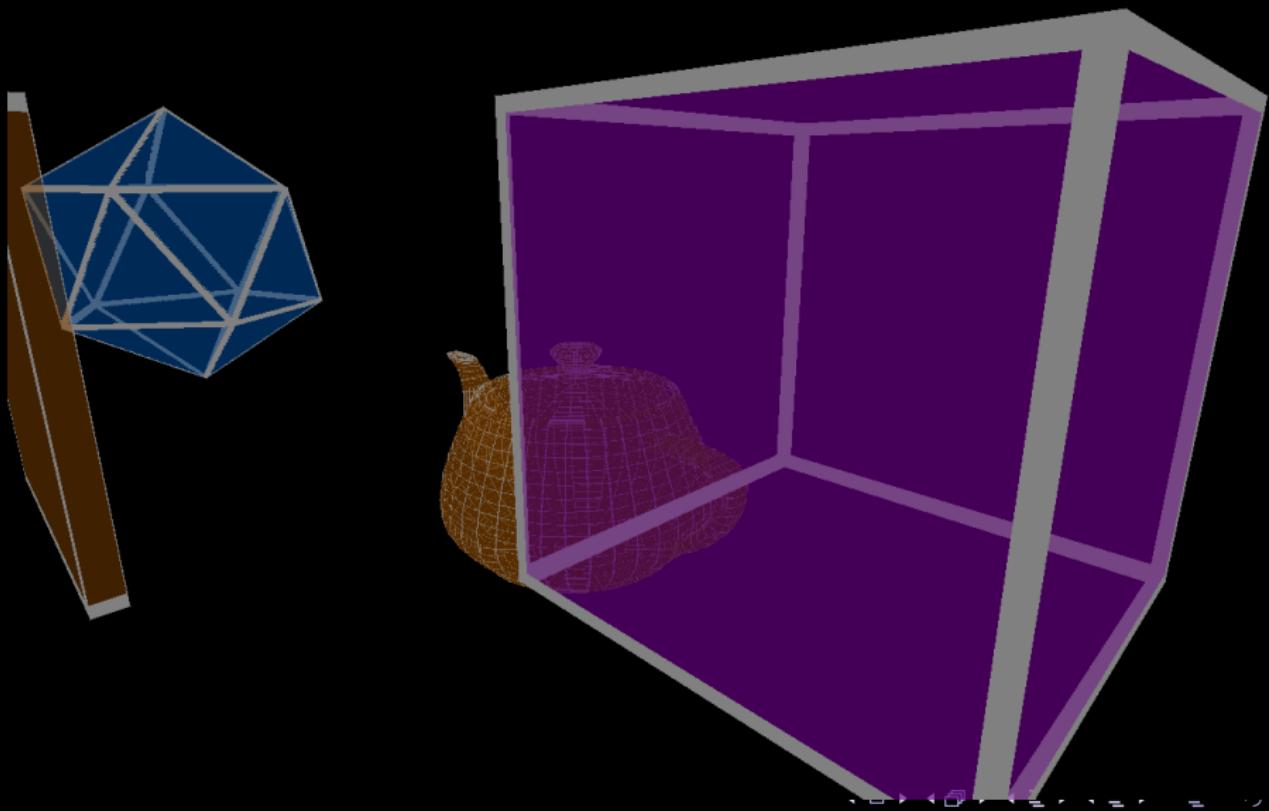
Скриншот



Отрисовка полупрозрачных объектов, blending

- Объекты отрисовываются от более дальних к более близким.
- Когда какой-то цвет в буфере (source) перезаписывает цвет уже находящегося в буфере пикселя (destination), вызывается blend функция и результат сохраняется в буфер.
- Blend функция принимает 2 цвета и возвращает результат их "смешивания" (тоже цвет). Например $f(s, d) = s * s.a + d * (1 - s.a)$, s - source цвет, d - destination цвет, $x.a$ - значение альфа канала цвета x .

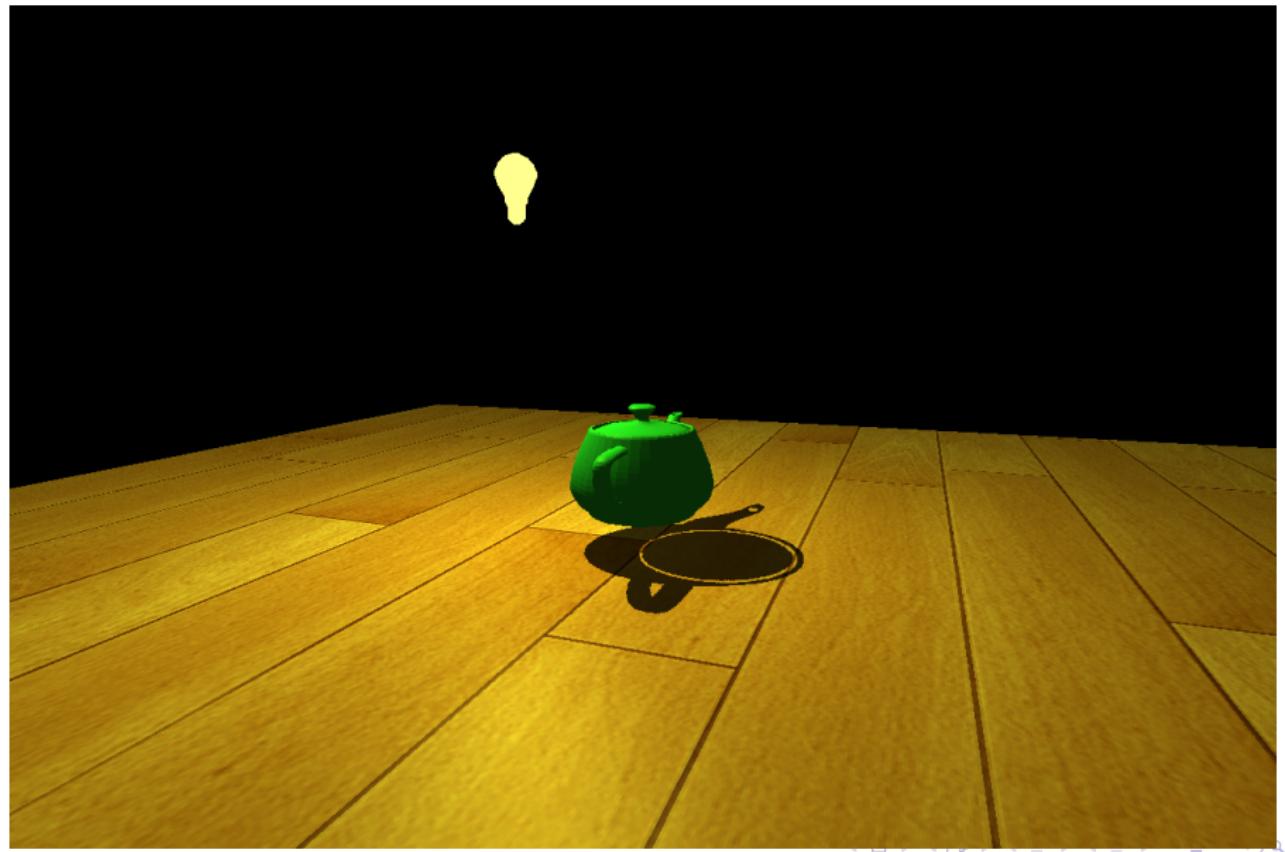
Скриншот



Shadow mapping

- Сначала отрисовываем сцену с позиции источника света (цвета нас не интересуют, а только буфер глубины).
- С помощью буфера глубины с прошлого шага можно определять, какие фрагменты сцены находятся в тени, а какие - нет.
- Отрисовываем сцену, используя буфер глубины (сейчас уже с цветами).

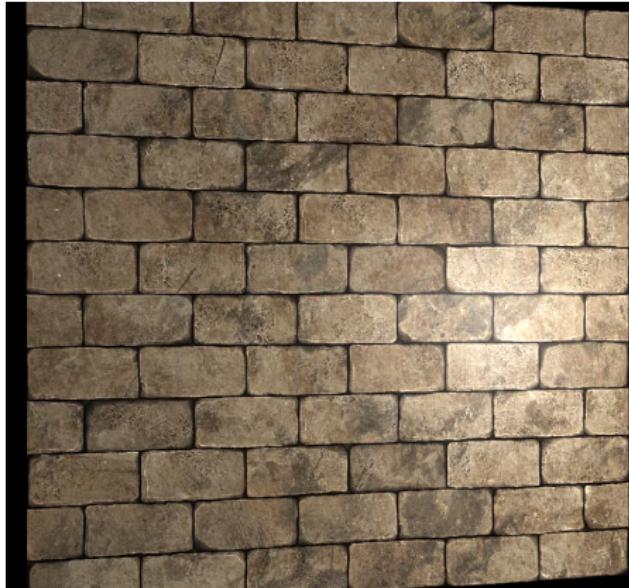
Скриншот



Normal mapping

- В шейдерах используем не один вектор нормали для всего примитива (треугольника).
- Вместо этого, мы сначала получаем вектор нормали для текущего фрагмента из карты нормалей (normal map), аналогично тому, как используются обычные текстуры для диффузной составляющей цвета.
- Преобразуем вектор из касательного пространства в пространство мира.
- Далее выполняем стандартные вычисления с полученным вектором (например Blinn-Phong затенение).

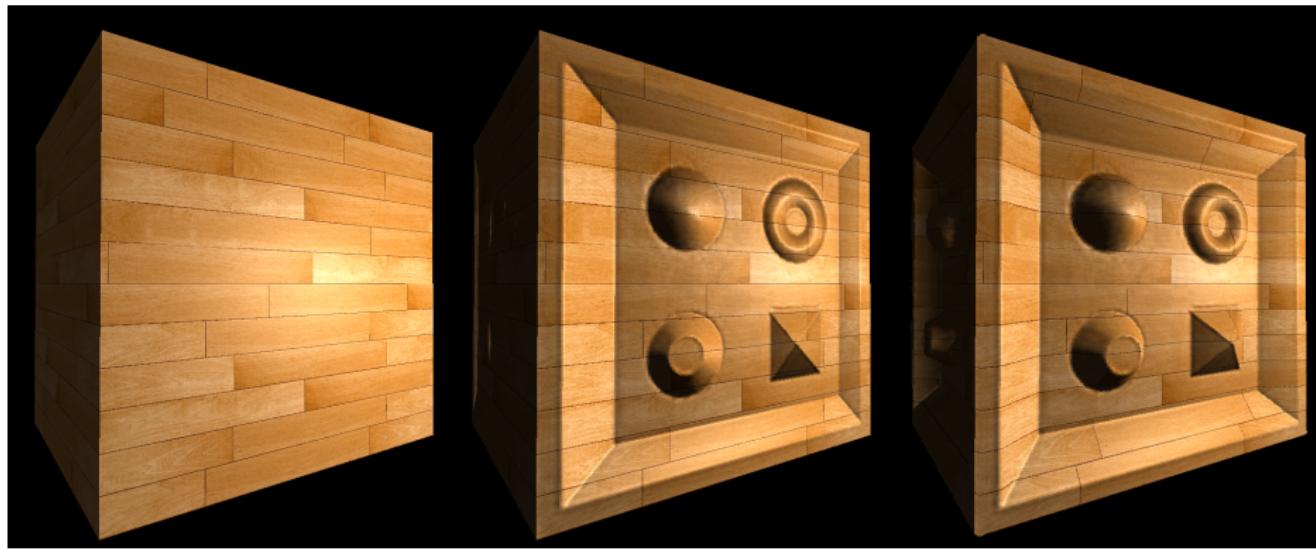
Скриншот



Parallax mapping

- Идея частично похожа на normal mapping.
- Но для каждого фрагмента теперь задаётся смещение из карты смещения (displacement map).
- По смещению и текстурным координатам фрагмента вычисляются новые текстурные координаты.
- Для всех текстур (normal map, diffuse map) мы используем новые текстурные координаты.

Скриншот



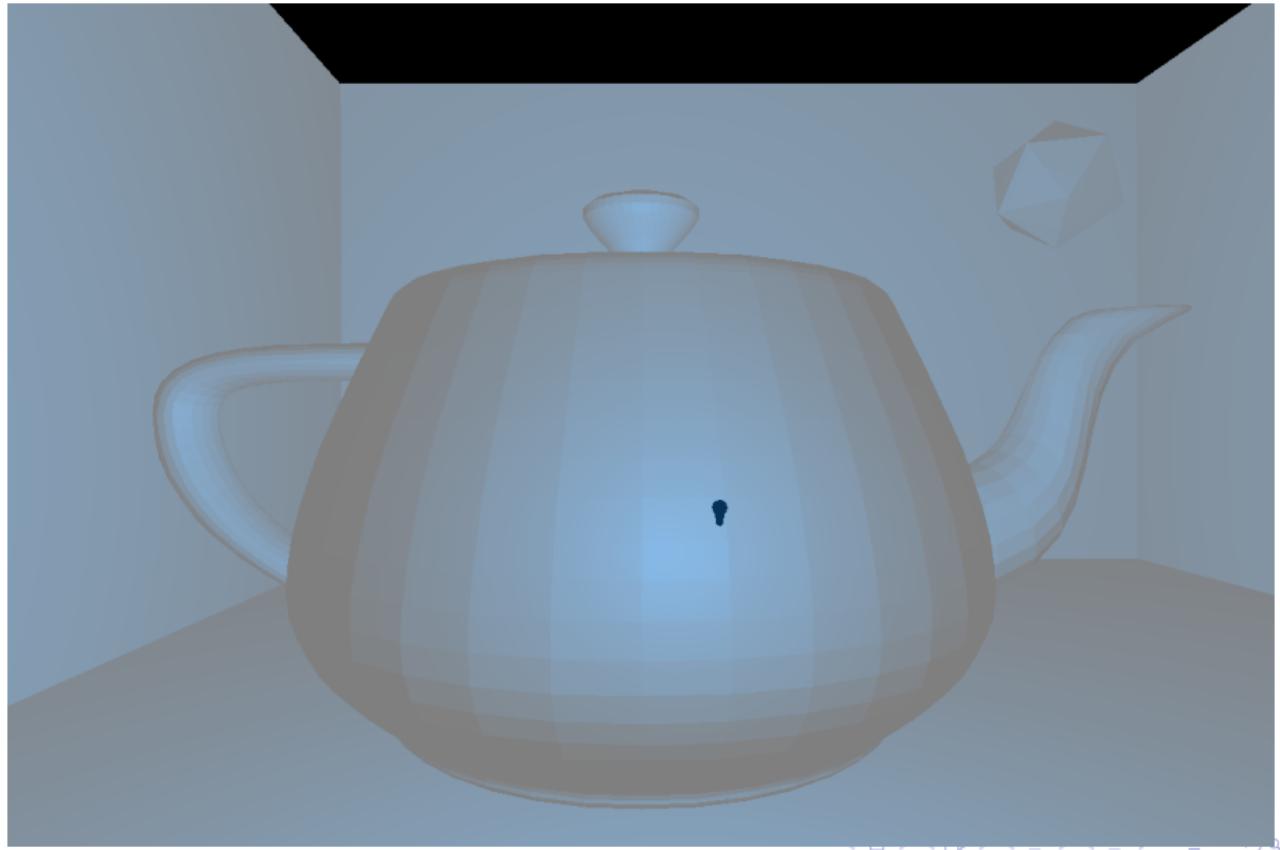
Отложенное освещение и затенение (deferred shading)

- Отрисовываем сцену в G-buffer. Элементами G-buffer-а могут например быть вектора позиции фрагмента, диффузной компоненты цвета, нормали и другие.
- Рассчитываем освещение с помощью шейдера, которому на вход подаются элементы G-buffer-а.

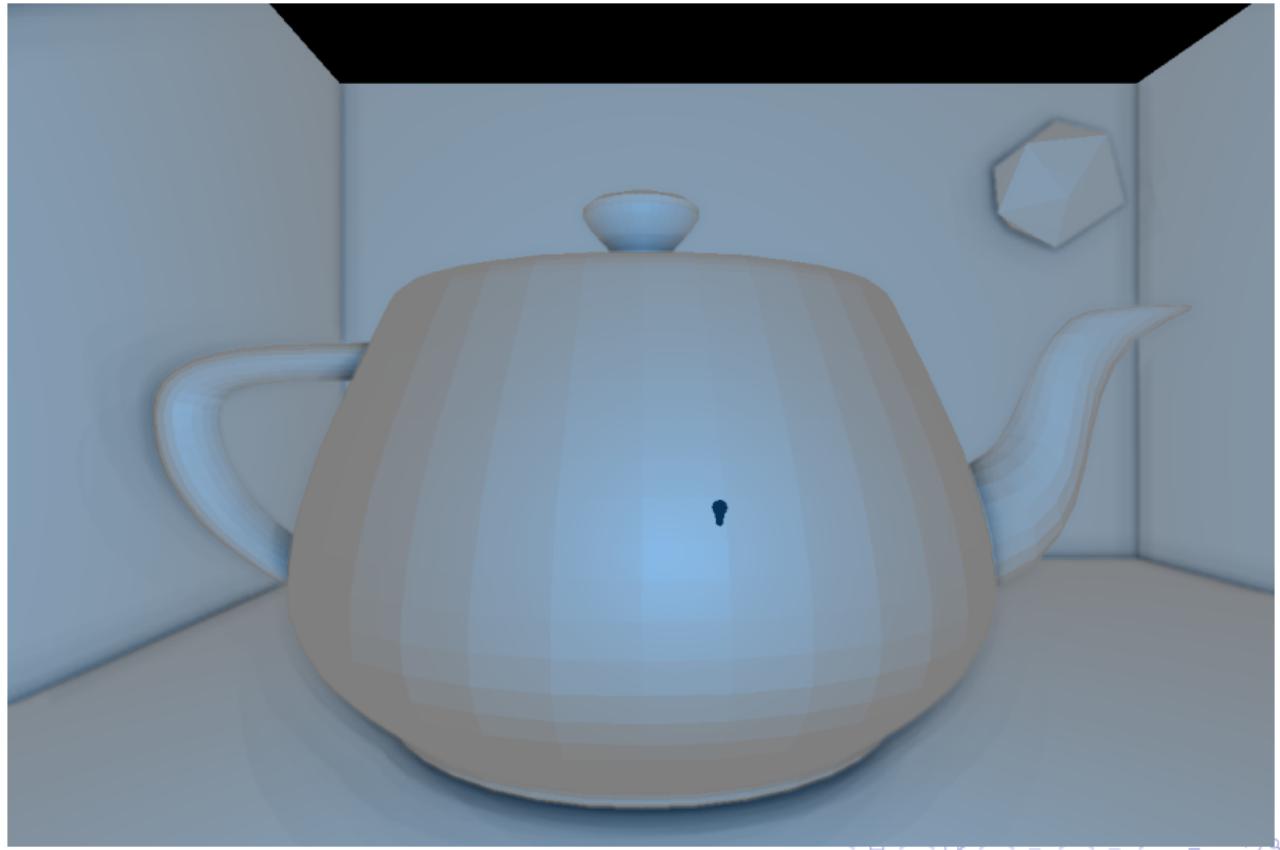
Screen space ambient occlusion (SSAO)

- SSAO построено на основе отложенного рендеринга.
- SSAO шейдер для каждого фрагмента вычисляет уровень "окклюзии" - доли сэмплов, которых не видно с позиции камеры (то есть находящихся за некоторым объектом), рассматриваемых вокруг фрагмента.
- В зависимости от уровня окклюзии меняем величину ambient освещения фрагмента.

Скриншот



Скриншот



Содержание

- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

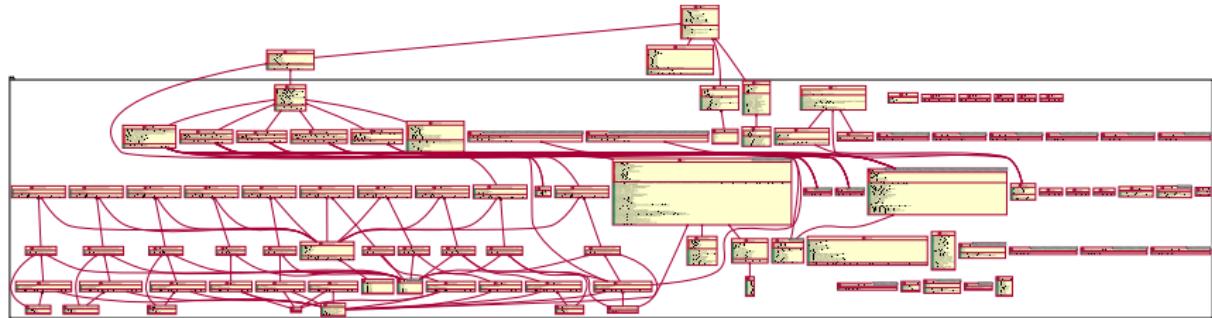
Основные классы

- *Application*
 - *void run()* - точка входа в event loop.
 - Отвечает за координацию своих полей - *SFMLRenderer*, *UserInterface*, *CameraControl*, *std::vector<Scene>*
- *UserInterface* - отвечает за пользовательский интерфейс.
- *CameraControl* - отвечает за обновление позиции и направления камеры (обрабатывает ввод с клавиатуры и мыши).
- *Scene* - хранит в себе вектор объектов и тип пайплайна (строку).
- *Assets* - класс синглтон, хранит текстуры и модели, лениво загружая их по запросу.
- *SFMLRenderer* - обёртка над *Renderer*, умеющая копировать буфер в SFML текстуру, которая отрисовывается на экране.

Основные классы

- *Renderer* - хранит в себе различные пайплайны и по переданной сцене определяет, какому пайплайну отрисовывать её.
- *Pipeline* - есть много классов пайплайнов, каждый из которых отвечает за реализацию конкретного алгоритма отрисовки сцены.
- *Mesh<VertexShader, FragmentShader>* - основной объект сцены, умеет отрисовывать себя в буфере, используя переданные шейдеры.
- *ColorAndDepthBuffer<...>* - набор из двух буферов (глубины и цвета), имеющий ряд полезных вспомогательных функций для работы с ним.
- ...

Структурная диаграмма классов UML



Содержание

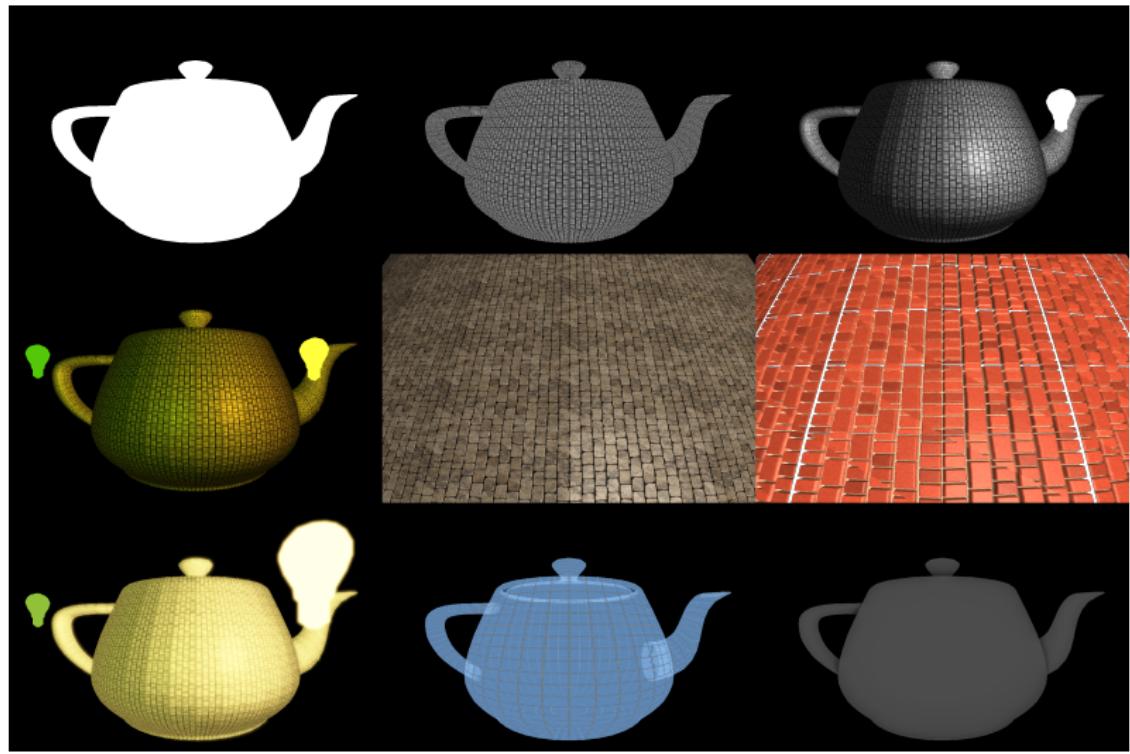
- 1 Предметная область
- 2 Актуальность задачи
- 3 Цель и задачи
- 4 Обзор существующих решений
- 5 Краткое описание реализованных алгоритмов
- 6 Детали реализации
- 7 Тестирование и результаты

Тесты на производительность

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

Тесты на производительность



Характеристики ПО

- Число строк кода: 2475 в ".h" файлах, 2168 в ".cpp" файлах. Итого: 4643 строчки кода.
- Объём кода в килобайтах: 147.
- Но необходимо учитывать, что данный проект - это продолжение проекта прошлого года, поэтому не уверен, что по данной статистике можно сделать какие-то определённые выводы.

Результаты

- Все поставленные задачи были выполнены.
- Главный результат работы - это интерактивное приложение.
- Были реализованы продвинутые алгоритмы 3D рендеринга: normal mapping, parallax mapping, HDR, bloom, blending (поддержка полупрозрачных объектов), shadow mapping, deferred shading, SSAO.

Спасибо за внимание.
Готов ответить на вопросы.