**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: Реализация на RayTracer

Дипломант: Научен ръководител:

Николай Дионисов Светлин Младенов

СОФИЯ

2015

# Увод

Рендерирането е фундаментален компонент от компютърната графика. Рендериране се нарича процеса по превръщане на описание на три дименсиална сцена в изображение. Алгоритмите за анимация, моделиране, текстуриране и други части от компютърната графика трябва да преминат през някакъв вид рендериращ процес за да могат да бъдат превърнати в видими изображения. Техниките за рендериране са разделени в две основни групи – растеризиране и тресиране на лъчи.

Растеризирането е метод използван в интерактивни приложения, който обхожда всички видими примитиви във сцената и геометрично ги прожектира върху платно. Растеризирането само по себе си не дава много добри резултати откъм фотореализъм, но е значително по-бързо от всички останали методи. Поради тази причина това е най-разпространенят метод в гейм индустрията. Видео картите имплементират работят по този начин.

Методите за синтез на изображение базирани на Тресирането на лъчи или „Ray tracing“ са вдъхновени от реалния физичен свят. Те са симулация на света около нас. Работата на рей трейсъра се изразява в проследяването на лъчи минаващи през виртуален филм. При прекъсването на лъча от обект във сцената, част от лъча рефлектира, а друга се абсорбира. Типично за разработката на рей трейсъри се използват снимки и компютърно пресъздадени сцени от снимките. След това резултатите от програмата се сравняват с оригиналните снимки. Основният проблем на този метод е дългото време на генериране на изображение. Рей трейсинга е предпочитаният метод за рендериране, там където има възможност по-голяма част от работата да се свърши предварително.

В тази дипломна работа е представена примерна имплементация на RayTracer, тип PathTracer.

# ПЪРВА ГЛАВА: МЕТОДИ И ТЕХНОЛОГИИ ЗА РЕАЛИЗИРАНЕ НА RAY TRACER

## Основни принципи

За синтезиране на изображения ни трябва някакъв модел на светлината. Предбладаващите модели на светлината са геометричните и физичните. Геометричната оптика моделира разпространението на светлина като лъчи, които се движат в прави линии и техните пътища са представени като серия от рефлекции и рефракции. Физиката гледа на разпространието на светлината като вълнов феномен, който е повлиян от ефекти на поляризация, смущения и дифракциии.

### Основни типове рей-трейсъри

Класическият рей трейсър, използващ геометричния модел, взима проби от идващата светлина от сцената, като пуска лъчи през съответните точки на филма и ги проследява до източниците на светлина. Изображенията генерирани от имплементацията на класическият рей трейсър може да изглежда добре, но също и нереалистично. Причината за това е, че този алгоритъм пренебрегва голяма част от домейна на интегралното уравнение, което описва сложната интеракция на светлината с обектите.

Разпределеният рей трейсър, усъвършенства класическият рей трейсър като използва Монте Карло техники. Вместо да използва резултата от само един вторичен лъч, той разпределя няколко вторични лъча за да може по-точно да пресметне нужните интеграли. Този подход е много по-скъп от страна на процесорно време, но е нужен за ефекти като меки сенки, depth of field, motion blur и други. Въпреки подобренията, които разпределеният рей трейсър внася, те не решават напълно проблема за фотореалистична графика. [1]

За да имаме реалистични изображения, трябва да взимаме в предвид и светлината отразена от останалите обекти в сцената. Този ефект в компютърната графика се нарича глобална илюминация и се противопоставя на локалната илюминация, чиито представители са представените до тук типове рей трейсъри.

Пътищния рей трейсър е разработен като решение на рендериращото уравнение и е силно базирано на Монте Карло техники. Той разглежда целия домейн на уравнението, докато разпределеният рей трейсър – само част от него. Този метод пуска не един, а много първични лъчи прези всяка една точка на филма, след което пробаблистично решава в каква посока да го отрази. Рекурсията в този метод е ненужна. Основния проблем на Пътищния рей трейсър е, че разликите в крайния резултата се възприемат като шум. При ползването на повече лъчи през всеки пиксел, покриваме по голяма част от домейна, изичисляваме интеглала с по-малка грешка и шума нямалява. Това обаче струва много прецесорно време – трябва да увеличим пробите 4 пъти, за да намалим шума двойно.[2] Възможно е да се използва техника наречена „Photon mapping“ за намаляването на шума, но тя не се разглежда тук.

### RayTracer tasks

За да извършат задачите си, рей-трейсърите трябва да могат да си отговорят на няколко фундаментално важни за тях въпроса относно:

1. Камери – Как и от къде се разглежда сцената? Камерите генерират първичните лъчи от изгледната точка към сцената.
2. Лъч-обект сечение – Пре пускане на лъч, кой обект и в коя точка ще бъде пресечен от него? Също така трябва да знаем на къде гледа повърхността от която е взета въпросната точка и какъв е материала на обекта.
3. Разпределение на светлината – Трябва да се моделира разпространението на светлината в сцената. Това включва позиците на източниците на светлина и по какъв начин се разпространява тяхната енергия в пространството.
4. Видимост – За да разберем дали даден източник на светлина достига дадена точка на повърхност, трябва да знаем дали между тях има непрекъснат път.
5. Материали – Всеки обект трябва да предоставя описание на своя външен вид като информация за това как светлината реагира с повърхността на обекна.
6. Рекурсивност – Понеже един лъч светлина може да е бил рефлектиран няколко пъти преди да достигне дадена точка. Понякога е нужда да бъдат преследени повече от един вторичен лъч за да се прихванат някои ефекти. Рефлектираните лъчи ще бъдат наричани вторични, а тези идващи директо от камерата - първични.
7. Разпространине на лъчите – Кога един светлинен лъч пътува в вакуум през пространството, неговата енергия остава константа на всяка една точка от него. Изображенията които типично бихме искали да занемем не са във вакуум и ни трябват модели по който да отнемаме от енергията за до създадем ефекти като мъгла, пушилка или просто атмосферата на земята. [2]

### The rendering equation

Цялата теорията върху която е изграден рей трейсингът е бязирана на т.н. “rendering equation”. Рендериращото уравнение, формулирано от Джеймс Каджия през 1986, описва движението на светлината от една точка върху повърхност до друга като сума на излъчен и рефлектиран радианс.

Където:

х е гочката на повърхността

посоката бягаща от повърхността

е излъчваният радианс

е рефлектираният радианс

Радиансът ни казва колко светлинна енергия пристига или напуска дадена повърхност през дадена единица време. В вакуум радианса е константа за всяка точка по даден лъч. Това прави рей трейсинга възможен.

Рефлектираната светлина може да се опише по следния начин:

Този интеграл взима в предвид цялата пристигаща светлина и изчислява рефлектинарата. Той взима в предвид и светлината отговаряща за глабална илюминация. Елементите в това уравнение са двупосочната рефлектираща функция, идващата свелина от дадена посока и косинусът от ъгъла между посоката от ноято идва текущия лъч и нормалата на повърхността. Двупосочната рефлектираща функция или „BRDF“ е функцията която дефинира материала на повърхността. Тя приема посоките на изходния и входния лъч и връща даден спектър – цветовете които ще бъдат отразени. Функцията може да бъде представена като 4 дименсиална функция от изходните и входните посоки.

[IMG]

За да може двупосочната рефлектираща функция да е физически възможна, тя трябва да спазва закона за запазвана на енергията и да се подчинява на резипрочния принцип на Хелмхолц[?]. Това значи че функцията винаги трябва да има резултат между 0 и 1. За да сме по-точни, диферециалната функция интегрирана върху хемисфера трябва да бъде по малка или равна на едно. Това означава, че не можем да рефлектираме повече светлина от колкото сме получили. Това може да бъде представено математически както следва:

[IMG]

Реципрочния принцип на Хелмхолц означава, че пробите от идващите и рефлектираните посоки от двупосочната рефлектираща функция могат да се обърнат и резултата ще остане същият.

[IMG]

Намирането на BRDF може да стане по няколко начина. Първият е чрез емпирически измервания и вместване на резултатите в математически функции. Премери за този метод са модели като Ламберт, Фонг и Блин-Фонг. Функцията за дифузен материал(ламберт) е просто константа.

Функцията Li описва пристигащия радианс в дадена точка. Тя може да бъде дълбоко рекурсивено интегрално уравнение, защото част от светлината може да идва индиректно - рефлектирана от други повърхностти. [4]

## Технологии за разработване на рей-трейсъри

Технически най-предизвикателната част от рей-трейсъра е сечението на лъчи с обекти. Това се дължи на факта, че то изисква сложни структури за съхранение в пространтвото, рекурсивно обхождане и много процесорно време. Представени са няколко готови решения подпомагащи

### Embree

Това е колекция от високо производителни рей трейсинг кърнели разработени от Интел. Кърнелите са оптимизирани за фотореалистично рендериране на последните Интелски процесори с поддръжка на SSE, AVX и AVX2. Ембрее съдържа алгоритми за некохерентни и кохерентни лъчи. Поддържат се и динамични сцени благодарение на дву-стъпалните индексни конструиращи алгоритми. [5]

### OptiX

Фреймуорк за създаване на рей трейсиг приложения вървящи върху видео картите на NVidia. OptiX не отраничава продуктите базирани на него само до рендериращи такива. Той позволява на разработчиците да свършат бързо всяка задача, която изиства рей трейсинг. [6]

### OpenCL

Стандарт, дефиниращ програмен интерфейс за паралелни изчисления. Имплементиран е от всички водещи производители на хардуерни изчислителни компоненти. Код написан за OpenCL може да бъде изпълнен от прецесори, видео карти и други. Това е изключително полезно за рей трейсърите като едни от най-гладните за изчислителна мощност приложения. [7]

### OpenMP

Рей трейсърите трябва да използват всички налични ресурси. Въпреки че OpenCL е изключителено мощен инструмент, той изисква начална инвестиция от имплементатора – да се запознае с стандарта, да създаде изчислителните кърнели и т.н. От друга страна е OpenMP, която е библиотека вградена във всички модерни C++ компилатори и платформи. Тя предоставя лесно използваеми примитиви за паралелизиране на кода върху всички процесорни ядра. [8]

## Съществуващи решения и реализации

### PBRT

PBRT е примерна имплементация на рей трейсър идваща с книгата „Physically based rendering“. Въпреки учебният си характер, това е напълно функционален и използван рей трейсър. На него са базирани много други продукти сред които са LuxRender и Mitsuba. Рей трейсърът е силно обектно ориентиран, използва всички налични ядра, поддържа volume rendering и path tracing.

### http://hof.povray.org/images/TopMod_StarBall.jpgPOV-Ray

Pov-Ray е един от първите рей трейсъри, развиващ се и до днес. Първоначалната имплементация е вървяла на Амига компютри и е била базирана на DKBTrace. В момента поддържа photon mapping, специфичен език за описване на сцените, няколко вида светлини, атмосферни ефекти, рефлекции, рефракции и много други. Примерното изображение от Janet Lowry, рендерирано чрез POV-Ray.

### YafaRay

# ВТОРА ГЛАВА: ПРОЕКТИРАНЕ НА СТРУКТУРАТА НА RAY TRACER

## Функционални изисквания към рей трейсъра

Тази дипломна работа цели да реалилиза рей трейсър със поддръжка на минимален сет от функции за да могат да се синтезират изображения от прости тестови сцени. Продукта трябва да е мултиплатформен и да използва всички налични ядра. Трябва да се поддържа камера от стенопеичен тип триъгълни мрежи и проста материална система с възможностите на модела на Фонг. Описанията на сцените да се зареждат от файлове с структура близка до JSON.

### Камера от стенопеичен тип

За да бъде синтезирано изображение, трябва да бъдат избрани първичните лъчи. Те трябва да имат начало и посока. Метода за генерация на тези лъчи се определя от камерата. Стенопеичната камера се използва за генериране на перспективни изображения, тоест – обектите в далечината изглеждат по малки от обектите намиращи се на близки разстояния. За да се постичне този ефект, този вид камера генерира всички лъчи с една и съща точка за начало и а посоките им са еднакво разпределени в дадена посока под формана на четириъгълна пирамида. Ако разгледаме един хоризоначен ред от лъчите и го погледнем отгоре, посоките на лъчите ще са равномерни разпредели между 2-та лъча с най-голям ъгъл между тях. Този ъгъл се нарича field of view и играе важна роля за правилното синтезиране на едно изображение. Field-of-view-то трябва да може да се определя от юсъра на приложението. Вертикалното разпределение на лъчите следва същите правила, с разликата че най-големият ъгъл е дефиниран от подаденото ‘aspect ratio’. То дава съотношение между въпросния ъгъл и field of view-то. Aspect ratio-то е също и съотношението на височината и широчината на крайното изображение. То не се подава, а се изчислява от програмата. Напрактика лъчите се генерират от подадени проби от крайния филм. В рей-трейсърите същесвува модул, който се нарича Sampler. Работата на този модул е да взима проби от различни сигнали, така че да прихване с максимална точност характеристиките на входния сигнал.

### Триъгълна мрежа

Стандартният начин за съхранение и представяне на 3Д модели е чрез тръгъгълна мрежа или по-точно списък с триъгълни полигони. Един рей трейсър трябва да може да зарежда от диска най-популарните формати съдържащи 3д модели. Тези модели трябва да могат да се използват в сцените както всяка друга поддържана форма.

### Материя на обектите

Обектите участващи в сцената трябва да съдържат в себе си информация за това как участват в рендериращото уравнение. Тази информация трябва да може да се контролира от потребителя чрез материята на обекта. В този тей трейсър се използва прост модел, в който материалите са дефинирани като смесица между 4 свойства – излъчване, огледална рефлекция, лъскавост и дуфизия.

### Съхранение на сцените

Сцените трябва да се съхраняват в лесен за редакция от човек файлов формат. За това сцената ще се съхранява в диалект на JSON. В този файл, освен информация за текущата сцена, трябва да се съдържа и описание на това как да се заснеме тази сцена. Това включва от какви позиции, с какви field-of-view-та и метод за заснемане/в дипломната е имплементиран само 1 метод/.

## Описание на главните алгоритми

Работата на Рей трейсъра се изразява в две основни стъпки. Те са зареждане на сцената и синтезирането на изображение от нея. С цел по добро разделения на задачите, цялата дипломна е разделена на 2 условни части около тези 2 задачи. Първият модул е този който се занимава с външния свят и платформата. Той дефинира ентри поинта на програмата, зарежда нужните ресурси в паметта и представя както крайния завърпен резултат на програмата, така и процеса по който се стига до него. Вторият модул е сърцевината на рей трейсъра. Той се занимава с генерирането на ефективни вътрешни структури и синтезиране на изображението. Докато първия модул се старае да бъде крос-платформен и е възможно да изисква допълнителна работа за да тръгне на други платформи, то ядрото винаги е напълно изолирано и портативно. Възможо е изполването на ядрото в външни програми под формата на библиотека.

### Зареждане на сцената и изгледите към нея

Първия вход от потребителя се подава на програмата като аргументи от командния ред. Те включват името на 1 файл описващ сцената и нужните кадри, и поредица от директории за търсене. Алогритъма за зареждане е както следва – зареждане на посочения файл в паметта, регистриране на директориите за търсене в абстакцията за зареждане на файлове, обработка на заредения файл и конструиране на сцената от данните в него.

Регистрирането на директориите за търсене е важна стъпка за да може да се зареди една сцена както трябва. Въпреки че е възможно цялата информация за сцената да се опише в основния файл, това рядко е полезно на практика. Моделите попринцип се експортират от програми за моделиране, а те експортират във външен за рей трейсъра формат. Когато трябва да се зареди външен модел, в конфигурационния файл се съдържа името на модела, а рей трейтрейсъра започва търсене за него в посочените директории. На практика потребителя може да предпочете да държи различните файлове в различни директории, и рей трейсъра трябва да може да ги намери.

Обработката на конфигурационния файл се изразява в разбиването му на лексеми, парсването им и извличането на информация. При грешка, всеки един от тези 3 процеса трябва да бъде изведено съобщение оказващо точно къде е проблема.

Конструирането на сцената се изразява в използването на извечената от последната стъпка информация за да се постои списък от обекти в паметта и зареждането на външни формати. Типичното за данните с които се работи в тази стъпка е че е важно рабположението им в пространстово. За списъка с обекти и за моделите трябва да бъдат построени външни индексиращи структири позволяващи бързото намиране на потенциални колизи между елементи и лъчи.

### Алгоритъм за синтезиране на изображенията

За да бъдат имплеменитани представените в т. 2.1 фичъри, ние се нуждаем от модули предоставящи ни решения на фундаменталните проблеми на рей трейсъра, както са дефинирани в т 1.1.2. Специфичните имплементации на тези модули за сега не ни интересуват. Това което е важно е защо ги има тези модули и как ще се впишат в една по-голяма система. Сега те ще бъдат разгледани в последователността в който извършват работата си.

#### Renderer

Това е компонентът който събира всички останали модули, и осъществява комуникацията между различните модули без те да знаят един за друг. Той контролира потока на изпълнение на програманата на най-високо ниво. Това значи,че се грижи и за паралелизирането на рей трейсъра. Разделянето на задачите се прави по детериминиран начин – изображението се разделя на малки парчета, след което тези парчета се разделят на броя на логическите ядра в процесора. По този начин работата е разделена предварително преди рендерирането да започне и никъде в програмата не присъстват критични секции или мутекси. Това е възможно защото за извършването на този процес са нужни няколко групи от данни – сцената, филмът и камерата. Сцената и камерата са константни през цялото време, а филмът където се съхравява крайния резултата е предварително разделен на части, така че никъде да не се застъпват различните нишки.

#### Sampler

Самплиране се нарича вземането на проби от даден аналогов сигнал с цел улавянето на колкото се може по-голяма част от информацията в него. Проблемът за който се грижи самплера е свързан с факта, че пикселът заема някакво пространство и цветът на цялото това пространтво е представето само от един единствен цвят. По този начин може да се загуби важна информация от входния сигнал.



Ако се взима цвета получен чрез лъч генериран през средата на пиксела, можем грешно за заключим какви трябва да са крайните цветове. Правилният метод за изчисление на крайния цвят е като се взимат повече развомерно разпределени през целия пиксел проби и намирането на средния резултат. 

Работата на самплера е да прецени колко най-малко проби са ни нужни и къде. Производителноста на рей-трейсъра и качеството на крайния резултат е силно зависима от този модул. Имплементираният самплер в тази дипломна работа минава през целия домейн и генерира нови точки през дадени интервал с цел лесния контрол над генерираните проби за всеки пиксел.

#### Камерата като функция

Камерата трябва да превърне пробите взети от предишната стъпка в лъчи. Камерата може да се разглежда просто като функция приемаща позиции някъде на входното платно и връщаща лъчи с начало и посока. Важно е да се спомене, че пробите генерирани в предишната стъпка са в растерна координатна система. Това значи, че коориднатите са стойности между (0,0) за горел ляв ъгъл и (резолюция Х, резолюция У) за долен десен. Тъй като камерата трябва да знае за големината на платното, за да знае релативното разположение на едни лъчи спрямо други, тази координатна система е лоша за нея. Това с което работи камерата е нормализирана координатна система на устройството(NDC). При ней, горния ляв ъгъл е отново с коориднати (0,0), но долния десен е с (1, 1). Рендерърът, който отговаря за потока на данни е длъжен да ги преобразува.

#### Scene

За следващата стъпка, освен лъча ни е нужна и друга информация – къде този лъч се прекъсва от сцената. Сцената е компонента, който съдържа всички обекти – позииците и ориентациите им в една единна координатна система, и материалите им. При подаване на лъч, тя трябва да може бързо да намира потенциално пресичащи се с него обекти. Втората отговорност на сцената е, да дърви всички светещи обекти в отделна структура, за да може да се идентифицират бързо чрез номер. Това е нужно за ефикасното взимане на проби от светлинните обекти, нещо което интегратора би искал да може да прави. Последното задължение на сцената е, при подаване на номер на обект, да може да върне свойствана му на рефлектиране на светлината – материала. Сцената е просто компонента, който описва целия виртуален свят и предоставя методи за изискване на части от информацията съхранена в него.

#### Integrator

Интеграторът е най-вайната част от рей трейсъра, дефинираща типа му. Той получава като вход дадени лъчи и техните точки на пресичане с сцената. С тази информация, той трябва да изчисли радианса, който идва идва посока обратна на посоката на лъча. Името му идва от факта, че неговата основна функция е да пресмята рендериращото уравнение. За да извърши работата си, интегратора може да генерира вторични лъчи и да се извика рекурсивно върху тях.

В тази дипломна е представен интегратор за пътищен рей трейсър. За всеки лъч, той итеративно симулира няколко подскока, като генерира всеки следващ вторичен лъч на напълно случаен принцип. Интегратора получава като вход също и сцената, за да може да намира точките на пресичане от вторичните лъчи.

#### Филм

Крайната спирка на потока от данни е филмът. Ролята на филма попринцип се играе от 2д решетка от стойности или просто 2д текстура. В допълнение филмът освен да съхранявя крайния резултат, той трябва и да филтрира входните данни. А те са пробата генерирана от самплера и радианса върнат от интегратора. Филтърът трябва да вземе в предвид всички проби попадащи върху даден пиксел в изчисленията си за крайния му резултат. В случая просто взимаме средното аритметично на влиаещите радианси.

#### Представяне на крайния резултат

Целия процес на синтезиране на изображения ще бъде видим на прозорец създаден от програмата. През някакви фиксирани интервали от време, вътрешния буфер ще бъде компиран върху текстурата на прозореца.

## Избор на език и среда за програмиране

За реализацията на дипломната работа ще се използва езикът C++, защото той е статичен език, компилиращ се до машинен код. По този начин не трябва плащаме за поддръжка на виртуална машина или какъвто и да е друг овърхед. Това ще доведе до изполване пълния потенциал на процесора за изчислението на гладните за процесорно време алгоритми. C++ е език с богата история. Има много качетвени компилатори следващи стандарта му. Почти всички са безплатни или дори с отворен код. За компилацията на програмата ще се изисква модерен компилатор с поддържа на C++ 11.

Като среда за разработка ще се изполва Microsoft Visual Studio 12 Professional. Тази версия на интегрираната среда за разработка е достъпна за учебни цели напълно безплатно. Малко по-късно след началото на тази дипломна работа от Майрософт пуснаха нова версия на своя продукт – Community Edition, който предоставя същите възможности като Professional, но напълно безплатно както за некомерсиална така и за комерсиална употреба. Основните причини да бъде избрана тази среда са вградените профилер и дебъгер.

# ТРЕТА ГЛАВА: ПРОГРАМНА РЕАЛИЗАЦИЯ НА RAY TRACER

## Програмата разледана отгоре, инитиализация и кода който държи всичко заедно

Програмата е разделена условно на модул за вход-изход и ядро. При стратиране, контрола се поема от вход-изходния модул. Той парсва параметрите подадени от командния ред за да разбере как се казва файла с описание на сцената и всички директории за търсене.

const char\* scene\_fn = "/default.scene";

if (argc > 1) {

scene\_fn = argv[1];

}

rt::sdl::FileLoader loader;

if (argc <= 1) {

loader.add\_directory("./scenes");

}

else{

for (int i = 2; i < argc; ++i) {

loader.add\_directory(argv[i]);

}

}

След това се прочита и обработва сценовия файл. Той трябва да се парсне и да се извлече информацията.

auto&& file = rt::sdl::load\_config\_file(scene\_fn, loader);

if (!file) {

return 0;

}

rt::sdl::ResourceManager manager(loader);

if (!rt::sdl::load\_scene\_and\_accelerate(\*file.get(), scene, manager)) {

std::cout << "Failed to load scene " << scene\_fn << std::endl;

return 0;

}

std::vector<rt::core::Renderer> renderers;

std::vector<rt::core::Film\*> films;

rt::core::MemoryArena arena(RENDER\_CONTEXT\_MEMORY\_ARENA\_SIZE);

if (!rt::sdl::load\_images(\*file.get(), renderers, scene, films, arena)) {

return 0;

}

file.reset();

Функциите които се грижат за зареждането на входните данни се грижат и за принтирането на грешки на стандартния изход. Това значи че при индикиране на грешка, изпълнението моге да се прекрати спокойно. Върнатия от първата функция файл е от тип unique\_ptr, което значи че указателя в него ще бъде освободен в всеки случай. С заредения файл, ние импортваме сцената в формат удобен на рей трейсъра и генерираме структури за търсене. След това извличаме списъка с изображения за рендериране и генерираме Рендъръри. Тъй като не знаем размера на тези рендеръри, използваме арена алокатор. Причината това да работи е, че много от структурите изполвани в програмата съдържат само прости данни. Накрая се ресетва умния указател, за да се освободи буфера на файла.

std::thread render\_thread(rendering\_thread, &renderers);

След като всички нужни данни са заредени, започваме рендерирането на изображенията на отделна нишка. По този начин, основния тред ни е свободен за продължително представяне на процеса на рендериране и крайния резултат. Новия тред ще премине през всички подадени рендеръра и ще им извика метода за рендериране. То от своя страна ще стартират достатъчно нови нишки за запълване на логическите ядра.

std::vector<rt::core::Surface2d\*> surfaces;

for (auto film : films) {

surfaces.push\_back(film->get\_surface());

}

rt::sdl::present\_rendering(surfaces);

Основната нишка продължава, като извлича референции към нужните данни за представяне и предава контрола на present\_rendering, който чрез SDL отваря прозорец и представя процеса на рендериране. Тази функция си завърта цикъл, от който се излиза при затваряне на прозореца.

for (auto& renderer : renderers) {

renderer.do\_not\_render();

}

render\_thread.join();

for (auto film : films) {

delete film;

}

manager.cleanup();

Последната стъпка е изчистване на ресурсите и терминиране на рендериращите нишки. Стандартната библиотека не предоставя метод за терминиране на нишки, а дори и да предоставаше нямаше да е добра идея да се използва. Ако бъде терминирана създадената рендерираща нишка, нишките създадени от нея няма да бъдат освободени. За това подхода който се използва е, маркиране на всички задачи, че не трябва да се рисуват и изчакване на рендериращата нишка да ги обиколи. Когато това се случи, тя ще освободи всичко което трябва и ще приключи работата си. Терминирането на рендериращите нишки е изключително важно, защото те пишат въхру ресурс менажиран от основната нишка. Ако този ресурс бъде унищежен, те ще започнат да пишат върху боклук, което е недефинирано държание.

## Вход на програмата - Компилиране и обработка на конфигурационния файл

Методите за извличане на информация от външен източник към структури на ядрото са точно 3 и са представени в предишната точка. Те могат да бъдат променени да извличат информация от всякакви формати. В случая се изполва текстови файл с структура подобна на JSON. Под тези 3 метода съществува друга абстракция за прочитане на данните от няколко прости унифицирани функции. Първата абстракция позволява напълната промяна на входния формат, а втората – промяна на диалекта на JSON. Всичко което съдържа хешове и листове може да се адаптира.

### Зареждане на файла

std::unique\_ptr<ConfigFile, ConfigFileDeleter> load\_config\_file(const char\* fn, FileLoader& loader) {

std::string str;

bool success = loader.load\_string\_file(fn, &str);

if (!success) {

std::cout << "Failed to open file " << fn << std::endl;

return 0;

}

json\_file\* file = sjson\_compile\_source(str.c\_str());

if (!file) {

std::cout << "Failed to compile " << fn << std::endl;

}

return std::unique\_ptr<ConfigFile, ConfigFileDeleter>((ConfigFile\*)file);

}

void free\_config\_file(ConfigFile\* f) {

sjson\_free\_file((json\_file\*)f);

}

Първото което се случва е зареждането на файла в паметта. Тази задача се пада на loader-а. Неговата работа е да качва цели файлове в паметта. По специалното за него е, че той търси съответните файлове последователно в всички директории за търсене. Всички файлове които се очаква да бъдат в тези директории трябва да бъдат зареждани по този начин. Мениджъра на ресурси винаги си държи валидна референция към лоадера.

### Компилиране на файла

Следващата стъпка е компилирането на съдържанието на файла. Резултата от компилацията е json-file който е абстрактен тип. Попринцип съдържа готови хешове и листове попълнени от подадения файл. Той се връща като умен пойнтър за да е сигурно освобождаването му.

Текущата имплементация на компилатора за SJSON(така наричат този диалект) се състои от максимално опростен лексер и минимален рекурсивен топ-даун парсер. Основната идея на този диалект е да направи синтаксиса по изчистен и приятен за потребителя. Целият интерфейс към резултата от компилатора е следният:

//get the root object of a certain file

JS\_Object sjson\_object\_root(json\_file\* file);

//get the count of an object's children

int sjson\_object\_count(const JS\_Object obj);

JS\_Object sjson\_object\_child(const JS\_Object obj, int index);

JS\_Object sjson\_object\_child(const JS\_Object obj, const char\* key);

//These will do any possible conversions, and exit if failed to do so

const char\* sjson\_object\_string(const JS\_Object obj);

int sjson\_object\_int(const JS\_Object obj);

float sjson\_object\_float(const JS\_Object obj);

int sjson\_object\_line(const JS\_Object obj);

JS\_Object е прост обект, който съдържа тип и данни. В него могат да се съдържат интеджър, флоат, низ, таблица и лист. Типа му може да се провери директно, но извличането на данни е препоръчително да стане чрез предоставените функции. Повечето обекти могат да кажат на коя линия е започнала дефиницията им, което е полезно при извеждане на грешки.

### Обработка на данните

При семантична грешка програмата принтира на стандартният изход очаваният тип и редовете и имената на всички обекти съдържащи го.

Loading Test Scene

Object count: 8

Scene desc parser stack:

Scene root : 1

Node List : 17

Single Node : 141

Shape : 146

Expected type: AT\_FLOAT

Failed to load scene /default.scene

За да бъде възможно това, се изполва ръчно менажиран стек от стрингове. Попринцип това е лошо за бързодействието на кода, но в случая се зареждат много външни файлове и наличието на този овърхед е пренебрежим. За да се улесни менажирането на стека и извеждането на грешки се изполват група макроси. Те се грижат за създаването на кадри на стека, освобождането им и консистентния формат на грешките. Някои от тези макроси могат да се видят в следното парче код:

bool load\_scene\_and\_accelerate(const ConfigFile& f, rt::core::Scene& scene, ResourceManager& manager) {

std::vector<std::string> consumer\_stack;

json\_file\* file = (json\_file\*)&f;

JS\_Object root = sjson\_object\_root(file);

STACK\_FRAME(consumer\_stack, "Scene root", root);

JS\_Object scene\_name = sjson\_object\_child(root, "name");

ENSURE\_TYPE(consumer\_stack, scene\_name, AT\_STRING);

std::cout << "Loading " << sjson\_object\_string(scene\_name) << "\n";

STACK\_FRAME взима стека върху който да работи, името на обекта и самия обект. Проверява дали подадения обект има записана линия в файла. Изполва тези данни за да създаде кадъра, да го добави в стека и да направи скрит обект, чиито деструктор да го премахне. ЕNSURE\_TYPE проверява типа на даден обект, като го сравнява с учаквания. Ако са различни, изполва стека за да форматира грешката, извежда на стандартния изход и прекъсва изпълнението на фунцията.

## Изход на програмата – Представяне на крайния резултат чрез SDL2

Представянето на изходните изображения става от отделна от рендериращите нишка. Тя получава като параметър списък с повърхностти или изображения за представяне. Тя инитиализира видео модула на SDL. Ако по някакав причина не успее, терминира цялата програма. След което създава прозореца в който ще бъдат показани текстурите. При грешка, няма какво друго да се направи освен да се терминира програмата. За това се грижи REQIRE макроса. Той е практически ASSERT, който остава и при релийз билдовете.

void present\_rendering(std::vector<rt::core::Surface2d\*>& surfaces) {

if (SDL\_Init(SDL\_INIT\_VIDEO) < 0) {

printf("Failed to init SDL");

exit(0);

}

SDL\_Window\* window = SDL\_CreateWindow("Raytracer", SDL\_WINDOWPOS\_UNDEFINED,

SDL\_WINDOWPOS\_UNDEFINED, WND\_SIZE\_X, WND\_SIZE\_Y, 0);

REQUIRE(window);

В момента нямаме поддръжа за оразмеряване на изображения към други резолюции, и копирането става пиксел по пиксел. За това трябва големината на SDL текстурата която представлява прозореца да бъде в големината на сърфесите на рей трейсъра.

int w = 0, h = 0;

SDL\_GetWindowSize(window, &w, &h);

REQUIRE(w == WND\_SIZE\_X);

REQUIRE(h == WND\_SIZE\_Y);

Поддръжка за промяна на резолюцията по време на изпълнения на програмата няма, а се дефинира по време на компилация от WND\_SIZE\_X/Y. След като прозореца е готов, се завърта един основен цикъл, който проверява за ивенти, и пренарисува прозореца с обновеното изображение.

int current\_image = 0;

bool running = true;

while (running) {

SDL\_Event e;

while (SDL\_PollEvent(&e) != 0)

{

if (e.type == SDL\_QUIT) {

running = false;

}else if (e.type == SDL\_KEYDOWN) {

if (e.key.keysym.sym == SDLK\_n) {

current\_image++;

if (current\_image >= surfaces.size()) {

current\_image = 0;

}

}

}

}

rt::core::Surface2d\* to\_copy = surfaces[current\_image];

SDL\_LockSurface(surface);

for (int y = 0; y < h; ++y) {

for (int x = 0; x < w; ++x) {

putpixel(surface, x, y, to\_copy->pixel(x, y));

}

}

SDL\_UnlockSurface(surface);

SDL\_UpdateWindowSurface(window);

SDL\_Delay(WAIT\_TIME);

}

С вход от клавиатурата могат да се избира кое от изображенията да се показва. Много важно е накравя да се извика SDL\_Delay, за да се освободи процесорно време за рендериращите нишки. Няма смисъл да се губи време за представяне на едно и също.

SDL\_DestroyWindow(window);

SDL\_Quit();

При излизане от цикъла, ресурсите на SDL се освобождават и основната нишка свършва работата си. Когато основната нишка е освободена, целия процес се терминира автоматично.

## Управление на ресурси. Мениджър за споделени ресурси.

Всички ресурси с които работи програмата могат да се разделят на 3 основни типа. Обекти нуждаещи се от деструктор(те поемат контрола над други ресурси, и трябва да ги освободят), Обекти съдържащи само прости структури и обекти които се преизполват няколко пъти в програмата(модели, текстури). Филмовете, които се освобождават накрая на програмата и са показани в точка 3.1 са от първият тип. За вторият тип са предоставени арени от памет, които в даден момент ще бъдат освободени. За последните същестува отделен мениджър, към който могат да бъдат подавани заявки. Ако даден ресурс е зареден, той директно се връща, а ако не е – той се зарежда. В списъка могат да бъдат добавяни и безименни обекти, които обикновено са междинни обекти генерирани от кода. В края на програмата обектите в списъка се освобождават.

Мениджъра за споделени ресурси знае как да зарежда два типа ресурси – текстури и 3д модели. За текстурите използва външната библиoтека std\_image[7], а за моделите assimp[8].

## Рендърър

Рендеръра контролира потока от данни в различните компоненти. Той дефинира конструктор, на който се подават инстанциите на нужните компоненти и два начина за стартиране на рендерирането.

Renderer(const Sampler& sampler,

const Camera& camera,

const Scene& scene,

const Integrator& integrator,

Film\* radiance\_film\_, Film\* normal\_film\_ = 0);

...

void run\_multithreaded(int chunk\_size = 16);

void run\_singlethreaded();

Конструктора само копира подадените референции в вътрешни променливи. Методите за стартиране на работата винаги блокират текущата нишка.

void Renderer::run\_singlethreaded() {

Sampler::SubSampler sub\_sampler =

\_sampler.create\_subsampler(glm::vec2(0, 0),

\_film->get\_surface()->get\_size());

MemoryArena arena(MATERIAL\_MEMORY\_ARENA\_SIZE);

process\_subsampler(sub\_sampler, arena);

}

Самплерите трябва да оперират върху кякаква част от платното. Тук се създава събсамплер, който да работи върху цялото платно. Метода който извършва рендерирането на дадено парче е process\_subsampler. Той приема събсамплера съдържащ в себе си региона върху който се работи и арената за алокиране на временни обекти. Тя се използва предимно за нуждите на материалната система.

void Renderer::run\_multithreaded(int chunk\_size) {

int size\_x = \_film->get\_surface()->get\_size().x;

int size\_y = \_film->get\_surface()->get\_size().y;

int columns = /\* calculate chunk colums \*/;

int rows = /\* calculate chunk rows \*/;

int chunks = columns \* rows;

//generate subsamplers

std::vector<Sampler::SubSampler> samplers;

samplers.reserve(chunks);

for (int i = 0; i < chunks; ++i) {

glm::vec2 pos(/\* calculate chunk start pos(left top) \*/);

glm::vec2 end\_pos(/\* calculate chunk end pos(right bottom) \*/);

glm::vec2 size(end\_pos - pos);

samplers.push\_back(\_sampler.create\_subsampler(pos, size));

}

std::vector<std::unique\_ptr<MemoryArena>> arenas;

int threads = omp\_get\_max\_threads();

for (int i = 0; i < threads; ++i) {

arenas.push\_back(std::unique\_ptr<MemoryArena>(new MemoryArena(MATERIAL\_MEMORY\_ARENA\_SIZE)));

}

#pragma omp parallel for

for (int i = 0; i < chunks; ++i) {

if (do\_not\_render\_flag) {

continue;

}

int tid = omp\_get\_thread\_num();

process\_subsampler(samplers[i], \*arenas[tid]);

arenas[tid]->free\_all();

}

}

Мултинишковия вариант на последната функция е представен отгоре. Неговата първа работа е да раздели полето на части според подадения параметър. Кода за тази част е опростен за да се чете по лесно. След това се създава по една арена за всяка нишка. Поради лиминатции на езика дори и арените са заделени динамично, а за да е сигурно освобождаването им е са използвани умни указатели. След това се използва цикъл за преминаване през всички части и обработката им. Над него е сложена инструкция към компилатора, която му казва да използва parallel for от вградената библиотека OpenMP. Резултата от това е, че тази библиотека ще направи допълнителни нишки с брой логически ядра без едно. Всекa една от новите нишки, заедно с текущата ще получат приблизително еднакъв брой части. Когато основната нишка прключи с своята работа, тя ще изчака останалите нишки да излязат преди да продължи. Този модел на паралелизъм се нарича fork-join.

void Renderer::process\_subsampler(SubSampler& sampler, MemoryArena& arena) {

SubSampler& sub\_sampler = sampler;

Sample\* samples = new Sample[sub\_sampler.max\_samples()];

while (int samples\_count = sub\_sampler.next\_samples(samples)) {

for (int i = 0; i < samples\_count; ++i) {

Ray ray;

//translate samples from 0..PixelCount to 0..1

glm::vec2 original\_position = samples[i].position;

samples[i].position /= \_film->get\_surface()->get\_size();

\_camera.find\_ray(samples[i], &ray);

Intersection intersection;

if (\_scene.intersect(ray, &intersection)) {

Spectrum color = \_integrator.calculate\_radiance(\_scene, ray, intersection, arena);

arena.free\_all();

\_film->apply\_radiance((int)original\_position.x, (int)original\_position.y, Color(color));

if (\_normals) {

\_normals->apply\_radiance((int)original\_position.x, (int)original\_position.y, Color(glm::abs(intersection.normal)));

}

}

}

}

delete[] samples;

}

Метода process\_subsampler преминава през всички генериран от събсамплера проби, трансформира ги към NDC и намира лъчите от които идва светлината през в тях. Прекарва лъчите през сцената и ако имам попадени подава резултате към интегратора, който знае как да намери спектър от тях. След това подава спектъра, заедно с това от къде е взет към крайния филма от цветове и филма от нормали. Първия съхранява крайното изображения, а втория рендерираната геометрия с цветове според посоките им в пространстово. Изполва се за дебъгване. Може да се забележи използването на класа Color. Това е друг начин за съхранения на цветове, които е по разпространен в съхранението на изображения. Color е 4 байта ARGB, a Spectrum e 3 реални числа RGB по 4 байта. Работата в по-голям формат е с цел да няма големи загуби на цветовете.

## Самплер

Самплера е обект, който създава СъбСамплери. Генерирането на проби се извършва от СъбСамплерите. Те съдържат в себе си и интервала, от който да взимат пробите. Този модел е избран с цел, ако се наложи поддръжката на нови видове самплер, представените тук класове просто да се превърнат в интерфейси. Самплера би станал фабрика. СъбСамплерите предоставят конструктор, който не приема аргументи. Причината за това е, че СъбСамплерите често се съдържат в масиви, които няма начин да бъдат инитиализирани на веднъж.

class Sampler {

float sampling;

public:

Sampler(float samples\_per\_pixel);

class SubSampler {

glm::vec2 \_pos, \_size;

int \_max\_samples;

int \_current\_position;

float \_sampling;

public:

SubSampler(glm::vec2 pos, glm::vec2 size, float sampling);

SubSampler() {}

int max\_samples();

int next\_samples(Sample\* samples);

};

SubSampler create\_subsampler(glm::vec2 pos, glm::vec2 size) const;

};

Взимането на проби, често е най-оптимално, когато се прави на веднъж. За това има функция max\_samples. Тя обозначава за този тип самплер, колко проби могат да се генерират най-оптимлано на веднъж. Това число трябва да е съобразено с факта, че мястото за тези проби ще бъде заделено динамично. Функцията next\_samples, приема масива заделен предварително и го попълва с проби. Връща броя на взетите проби – може да е по-малък от max\_samples. Ако броя на генерираните е нула, значи че други проби няма да бъдат взимани и този събсамплер е приключил работата си.

int Sampler::SubSampler::next\_samples(Sample\* samples) {

int completed\_samples = \_current\_position;

int target\_samples = (int)(\_size.x \* \_size.y \* (\_sampling \* \_sampling));

if (target\_samples > \_max\_samples + completed\_samples){

target\_samples = \_max\_samples + completed\_samples;

}

for (; \_current\_position < target\_samples; ++\_current\_position) {

int row = \_current\_position / (int)(\_size.x \* \_sampling);

int column = \_current\_position % (int)(\_size.x \* \_sampling);

samples[\_current\_position - completed\_samples].position =

\_pos + glm::vec2(column / \_sampling, row / \_sampling);

}

return \_current\_position - completed\_samples;

}

Имплементацията на стандартния събсамплер, е доста проста. Намира се броя на пробите, които са останали за взимане. Това е броя на всички проби за целия регион на събсамплера без броя на генерираните до този момент проби. Ако те са повече от позволените за една итерация, се маркира да се изгенерират пробите от мястото до което е стигнал самплера до същото място плюс максималните проби. Ако не – взимат се всички останали. Минава се през всяка една проба и се изчислява точната и позиция. Алгоритъма се възполва от характеристиката на този метод, позволяваща последователната номерация на пробите.

## Камера

Камерата има конструктор, който напълно построява инстанцията и. Тя не се променя повече. Ако е нужен друг изглед – създава се друга камера.

class Camera {

public:

Camera(glm::vec3 eye, glm::vec3 target, float fov, float aspect\_ratio) {

\_camera\_to\_world = create\_camera\_to\_world(eye, target);

//we are interested in their halves;

fov /= 2;

float yfov = fov / aspect\_ratio;

float x\_coeff = 1 / glm::sin(to\_radians(90 - fov));

float y\_coeff = 1 / glm::sin(to\_radians(90 - yfov));

\_field\_of\_view.x = x\_coeff \* glm::sin(to\_radians(fov));

\_field\_of\_view.y = y\_coeff \* glm::sin(to\_radians(yfov));

//this operation would look better in the find\_ray funtion

//but here it is calculated just once.

\_field\_of\_view \*= 2.0f;

}

void find\_ray(Sample sample, Ray\* ray) const;

private:

glm::mat4x4 \_camera\_to\_world;

glm::vec2 \_field\_of\_view;

};

Първата инструкция инитиализира матрицата, отговаряща за превръщането на лъчи от локалната координатана система на камерата към тази на света(или сцената). Под капака create\_camera\_to\_world прави следното:

glm::mat4 matrix(1.0f);

matrix = glm::lookAt(eye, target, glm::vec3(0, 1, 0));

matrix = glm::inverse(matrix);

Математическите библиотеки обикновено са създавани за работа с интерфейси като OpenGL и DirectX. В приложенията базирани на тях, данните най-често са в локална за съотвения модел координатна система. Те трябва да се превърнат в координатаната система на света, а след това в тази на камерата. Обработката на данните е на обратно – за това в този случай тук, резултата от функцията lookAt трябва да бъде инвърснат?.

След като матрицата е готова, трябва да се изчисли вектора field\_of\_view. Той съдържа коефициентите които определят колко са разперени лъчите. За да се намерят, се изполва синусова теорема и се приема че разстоянието на платното е на точно 1-ца от окото. Целта е да се намери големината на платното, ако се знае ъгъла и растоянието му. Размерите след това се ползват като коефициенти. Тъй като пробите са е интервала [-0.5; 0.5], а не в [-1; 1], коефициентите се умножават веднъж по 2.

void Camera::find\_ray(Sample sample, Ray\* ray) const {

ray->origin = glm::vec3(0, 0, 0);

glm::vec2 orientation = sample.position - glm::vec2(0.5, 0.5);

orientation \*= \_field\_of\_view;

ray->direction = glm::normalize(glm::vec3(orientation, 1));

ray->origin = glm::vec3(\_camera\_to\_world \* glm::vec4(ray->origin, 1));

ray->direction = -glm::vec3(\_camera\_to\_world \* glm::vec4(ray->direction, 0));

}

В координатаната система на камерата, тя се намира на координати (0,0). За да разпределим лъчите по равно от 2-те страни от центъра на платното, изваждаме от пробите (0.5, 0.5). След което се взима в предвид ъгъла на камерата и се умножава по field\_ov\_view. Пробите се превръщат в вектори към платното от камерата и се нормализират. Всички посоки трябва да са нормализирани. Следващите два реда превръщат векторите в координатаната система на света.

## Интегратор

Интеграторът е дефиниран като интерфейс с един метод:

virtual Spectrum calculate\_radiance(const Scene& scene, Ray ray, Intersection isect, MemoryArena& arena) const = 0;

Рей-тресъра предлага една имплементация на този интерфейс – Пътищния интегратор. При извикването му, първата му работа е да инитиализира източниците на превдо-случайни числа. Изполва се стандартната библиотека на C++ 11. Алгоритъма mt19937 се сийдва от случайното устройство rd. След това се създава обект който да разпределя числата от последния обект в даден интервал – от 0 до 1.

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<float> dis(0, 1);

След това се инитиализират няколко основни променливи. Целта на метода е да изчисли каква светлина идва по даден лъч. Path\_throughput показва до колко влиае лъча в текущата итерация-отскок. L е акумулираната светлина. Max\_depth е максималния брой на отскоците.

Spectrum path\_throughput(1, 1, 1), L(0,0,0);

const int max\_depth = 3;

След това се започва последователното проследяване на лъча. То не е рекурсивно - след всяко рефлектиране има само един вторичен лъч. За всяка итерация са дефинирани няколко важни параметъра – isect и ray. Това е текущия лъч и мястото в което се пресича с сцената. Преди преминаване на следващата итерация, двата параметъра се обновяват.

for (int bounces = 0; ; ++bounces) {

const Material& mat = scene.material\_by\_id(isect.material);

L += mat.emitted \* path\_throughput;

BSDF\* bsdf = mat.get\_brdf(isect, arena);

glm::vec3 outgoing = -ray.direction;

Намира се материала на обекта и както е дефинирано в рендериращото уравнение се добавя светлината излъчвана от него. Материала си създава обект, който да опише двупосочната му дистрибуторна функция. След това се създава променливата outgoing, която дава по описателно име на посоката на лъча в контекста на сечението. Следва взимането на проби от източниците на светлина.

============================================================

Spectrum sampled\_light\_contribution;

for (int i = 0; i < SAMPLE\_LIGHT\_COUNT; ++i) {

//choose random emitting object in the scene

Node& node = scene.sample\_light(dis(gen));

//choose random position on the surface of this object

glm::vec3 light\_pos = node.sample\_position(dis(gen), dis(gen), dis(gen));

glm::vec3 light\_incident = light\_pos - isect.position;

float light\_distance = glm::length(light\_incident);

light\_incident = glm::normalize(light\_incident);

Първата работа на алгоритъма е да избере на случаен принцип един източник на светлина. След това се намира случайна позиция на повърхността на обекта. Препоръчително е за източници на свелина да се използват прости триъгълни мрежи или сфери, защото алгоритъма за избиране на еднакво разпределни точки върху повърхносттите е значително опростен. Накрая се намира посоката от която идва светлината от избраната точка. Следва проверка за видимост. Построява се допълнителен лъч за да се провери дали има нещо прекъсва връзката между 2-те точки.

Intersection temp;

Ray tray; tray.origin = isect.position; tray.direction = light\_incident;

if (!scene.intersect(tray, &temp)) {

continue;

}

if (temp.d < light\_distance) {

continue;

}

Ако има видимост, се прилага рендериращото уравнение за рефлектирания компонент. Взима се в предвид до колко допринася текущата итерация към крайния резултат.

sampled\_light\_contribution += path\_throughput \* bsdf->evaluate\_f(outgoing, light\_incident) \* glm::abs(glm::dot(light\_incident, isect.normal));

}

L += sampled\_light\_contribution / (float)SAMPLE\_LIGHT\_COUNT;

glm::vec3 incident;

float pdf;

Spectrum f = bsdf->evaluate\_sample\_f(outgoing, &incident, dis(gen), dis(gen), &pdf);

if (glm::length(f) < 0.000001 || pdf == 0.) {

break;

}

path\_throughput \*= f \* glm::abs(glm::dot(incident, isect.normal)) / pdf;

//check lights

if (bounces == max\_depth) {

break;

}

ray.origin = isect.position;

ray.direction = incident;

if (!scene.intersect(ray, &isect)) {

break;

}

}

return L;

## Филм

Филмът консумира потока от данни, филтрира го и записва резулата в дадената текстура. За тази цел той има един метод за обозначаване на радианс на дадена точка – apply\_radiance.

class **Film {**

Surface2d\* \_surface;

int\* samples;

public:

void apply\_radiance(int x, int y, Spectrum value);

Surface2d\* get\_surface();

//The surface2d is destroyed, when the film is.

Film(Surface2d\* target);

~Film();

};

Съответната функция взима спектъра и го преобразува с 4 байтов RGB(1 излипен байт). След това извлича информацията за съотвения пиксел, добавя новия цвят и преизчислява средното аритметично. Записа се поддържа максимално близък до реалната стойност през цялото акумулиране на цветовете за да може да бъде представен без допълнителна обработка от основната нишка.

Color value\_as\_color(value);

int width = (int)\_surface->get\_size().x;

unsigned int samples\_here = samples[y \* width + x];

unsigned int X = \_surface->pixel(x, y).r \* samples\_here;

unsigned int Y = \_surface->pixel(x, y).g \* samples\_here;

unsigned int Z = \_surface->pixel(x, y).b \* samples\_here;

X += value\_as\_color.r;

Y += value\_as\_color.g;

Z += value\_as\_color.b;

samples\_here += 1;

X /= samples\_here;

Y /= samples\_here;

Z /= samples\_here;

samples[y \* width + x] = samples\_here;

\_surface->pixel(x, y) = Color(X,Y,Z, 255);

Този итеративен метод и ниския брой битове за цвят правят така че да се натрупва грешка при всяка итерация, но за целите на дипломната работа може да се пренебрегне.

## Индексиращи структури

Работата на сцените е да събират цялата информация за виртуалната среда на едно място. Те не знаят как да търсят ефективно обектите с които се пресича един лъч. Индексиращите структури са допълните структури които подпомагат този процес. Обикновено това са дървета, чийто клони са ориентирани по някакъв начин в пространството. В тази дипломна работа е имплементиранa интексираща структура наречена Kd-tree в най-простия и възможен вариант.

Дървото се състои от вътрешни и външни възли. Всеки вътрешен възел или клон съдържа информация за кои са децата на тези клони и как са разположение в пространството. Ако възела е външен, той съдържа списък с елементи.

class KdTreeNode {

public:

enum NODE\_TYPE {

NT\_LEAF,

NT\_INTERIOR

} node\_type;

AABB self\_aabb;

//if leaf

std::vector<int> elements;

//inf interior

float split;

int axis;

KdTreeNode\* left;

KdTreeNode\* right;

};

Основните действия за една индекстираща стурктура са търсенето за пресичания и построяването и. Построяването на дървото е рекурсивен процес.

KdTreeNode\* build\_tree(AABB nodeBounds, int\* elements, int nPrims, int depth) {

KdTreeNode\* node = new KdTreeNode;

node->self\_aabb = nodeBounds;

if (depth >= KDTREE\_MAX\_DEPTH || nPrims <= KDTREE\_MIN\_ELEMENTS) {

node->node\_type = KdTreeNode::NT\_LEAF;

node->elements.resize(nPrims);

for (int i = 0; i < nPrims; ++i) {

node->elements[i] = elements[i];

}

return node;

}

Терминирането на рекурсията се случва в началото на функцията, ако се изпълнени едно от две условия – достигната е максималната дълбочина на дървото или минималните елементи които са позволени в краен възел. Ако не – по някакъв начин се избира ординатата на делене на клетката.

glm::vec3 d = nodeBounds.\_max - nodeBounds.\_min;

int axis;

if (d.x > d.y && d.x > d.z) axis = 0; else axis = (d.y > d.z) ? 1 : 2;

След това се изчисляват кутиите на следващите възли от дървото.

glm::vec3 mid = (nodeBounds.\_min + nodeBounds.\_max) / 2.0f;

AABB left = nodeBounds, right = nodeBounds;

switch (axis) {

case 0:

left.\_max.x = mid.x;

right.\_min.x = mid.x;

break;

case 1: … //\_max.y & \_min.x = mid.y

case 2: … //\_max.z & \_min.z = mid.z

};

След това се минава през всички елементи и се проверява кои, към кои от двата възела принадлежат. Възможно е да се добавят и в двата.

for (int i = 0; i < nPrims; ++i) {

AABB obj = \_adapter.get\_bounding\_box(elements[i]);

if (left.intersect(obj))

left\_list.push\_back(elements[i]);

if (right.intersect(obj))

right\_list.push\_back(elements[i]);

}

След като елементите са разделени, функцията се извикра рекурсивно и връща текущия възел.

node->node\_type = KdTreeNode::NT\_INTERIOR;

node->axis = axis;

node->left = build\_tree(left, left\_list.data(), left\_list.size(), depth + 1);

node->right = build\_tree(right, right\_list.data(), right\_list.size(), depth + 1);

Метода за построяване на дървото не е най-оптималния. Възможни подобрения са изполване на повърхността на отделните елементи за балансиране на дървото и пакетирането на елементите в линеен масив.

Търсенето на потенциални елементи пресичащи се с лъч, става като се преминава рекурсивно през всички възли и се проврява дали кутията на текущия възел се пресича с лъча. Ако не се – обхождането на този клон се прекъсва. Ако да – продължава се и с децата на този клон. Когато се достигне външен възел, елементите в него се проверяват за сечение. Това не се извършва от индексиращата структра.

## Примитивни обекти и форми

В рей трейсъра, геометричните тела – триъгълните мрежи и сферите са представени от един общ интерфейс. Този интерфес скрива формата, като предоставя няколко метода към геометричното тяло – за проверка на пресичане с лъч, за изчисление на кутията и взимане на случайна точка на повърхността му.

class Shape {

public:

virtual bool intersect(Ray ray, Intersection\* result) const = 0;

virtual AABB get\_bounding\_box() const = 0;

virtual glm::vec3 sample(float u1, float u2, float u3) const { return glm::vec3(0, 0, 0); };

virtual ~Shape() {}

};

За някои тела може, да не е възможно да се избере случайна точка върху тях, за това по подразбиране се връща центръра на координатаната система. Класа Shape работи в своя локална координатна система, където центъра обикновено означава и цнтъра на съотвения обект. Предоставен е и витуален деструктор на класа. Класовете наследяващи Shape е нормално да държат динамично заделени ресурси, а освобождаването им да става при унищожаването на инстанцията. В C++, деструкторите на наследните класове се извикват при оператор delete на базов клас, само ако деструктора на базовия клас е виртуален.

Преди следващите точки е хубаво да се представи Intersection структурата. Тя описва с прости типове данни отделните точки от повърхностите – потенциален резултат при сечение лъч – обект.

struct Intersection{

float d;

glm::vec3 position;

glm::vec3 normal;

glm::vec2 uv;

MaterialId material;

};

### Сфера

Най-простия тип геометричен обект е сферата.

class Sphere : public Shape {

glm::vec3 \_pos;

float \_radius;

public:

Sphere(glm::vec3 pos, float radius)

: \_pos(pos), \_radius(radius) {}

bool intersect(Ray ray, Intersection\* result) const;

AABB get\_bounding\_box() const;

};

Принципа по който се намира точна на сечение между права и сфера е базиран на следните факти:

1. |x-c|^2 = r^2, където
   * **х** е която и да е точка от повърхността на сфера
   * **c** e центъра на сферата
   * **r** e радиуса и
2. x = o + dl, където
   * **x** e точка върху дадена линия
   * **l** е посока на линията
   * **o** е начална точка на линията
   * **d** e дистанцията от началната точка до x

Намирането на точките на пресичане е комбинирането на представените уравнения и намирането на d.

### Триъгълник

Триъгълника съдържа в себе си 3 тридименсиални вектора – за точките които го съставят и 3 двудименсиални вектора – за текстурните координати.

class Triangle : public Shape {

glm::vec3 p0, p1, p2;

glm::vec2 uv0, uv1, uv2;

public:

Triangle(const glm::vec3 a, const glm::vec3 b, const glm::vec3 c,

const glm::vec2 \_uv0, const glm::vec2 \_uv1, const glm::vec2 \_uv2);

bool intersect(rt::core::Ray ray, rt::core::Intersection\* result) const;

rt::core::AABB get\_bounding\_box() const;

glm::vec3 sample(float u1, float u2, float u3) const;

};

Принципа на намиране на точка на точката през която един лъч пресича през триъгълник е изведен от преставянето на точките от повърхността на триъгълника чрез барицентрични координати.

X = b1 \* p1 + b2 \* p2 + (1 – b1 – b2) \* p3

Където X e дадена точка от повърхността на триъгълника

P1,p2,p3 са краищата описващи триъгълника

B1, b2 са барицентричните координати.

Намирането на случайна точка от повърхността става чрез взимането на 2 случайни стойностти между 0 и 1 и изполването им като барицентрични координати.

### Триъгълна мрежа

Триъгълната мрежа е представена от класа Mesh, който може да съдържа два линейни масива. Първият е винаги наличен и той съдържа списък с всички точки от модела. Броя на елементите винаги е кратен на 3 защото всички модели се триангулират при зареждането. Точките в масива, групирани по 3 образуват полигоните от които се състои модела. Вторият масив, съдържа UV координатите на отделните върхове от модела. Ако модела няма UV координати, този масив е празен. UV координаните показват каква част от тектурата на даден обект да се използва за съотвения полигон. Координатите попринцип са между 0 и 1, но много системи позволяват изполването на стойности извън този интервал с цел получаването на различни ефекти. Най-често това са повтаряне на текстурата или обръщането и.

std::vector<glm::vec3> points;

std::vector<glm::vec2> \_uvs;

Триъгълнатата мрежа предоставя интерфейс, който позволява изполването на външни индексиращи структури. При заявка за сечение с лъч, се проверява дали лъча пресича осово подравнената очертаваща кутия(AABB) и ако това се потвърди се извиква прикачената индексираща структура да намери индексите на най-вероятните полигони. Меша прочита тези индекси, създава временни обекти от тип Triangle от тях и извиква съответните им методи за сечения.

## Материална система

За описването на начина по който се рефлектира светлина от повърхностите се използват материали. Материалът в тази дипломна е обект, който знае как да генерира правилна поредица от обекти представящи двупосочни дистрибуторни функции(BRDF) в подадена арена за памет. Първата от поддържаните функции е базирана на закона на Ламберт и описва еднакво рефлектираща повърхност от всички ъгли – това е дифузния компонент. Втората е огледано рефлектираща, а третата е за лъскави ефекти. Същестува четвърта функция, която има възможността да съхранява и да смесва други функции.

За да е възможна работата на рей трейсъра, всяка функция трябва да предоставя 3 метода – calculate\_f, calculate\_sample\_f и calc\_pdf. В един идеален свят, бихме имали само calculate\_f. Тази фукнция приема посока на идване и рефлектиране и връща спектъра, който се отразява като коефициент. За да бъде намерена правилната светимост, трябва да се вземат в предвид всички възможни изходни посоки, при които резултата не е 0. Това е невъзможно, защото при перфектните огледални повърхностти, за всеки входен лъч има само един изходен за който отразяването е 0. Поради тази причина са добавени останалтите методи. Sample\_f отговаря на нуждите на Монте Карло алгоритъма. Този метод получава само изходния лъч(този към наблюдателя) и намира входния, след което извиква calculate\_f за да вършне коефициента. С помощта на 3-тия метод, той намира какъв е бил шансът да бъде избран този лъч. Това е друго парче информация нужно на Монте Карло.

Примерна имплементация на огледалната двупосочна функция:

Spectrum evaluate\_sample\_f(glm::vec3 outgoing\_w, glm::vec3\* incident\_w, float u1, float u2, float\* pdf) const {

\*incident\_w = glm::vec3(-outgoing\_w.x, -outgoing\_w.y, outgoing\_w.z);

\*pdf = 1;

return \_R / glm::abs(outgoing\_w.z);

}

Новият лъч е с два обърнати компонента, защото входа и изхода на материалната система са в координатна система на рефлекцията. (0,0,1) вектора е нормалата на точката върху която се работи. За X и Y е гарантирано, че ще са консистентни, но не е определена някаква строга връзка с външната среда. Шансът да бъде избран лъча, на който сме се спряли е 100%, затова pdf = 1. \_R e константа на текущата функция, която определя пропусканите цветове(примерно за цветни огледала). Резултата се дели на косинуса от падащия ъгъл(който съвпада с outgoing\_w.z) за да се неутрализира отпадането на осветеност с ъгъла(неутрализира се умножение което се прави от Монте Карло).

## Обработка на текстури

3.14 Сцена

# ЧЕТВЪРТА ГЛАВА: РЪКОВОДСТВО НА ПОТРЕБИТЕЛЯ

# ЗАКЛЮЧЕНИЕ

# ИЗПОЛЗВАНА ЛИТЕРАТУРА

|  |  |
| --- | --- |
| [1] | Z. Waters, „Realistic Raytracing,“ [Онлайн]. Available: http://web.cs.wpi.edu/~emmanuel/courses/cs563/write\_ups/zackw/realistic\_raytracing.html. |
| [2] | M. Pharr и G. Humphreys, Physically Based Rendering, 2012. |

Съдържание

[Увод 2](#_Toc411347116)

[1 ПЪРВА ГЛАВА: МЕТОДИ И ТЕХНОЛОГИИ ЗА РЕАЛИЗИРАНЕ НА RAY TRACER 3](#_Toc411347117)

[1.1 Основни принципи 3](#_Toc411347118)

[1.2 Технологии за разработване на рей-трейсъри 7](#_Toc411347119)

[1.3 Съществуващи решения и реализации 8](#_Toc411347120)

[2 ВТОРА ГЛАВА: ПРОЕКТИРАНЕ НА СТРУКТУРАТА НА RAY TRACER 9](#_Toc411347121)

[2.1 Функционални изисквания към рей трейсъра 9](#_Toc411347122)

[2.2 Описание на главните алгоритми 10](#_Toc411347123)

[2.3 Избор на език и среда за програмиране 16](#_Toc411347124)

[3 ТРЕТА ГЛАВА: ПРОГРАМНА РЕАЛИЗАЦИЯ НА RAY TRACER 18](#_Toc411347125)

[3.1 Програмата разледана отгоре, инитиализация и кода който държи всичко заедно 18](#_Toc411347126)

[3.2 Вход на програмата - Компилиране и обработка на конфигурационния файл 20](#_Toc411347127)

[3.3 Изход на програмата – Представяне на крайния резултат чрез SDL2 23](#_Toc411347128)

[3.4 Управление на ресурси. Мениджър за споделени ресурси. 24](#_Toc411347129)

[3.5 Рендърър 25](#_Toc411347130)

[3.6 Самплер 28](#_Toc411347131)

[3.7 Камера 30](#_Toc411347132)

[3.8 Интегратор 31](#_Toc411347133)

[3.9 Филм 34](#_Toc411347134)

[3.10 Индексиращи структури 35](#_Toc411347135)

[3.11 Примитивни обекти и форми 37](#_Toc411347136)

[3.12 Материална система 40](#_Toc411347137)

[3.13 Обработка на текстури 41](#_Toc411347138)

[4 ЧЕТВЪРТА ГЛАВА: РЪКОВОДСТВО НА ПОТРЕБИТЕЛЯ 43](#_Toc411347139)

[ЗАКЛЮЧЕНИЕ 44](#_Toc411347140)

[ИЗПОЛЗВАНА ЛИТЕРАТУРА 45](#_Toc411347141)