Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет Программной инженерии и компьютерной техники (ФПИ и КТ)
Образовательная программа: Системное и прикладное программное обеспечение
Направление подготовки (специальность): 09.04.04, Программная инженерия

О Т Ч Е Т

о научно-исследовательской практике

Тема задания: *Методы сетевого взаимодействия между системой синтеза изображений и сценой для системы федеративного рендеринга*

Обучающийся: *Кульбако Артемий Юрьевич, P4215*

Научный руководитель: *Потемин Игорь Станиславович, кандидат технических наук, доцент, ведущий научный сотрудник*

Дата: 26.01.2024

Санкт-Петербург
2024

ОГЛАВЛЕНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

GPU (англ. graphics processing unit) — отдельное устройство персонального компьютера, выполняющее графический рендеринг. Отличительными особенностью является архитектура, максимально нацеленная на увеличение скорости расчёта текстур и сложных графических объектов.

CPU (англ. central processing unit) — электронный блок либо интегральная схема, исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера.

R&D (англ. research and development) – совокупность работ, направленных на получение новых знаний и практическое применение при создании нового изделия или технологии.

Модель – это описание объектов или явлений реального мира, которые можно визуализировать (в контексте процесса рендеринга).

Узел – устройство, соединённое с другими устройствами как часть компьютерной сети. Узлами могут быть компьютеры, мобильные телефоны, любые другие вычислительные устройства. Термин узел основывается на аналогии с прототипом компьютерной сети: реальная сеть, например рыболовная, состоит из нитей, соединяющих друг с другом множество узлов.

Протокол – набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами.

Брокер сообщений – приложение, которое преобразует сообщение по одному протоколу от приложения-источника в сообщение протокола приложения-приёмника, тем самым выступая между ними посредником.

ВВЕДЕНИЕ

Практика – важнейший этап образовательного процесса, направленный на проверку и закрепление компетенций обучающегося, полученных в процессе академического обучение.

Целью третьего этапа научно-исследовательской практики является создание прототипа продукта, который будет развит в рамках выполнения ВКР. Работа над НИР велась совместно с Лопатиным Алексеем Владимировичем (P4216) и двумя другими командами, которые отвечали за разработку компонентов системы.

Инструктаж обучающегося

В первую очередь, был проведён инструктаж студентов по работе с страницей "Практика" в информационной системе университета, определены основные этапы индивидуального задания и сроки их выполнения в днях (с учётом инструктажа и оформления отчётных документов).

НАУЧНО-ИССЛЕДОВАТЕЛЬСКАЯ РАБОТА

Введение в предметную область

Рендеринг – процесс получения изображения по модели (сцене) с помощью компьютерной программы. Для достижения максимального сходства генерируемого изображения с наблюдаемой действительностью, необходимо достоверно смоделировать физику света, в-первую очередь, распространение световых лучей в пространстве. Это сложная вычислительная задача, ввиду аналоговой природы нашего мира, и дискретной сущности компьютеров. Чем больше физических явлений будет воспроизводить алгоритм рендеринга, тем точнее будет полученное изображения, что требуется очень мощное оборудование.

Так как лучи света распространяются независимо друг от друга, то алгоритм рендеринга можно распараллелить, к примеру, просчитывая путь отдельного луча света на отдельном вычислителе (таким вычислителем может быть CPU или GPU). Если мощности отдельного вычислителя недостаточно для выполнения рендеринга за требуемое время, необходимо подключить к системе несколько вычислителей, а данными обмениваться по сети. Такой подход называется распределённым рендерингом.

В современном мире, рендеринг применяется во множестве сфер: рекламе, R&D, строительстве. Таким образом, модель, для которой осуществляется рендеринг, может являться коммерческой/государственной или иной тайной, но само по себе сгенерированное изображение может быть открытым. К примеру, при проектировании метро, можно продемонстрировать гражданам внешний вид вестибюля станции, но предоставлять модель станции нельзя. Ввиду экономических причин, не каждый бизнес, которому потребуется сгенерировать изображение, будет готов полностью закупить дорогостоящее вычислительное оборудование и программное обеспечение, связать его в сеть. Таким образом, возникает

необходимость отдать "секретные сведения" третьему лицу, специализирующемуся на услугах рендеринга, на что не каждый заказчик может согласиться. Для решения проблемы нужно применить принципы федеративных вычислений к рендерингу.

Федеративные вычисления подразумевают, что каждый из узлов распределённой системы обладает минимальным и достаточным количеством информации для выполнения своей задачи, но не получает все данные полностью. Таким образом, снижается риск утечки конфиденциальных данных, а эффективность передачи наоборот – увеличивается (за счёт минимизации объёма). В контексте рендеринга, классический распределённый подход подразумевает, что каждый узел вычислительной сети обладает своей собственной копией модели. Федеративный подход подразумевает, что модель будет хранится в едином экземпляре, к примеру, на сервере заказчика. Узлы же, находящиеся под контролем исполнителя, смогут выполнить самые "тяжёлые" вычисления, но итоговая визуализация сцены будет возможна исключительно на сервере заказчика, и не потребует больших мощностей.

Проектирование формата сообщения и событий

Высокоуровневая архитектура федеративной системы представляет из себя два компонента:

1. Компонент рендеринга (рендерер) – реализует алгоритмы моделирования света; запрашивает необходимые данные о сцене у компонента клиента. Таких компонентов может быть несколько, ведь их поведение никак не зависит от данных, которые им будут отданы.
2. Клиентский компонент – отвечает за хранение и отдачу данных сцены, таких как: геометрия вершин, позиция и направление виртуальной камеры, источники света и их характеристики,

материалы поверхностей и сред, текстуры. Компонентов клиента также может быть несколько, а каждый из них может хранить только один тип данных из перечисленных выше.

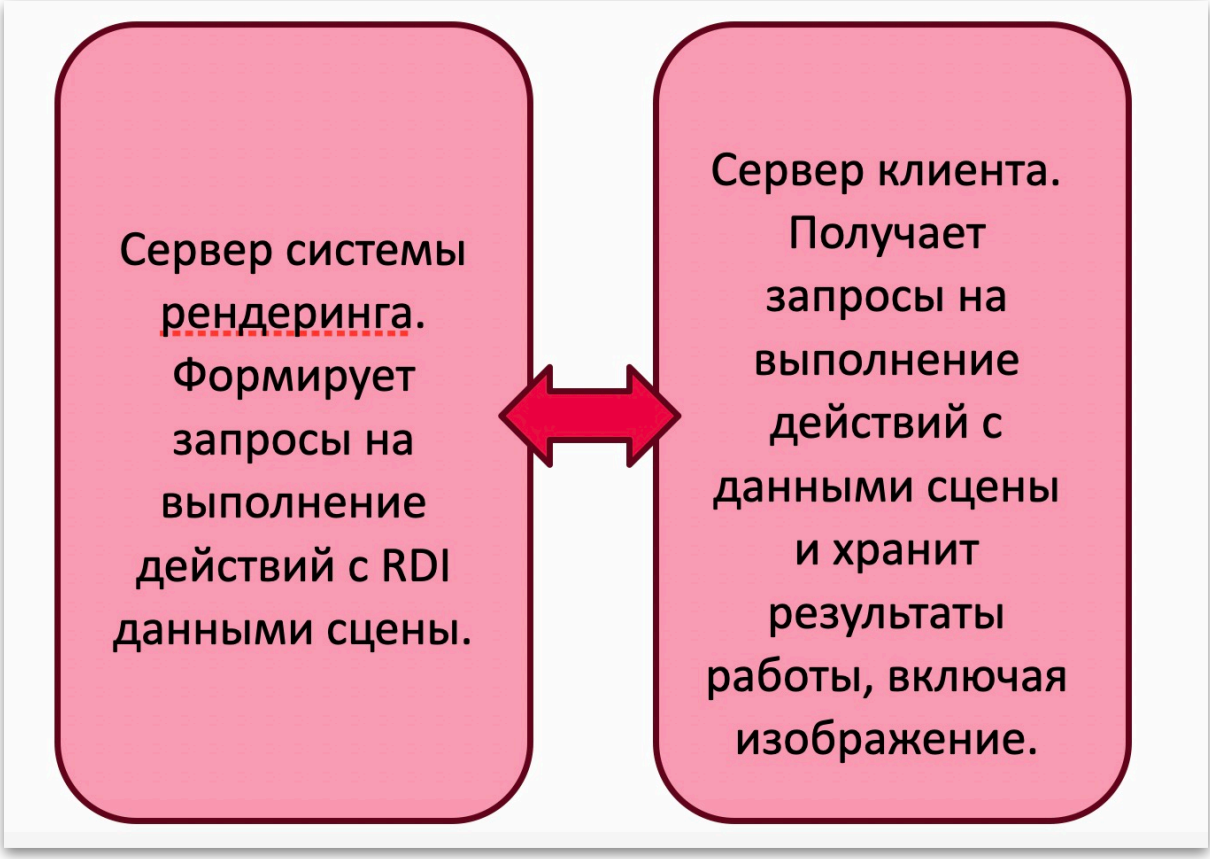

Рис. 1 - Архитектура компонентов

Таким образом, масштабирование системы возможно с обоих сторон: как с стороны заказчика, который может хранит разнотипные данные на разных серверах (или заказчиков может быть несколько), так и со стороны исполнителя, который может развернуть множество вычислительных узлов. В зависимости от требуемого заказчиком качества визуализируемого изображения, сроков, а также количества заказов, узлы можно свободно конфигурировать как в одну большую сеть, так и в множество маленьких.

Реализацией непосредственно алгоритмов рендеринга и хранения модели занимаются другие команды под руководством научного руководителя, наша же часть заключалась в создании кода сетевого взаимодействия для обоих модулей. В первые очередь, совместно с другими командами, мы определили класс `Scene`, который хранит в себе информацию

о модели: геометрию сцены, её связи с объектами материалов и источников света, а также доступ на указатель сцены в памяти совместимый с библиотекой `Intel Embree` – это open-source библиотека нужна для выполнения различных прикладных операций.

Второй моделью, который необходимо будет обмениваться, стала `Camera`. Она позволяет задать позицию виртуального наблюдателя относительно сцены.

Рендерер будет обращаться к открытым интерфейсам этих классов в клиентском компоненте и получать необходимую информацию.

И наконец, нужно было определить модель данных, представляющую из себя результат работы компонентов рендеринга. Таким результатом является байтовой массив цветов пикселей, который мы назвали `HdrReplay`.

Модель была согласована всеми тремя командами, работающими над проектом, и были добавлена в репозиторий. Код в Приложении 1.

Выбор сетевых протоколов

Следующим этапом мы выбирали сетевой протокол взаимодействия между узлами системы. Ключевыми критериями при выборе были:

- Удобство программирования распределённых систем из более чем 2 узлов
- Безопасность
- Понятный формат сериализации

Поиск в интернете определил следующих кандидатов на место подходящего сетевого протокола:

- `HTTP`: классический, удобный и понятный вариант обмена информацией в системах с клиент-серверной архитектурой. Но использование этого протокола требует держать на каждом узел сервер для входящий сообщений, а также самостоятельно писать модуль для распределения сообщений с клиента между узлами.

- MQTT: лучше подходит для задачи обмена сообщениями между множеством узлом ввиду того, что этим распределением будет заниматься брокер сообщений. Минус протокола заключается в том, что брокер сообщений является отдельным независимым программным обеспечением и не может быть легко встроен в наши модули. Его необходимо конфигурировать отдельно и следить за ним. Если по каким-либо причинам он перестанет работать, все остальные узлы станут бесполезны, так как не смогут получать данные и команды от других узлов.
- Java RMI: интерфейс для удалённого вызова функций с одного клиента на другом. Существует несколько реализаций интерфейса. Программно, такие вызовы очень удобны, так как требуют от программиста лишь вызвать метод определённого класса на одном узле, а другой узел это вызов неявно перехватит. Выглядит как подходящий вариант. Главный минус – все узлы распределённой системы должны быть реализованы на языке программирования Java, что сильно привязывает систему к определённому программному стеку. В будущем, может возникнуть необходимость сменить стек на тот, что находится на более близком уровне к железу.
- gRPC: то же самое что и Java RMI, но имеет реализацию для большого количества языков программирования. К тоже же, в качестве транспортного протокола используется современный HTTP/2 с улучшенной скоростью и безопасностью. Именно gRPC мы и выбрали.

Разработка модуля передачи сообщений

Определившись с сетевым протоколом, мы приступили к реализации. Языком программирования был выбран Java, так как опыт работы с ним был у всех членов команды. Была изучена официальная документация библиотеки gRPC, а также примеры работы с ней на сайте medium.com. Сама библиотека

была подключена средствами системы сборки и репозитория пакетов Maven. Сначала нами был написан манифест файл формата Protocol Buffers (далее protobuf), который определяет то, как будут выглядеть пакеты передаваемых данных для классов, которые были описаны в пункте "Проектирование формата сообщения и событий". Данный файл одинаковый для компонентов рендеринга и клиента. Возникли проблемы с сериализацией поля Scene.embreeScene ввиду того, что оно являлось указателем на объект в памяти, а не настоящим Java-объектом. Как описывалось ранее, для проведения некоторых расчётов используется библиотека Intel Embree. Реализации её для Java не существует, поэтому оно подключалась в Java-контекст с помощью Java Native Interface (JNI). Он позволяет подключать и вызвать из Java-кода библиотеки, написанные на C/C++. В таком случае, объекты из подключенной библиотеки существуют как объекты класса Pointer, который лишь даёт указатель в памяти на C-объект. Мы решили эту проблему с помощью дополнительной сериализации этого поля средствами Intel Embree в строку. После этого, весь класс Scene стал сериализовываться в protobuf.

Класс Camera, будучи полностью Java-объектом, сериализовывался в protobuf без проблем.

Сериализацию последнего класса – HdrReplay, выполнить было проще всего, ввиду того, что protobuf превращает всё при передаче в бинарный формат, а HdrReplay содержит в себе только бинарный массив. Исходный код манифеста protobuf доступен в Приложении 2.

Далее, нами были разработаны два класса: SceneClient и RendererServer, для клиентского компонента и компонента рендерера соответсвенно. Эти классы занимались вызовов удалённых процедур между узлами и передавали сериализованные protobuf-данные. Были написаны обработчики ошибок для обоих классов. В класс RendererServer была добавлена многопоточность, чтобы обеспечить ускорение не просто за счёт мультипликации узлов распределённой системы, но и за счёт параллелизма в

рамкой отдельного узла. Исходные коды этих классов доступны в Приложении 3.

Внедрение модуля передачи с модулями сцены и рендеринга

Завершив разработку классов, мы скооперировалась с другими командами, загрузили исходные коды в репозиторий на github.com, и принялись их интегрировать с остальным кодом. Так как gRPC-обработчики должны слушать другие узлы на наличие входящий сообщений, и делать это в отдельном потоке, чтобы не блокировать основной, мы модифицировали точку входа в программу (функцию `main()`), добавив в неё вызов дополнительного потока-слушателя. Сделать это нужно было для обоих модулей (клиентского и рендерера), но принципиально, сами обработчики ничем друг от друга не отличаются, поэтому проблем с внедрением не возникло.

Помимо обработчиков, мы добавили перехват аргументов командной строки при запуске программы, чтобы можно было указывать адреса узлов, подключённых к распределённой системе.

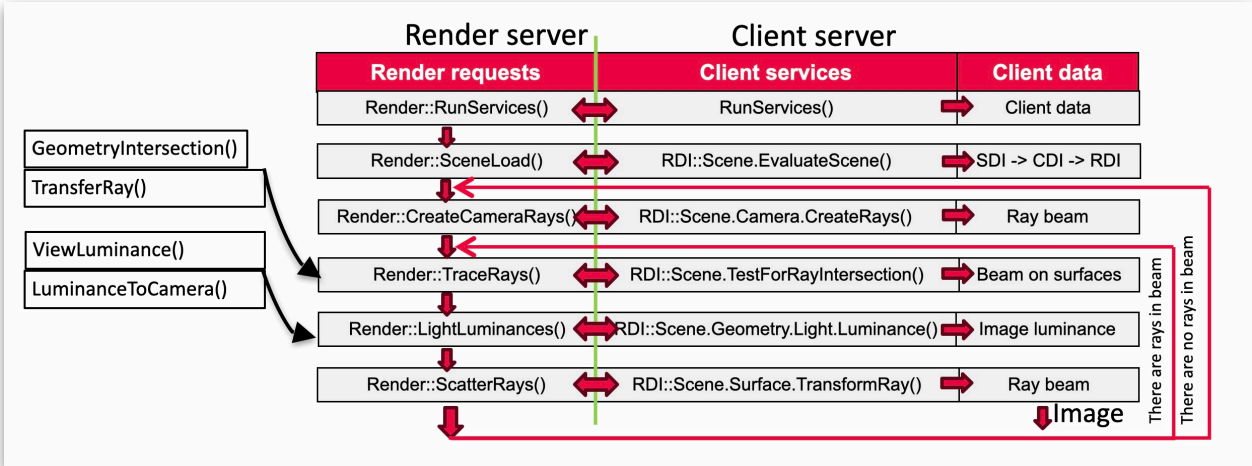

Рис. 2 - Схема удалённых вызовов между компонентами системы

Общее тестирование

Работа полученного программного обеспечения была проверена на нескольких компьютерах, как локально (дома), так и в присутствии преподавателя (в университете):

Таблица 1 - Оборудование для тестирования

	Компьютер 1 (рендерер)	Компьютер 2 (клиент)	Сеть	Время рендеринга
Дома	AMD Ryzen 7 2700X: • 4.3 GHz • 8 ядер • 16 потоков 32ГБ ОЗУ	Intel Core i5 7300HQ: • 3.5 GHz • 4 ядра • 4 потока 12ГБ ОЗУ	Проводной ethernet: • 1Гб/с	22 сек
В университете	Intel Core i5 8250u: • 3.4 GHz • 4 ядра • 8 потоков 32ГБ ОЗУ	Intel Celeron N5095A: • 2.0 GHz • 4 ядра • 4 потока 16ГБ ОЗУ	Wi-Fi 6: • 802.11ax • до 1200Мб/с	1 мин

Результаты вполне ожидаемы и объяснимы: в первом случае (дома), оборудование было мощнее чем в университете, а сеть, хоть по спецификации и медленнее, но использование прямого проводного соединения между двумя компьютерами позволило получить реальную скорость равную теоретической. При использовании Wi-Fi-соединения, добиться максимальной скорости можно только в идеальных условиях. Примеры сгенерированных изображений в Приложении 4.

Первоисточники

1. Таненбаум Э., Стеен М. Распределённые системы. Принципы и парадигмы. - Санкт-Петербург: Питер, 2003. – 877 с. – ISBN 5-272-00053-6.
2. Документация gRPC [Электронный ресурс] – URL: *https://grpc.io/*. – Режим доступа: свободный. – Дата обращения: 24.01.2024.
3. Гамбетта Г . Компьютерная графика. Рейтрейсинг и растеризация. - Санкт-Петербург: Питер, 2022. – 224 с. - ISBN 978-5-4461-1911-0.
4. Sung K., Shiuan J., Ananda A.L. Ray tracing in a distributed environment. Department of Information Systems and Computer Science, National University of Singapore. 1996. DOI 10.1016/0097-8493(95)00091-7.
5. Инструкция по использованию gRPC с Java [Электронный ресурс] – URL: *https://pamodaaw.medium.com/handson-introduction-to-grpc-with-java-1195870027fb*. – Режим доступа: свободный. – Дата обращения: 18.01.2024.
6. Huiming Chen, Huandong Wang, Depeng Jin, Yong Li. Advancements in Federated Learning: Models, Methods, and Privacy. 2023. DOI 10.1117/2302.11466.
7. Ефремов М. А., Холод И. И. Разработка архитектуры универсального фреймворка федеративного обучения – Программные продукты и системы. – 2022. – Т. 35. – №. 2. – С. 263-272. – статья, описывающая архитектуру и обмен данными в федеративной системе машинного обучения.
8. Запечников С. В. Доказательства с нулевым разглашением и их применения при обработке информации в недоверенных средах – Вестник современных цифровых технологий. – 2021. – №. 6. – С. 11-22. – статья, описывающая проблематику федеративных вычислений, в сфере криптографии.

9. Zhdanov D., Zhdanov A., Sokolov V., Wang Yan. Federative rendering model. ITMO University. 2023. DOI 10.1117/12.2686883.
10. Фролов В. А., Волобой А.Г., Ершов С.В., Галактионов В.А. Современное состояние методов расчёта глобальной освещённости в задачах реалистичной компьютерной графики. Московский государственный университет имени М.В. Ломоносова. 2021. DOI 10.15514/ISPRAS–2020–33(2)–1.

ПРИЛОЖЕНИЕ 1

Модель данных

```
class Scene {
    List<Point3f> vertices = new ArrayList<>();
    Map<Point3f, Integer> verticesToIndices = new HashMap<>();
    List<SceneObject> sceneObjects = new ArrayList<>();
    List<LightSource> lightSources = new ArrayList<>();
    Pointer embreeDevice = EmbreeNatives.rtcNewDevice(Pointer.NULL);
    Pointer embreeScene = EmbreeNatives.rtcNewScene(embreeDevice);
    Map<Integer, SceneObject> objectsByGeomId = new HashMap<>();
}

class Camera {
    Point3f pos;
    Vector3f up;
    Vector3f dir;
    float fov; //horizontal, degrees
    int width;
    int height;
}
```

ПРИЛОЖЕНИЕ 2

protobuf манифест

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.example";
option java_outer_classname = "HdrStreamingProto";
option objc_class_prefix = "HLWS";


// The greeting service definition.
service StreamingHdr {
  // Streams a many greetings
  rpc SayHdrStreaming (stream SceneRequest) returns (stream HdrReply) {}
}

// The request message containing camera and scene JSON
message SceneRequest {
  string scene = 1;
  string camera = 2;
}

// The response message containing HDR file
message HdrReply {
  bytes image = 1;
}
```

ПРИЛОЖЕНИЕ 3

Код классов SceneClient и RendererServer

```
public class SceneClient{
  private static final Logger logger = Logger.getLogger(SceneClient.class.getName());

  public static void main(String[] args) throws InterruptedException {
    NameResolverRegistry.getDefaultRegistry().register(new DnsNameResolverProvider());
    final CountDownLatch done = new CountDownLatch(1);

    // Create a channel and a stub
    ManagedChannel channel = ManagedChannelBuilder
        .forAddress("localhost", 50051)
        .usePlaintext()
           .maxInboundMessageSize(Integer.MAX_VALUE)
        .build();
    StreamingHdrGrpc.StreamingHdrStub stub = StreamingHdrGrpc.newStub(channel);
    ClientResponseObserver<SceneRequest, HdrReply> clientResponseObserver =
        new ClientResponseObserver<SceneRequest, HdrReply>() {
          ClientCallStreamObserver<SceneRequest> requestStream;
          @Override
          public void beforeStart(final ClientCallStreamObserver<SceneRequest> requestStream) {
            this.requestStream = requestStream;
            requestStream.disableAutoRequestWithInitial(1);
            requestStream.setOnReadyHandler(new Runnable() {
              @Override
              public void run() {
                while (requestStream.isReady()) {
                      String scene_json = "";
                      String camera_json = "";
                       try {
                              scene_json = SceneClient.getScene();
                              camera_json = SceneClient.getCamera();
                       } catch (Exception e) { e.printStackTrace(); }
              SceneRequest request =
 SceneRequest.newBuilder().setScene(scene_json).setCamera(camera_json).build();
              requestStream.onNext(request);
              requestStream.onCompleted();
            }
          }
         });}

          @Override
          public void onNext(HdrReply value) {
               ObjectReader or = new ObjectMapper().reader();
               try {
                ByteString content_proto = value.getImage();
                byte[] content = content_proto.toByteArray();
                Path content_path = Paths.get("output.hdr");
                Files.write(content_path, content);
               } catch (Exception e) { e.printStackTrace(); }
            requestStream.request(1);
          }
          @Override
```

```
          public void onError(Throwable t) {
            t.printStackTrace();
            done.countDown();
          }
          @Override
          public void onCompleted() {
            logger.info("All Done");
            done.countDown();
          }};

    stub.sayHdrStreaming(clientResponseObserver);
    done.await();
    channel.shutdown();
    channel.awaitTermination(1, TimeUnit.SECONDS);
  }

  public static String getScene() throws Exception {
        NativeLibsManager.prepareNativeLibs();
        System.setProperty("jna.debug_load", "true");
        Scene scene = SceneLoader.loadOBJ("models/casa/casa.obj");
        scene.initEmbree();
        scene.getLightSources().add(new LightSource(
                1e1f, new Color3f(0.1f,0.1f,0.1f), new Point3f(213, 300, 280))
        );
        scene.getLightSources().add(new LightSource(
                1e1f, new Color3f(1,1,1), new Point3f(278, 273, -500)
        ));
        scene.getLightSources().add(new LightSource(
                1e1f, new Color3f(0.1f,0.1f,0.1f), new Point3f(213, 300, 330)
        ));
        ObjectWriter ow = new ObjectMapper().writer();
        return ow.writeValueAsString(scene);
  }

  public static String getCamera() throws Exception{
        Point3f cameraPos = new Point3f(10, 10, 10); // for casa model
        Point3f cameraLookAtPos = new Point3f(0, 0, 0); // for casa model
        Vector3f cameraDirection = new Vector3f();
        cameraDirection.sub(cameraLookAtPos, cameraPos);
        Camera camera = new Camera(
                cameraPos, new Vector3f(0, 1, 0), cameraDirection, 40
        );
        camera.setWidth(1000);
        camera.setHeight(1000);
        ObjectWriter ow = new ObjectMapper().writer();
        return ow.writeValueAsString(camera);
  }
}



public class RendererServer{

  private static final Logger logger =
      Logger.getLogger(RendererServer.class.getName());
```

```
  public static void main(String[] args) throws InterruptedException, IOException {
            StreamingHdrGrpc.StreamingHdrImplBase svc = new
StreamingHdrGrpc.StreamingHdrImplBase() {
              @Override
              public StreamObserver<SceneRequest> sayHdrStreaming(final StreamObserver<HdrReply>
responseObserver) {
               final ServerCallStreamObserver<HdrReply> serverCallStreamObserver =
(ServerCallStreamObserver<HdrReply>) responseObserver;
                serverCallStreamObserver.disableAutoRequest();

                class OnReadyHandler implements Runnable {
                  private boolean wasReady = false;

                  @Override
                  public void run() {
                    if (serverCallStreamObserver.isReady() && !wasReady) {
                      wasReady = true;
                      logger.info("READY");
                      serverCallStreamObserver.request(1);
                    }
                  }
                }
                final OnReadyHandler onReadyHandler = new OnReadyHandler();
                serverCallStreamObserver.setOnReadyHandler(onReadyHandler);

                // Give gRPC a StreamObserver that can observe and process incoming requests.
                return new StreamObserver<SceneRequest>() {
                  @Override
                  public void onNext(SceneRequest request) {
                    // Process the request and send a response or an error.
                    try {
                      // Accept and enqueue the request.
                      String scene_json = request.getScene();
                      String camera_json = request.getCamera();
                      MessageDigest md = MessageDigest.getInstance("MD5");
                    byte[] bytesOfMessage = scene_json.getBytes("UTF-8");
                      String message = "Scene MD5 sum is " + new String(md.digest(bytesOfMessage),
"UTF-8");
                      logger.info("⟵ " + message);
                      bytesOfMessage = camera_json.getBytes("UTF-8");
                      message = "Camera MD5 sum is " + new String(md.digest(bytesOfMessage),
"UTF-8");
                      logger.info("⟵ " + message);
                      ObjectWriter ow = new ObjectMapper().writer();
                      ObjectReader or = new ObjectMapper().reader();
                      Scene scene = or.readValue(scene_json, Scene.class);
                      Camera camera = or.readValue(camera_json, Camera.class);

                      // Криво, но уже не стал разбираться как в string форматнуть HDRImageRGB
                      // На первый взгляд никак
                      HDRImageRGB image = Renderer.renderParallelN(scene, camera, 6, 200);
                      HDREncoder.writeHDR(image, new File("output.hdr"));
                      Path content_path = Paths.get("output.hdr");
                      byte[] output_content = Files.readAllBytes(content_path);
                      HdrReply reply =
HdrReply.newBuilder().setImage(ByteString.copyFrom(output_content)).build();
                      responseObserver.onNext(reply);
                      // Check the provided ServerCallStreamObserver to see if it is still ready
to accept more messages.
                      if (serverCallStreamObserver.isReady()) {
```

```
                    serverCallStreamObserver.request(1);
                  } else {
                    // If not, note that back-pressure has begun.
                    onReadyHandler.wasReady = false;
                  }
                } catch (Throwable throwable) {
                  throwable.printStackTrace();
                  responseObserver.onError(
                      Status.UNKNOWN.withDescription("Error handling
request").withCause(throwable).asException());
                }
              }

              @Override
              public void onError(Throwable t) {
                // End the response stream if the client presents an error.
                t.printStackTrace();
                responseObserver.onCompleted();
              }

              @Override
              public void onCompleted() {
                // Signal the end of work when the client ends the request stream.
                logger.info("COMPLETED");
                responseObserver.onCompleted();
              }
            };
          }
        };

        final Server server = ServerBuilder
            .forPort(50051).addService(svc).build().start();
        logger.info("Listening on " + server.getPort());
        Runtime.getRuntime().addShutdownHook(new Thread() {
          @Override
      public void run() {
            // Use stderr here since the logger may have been reset by its JVM shutdown hook.
            System.err.println("Shutting down");
            try { server.shutdown().awaitTermination(30, TimeUnit.SECONDS); }
            catch (InterruptedException e) { e.printStackTrace(System.err); }
          }
        });
        server.awaitTermination();
      }}
```

ПРИЛОЖЕНИЕ 4

Результаты рендеринга

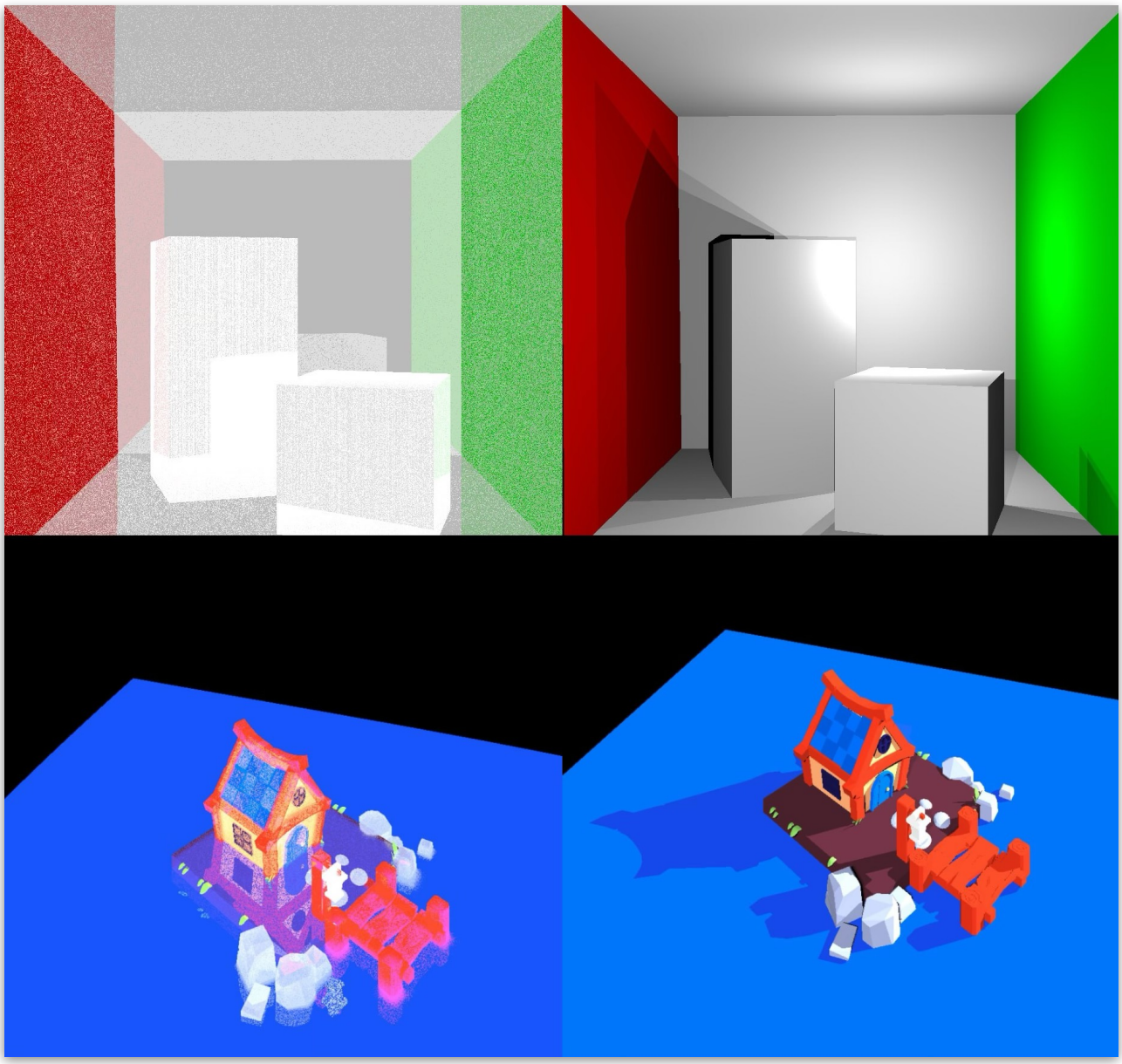

Рис. 3 - Слева без расчёта глобального освещения / справа с глобальным освещением

ОФОРМЛЕНИЕ ОТЧЁТНЫХ ДОКУМЕНТОВ

Мы начали выполнение практики с заполнения соответствующего раздела в Информационной системе Университета (далее ИСУ) ИТМО. Задание выглядело следующим образом:

Таблица 2 - Задание на практику

№ этапа	Наименование этапа	Продолжительность этапа	Задание этапа
1	Инструктаж обучающегося	1	Инструктаж обучающегося по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, а также правилами внутреннего трудового распорядка.
2	Проектирование формата сообщений и событий	20	Исследование содержание, объём и тип сообщений, которые будут использованы для обмена данными между узлами системы рендеринга, а также события, которые будут гененрировать и обрабатывать сообщения.
3	Выбор сетевых протоколов	20	Спрогнозировать нагрузку на систему. Выбрать наиболее подходящий прикладной и транспортный сетевые протоколы, исходя их объёма сообщения и прогнозируемой нагрузки на систему.

4	Разработка модуля передачи сообщений	30	Разработать модуль передачи сообщений, протестировать его с помощью искусственно сгенерированных данных под тестовой нагрузкой.
5	Внедрение модуля передачи в модули сцены и рендеринга	30	Программно связать модуль передачи с модулями сцены и рендеринга, отладить, провести тестирование с упором на сетевой модуль.
6	Общее тестирование	10	Провести общее тестирование разработанной системы с учётом всех компонентов (сцена + сетевой модуль + модуль рендеринга).
7	Оформление отчетных документов в соответствии с требованиями	10	Подробно описать процесс выполнения предыдущих этапов в отчёте, оформленном согласно методическому пособию (*https://books.ifmo.ru/* file/pdf/2622.pdf) с соблюдением требуемой структуры.

После разработки системы федеративного рендеринга и демонстрации её работы научному руководителю, команды приступили к заполнения отчётных документов: каждая команда описывала только ту часть созданной системы, за которую отвечала. Нами были агрегированы полезные ссылки на страницы в интернете и книги, на которые мы использовали.

Опираясь на правила оформления из методического пособия "ПРОИЗВОДСТВЕННАЯ ПРАКТИКА МАГИСТРАНТОВ: ОРГАНИЗАЦИЯ И ПРОВЕДЕНИЕ", авторы Т.А. Маркина, А.В. Пенской, Д.Г. Штенников,

Е.Ю. Авксентьева, А.Г. Ильина, был создан docx документ с описание процесса проведения научно-исследовательской работы. Полученный документ был перепроверен и загружен в систему ИСУ ИТМО.

ЗАКЛЮЧЕНИЕ

В процессе выполнения этого этапа научной работы мною были получены практические навыки разработки модуля взаимодействия узлов системы федеративного рендеринга: был разработан программный прикладной интерфейс, модель данных и выбран наиболее подходящий для задачи сетевой протокол взаимодействия.

Нашими командами была создан прототип системы федеративного рендеринга, которая поддерживает работу с разными пользовательскими моделями (сценами), несколькими типами расчёта освещения, и распределённой моделью взаимодействия.

В дальнейшем, исходный код в модифицированном и дополненном виде будет использован для реализации проекта в рамках выпускной квалификационной работы.