**Колледж Автономной некоммерческой образовательной организации высшего образования**

**«Научно-технологический университет «Сириус»**

**Реферат**

**по дисциплине: “Введение в специальность”**

**На тему: “2d/3d рендеринг и работа графики в играх изнутри”**

**Работу выполнил:**

**Студент 1 курса группы**

**К0709-24/2**

**Кучеров Данил Максимович**

**Вступление**

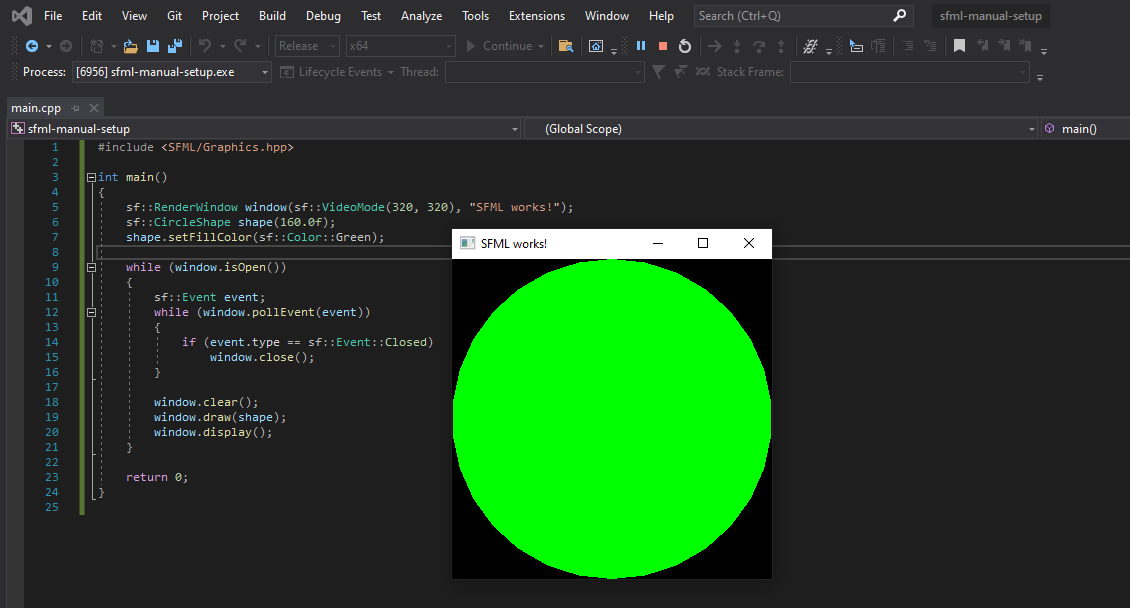
Итак, что же эта тема означает и зачем же я её выбрал? Она довольно широкая и узкая одновременно, многие (наверное) хотят узнать, как работают их любимые компьютерные игры, но, одновременно, мало кто из них захочет в эти дебри лезть. Да, многие сейчас делают игры не с самого нуля, а на игровых движках, по сути инструментах, любезно предоставленных разработчиками для того, чтобы бедным игроделам не пришлось писать 3 тысячи строк кода ради одного кубика. Но всё-же интересно разобраться как же все эти огромные миры, освещения, отражения работают на более низком уровне, какие технологии для этого применяются и к каким хитростям порой прибегают разработчики в угоду оптимизации особенно в то время, когда компьютеры были далеко не такие мощные как сейчас.

**1 GPU и шейдеры**

И начать можно с того, на чём графика запускается. Основные ядра процессора под такие задачи не особо годятся, поэтому в компьютерах стоит либо отдельное ядро в процессоре, отвечающее за графику, либо отдельный модуль со своим чипом - графический процессор, GPU или просто видеокарта. А то, что на видеокарте запускается, те программы, которые говорят чипу, что делать называются шейдерами и по сути из них и состоят эффекты текстурирования, затенения, освещения, отражений и прочего.

**2 2D**

Самая примитивная графика из рассмотренных - конечно двумерная. Здесь вся графика - это просто пиксели закрашенные на мониторе в нужном месте в нужном окне.

***2.1 2д на примере C++ и SFML***

*-Стандартный код из документации SFML с отображением простой фигуры*

Вот пример такой 2д графики - шаблон из документации к графической библиотеке SFML, где в результате видно круг, который в программе - фигура, а на выходе - просто кучка закрашенных пикселей. То же самое происходит и со спрайтами - так называют двумерные объекты - такой спрайт переносится в окно программы вместе с остальными объектами как одно общее изображение.

***3 3D***

Но всё-же 2д графика это не так интересно, просто по сути пиксели закрашиваются на мониторе и поэтому переходим к основной части доклада - 3д графике.

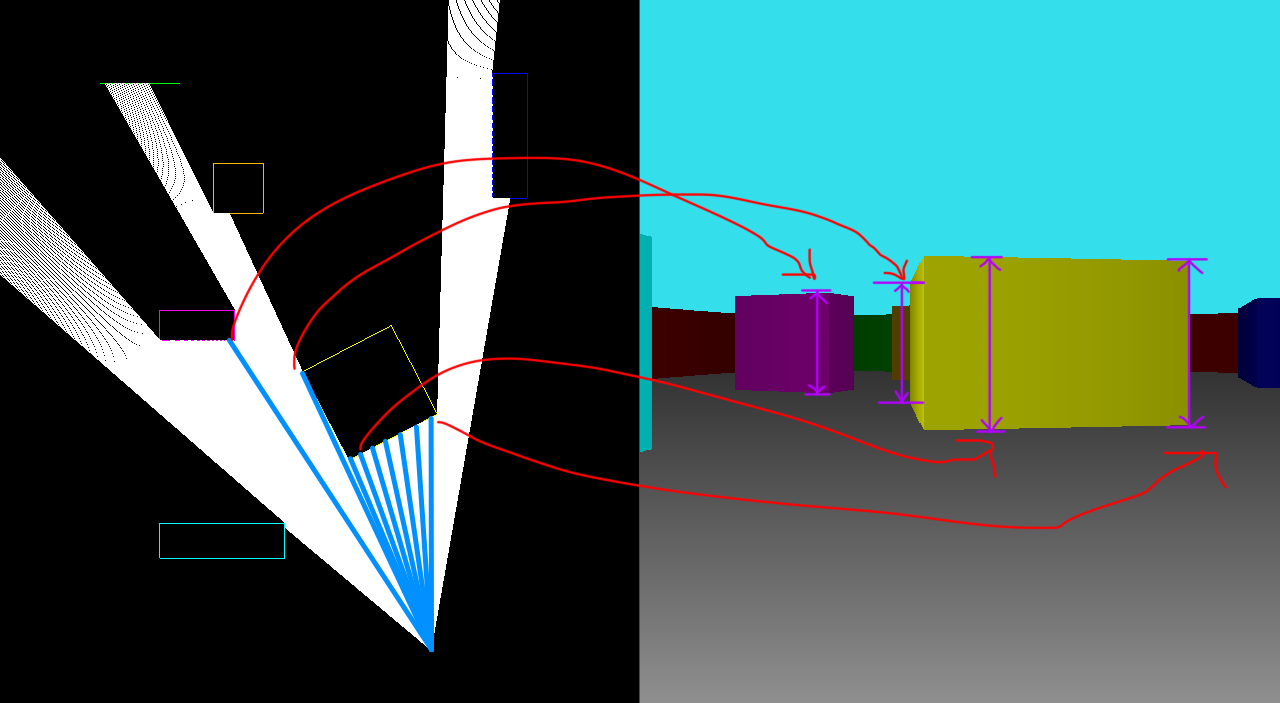
***3.1 Сложности 3д графики***

И тут уже наступают некоторые проблемы и трудности, а дело всё в том, что мониторы наши всё ещё плоские и надо каким-то образом трёхмерные обьекты на них отображать, и есть несколько таких основных способов, которые придумали товарищи разработчики для решения данной проблемы и продвижения графики в играх вперёд.

***3.2 Основные способы отрисовки 3д графики***

И этих основных способа 3: Raycasting, проекция с растеризацией, и ray tracing. Сейчас и рассмотрю каждый из них.

***3.2.1 Raycasting и его работа на примере C++ SFML***

Первый такой способ и по совместительству самый простой - raycasting. Придумали его ещё в далёких 90-х и впервые использовали в массовых играх в первом Wolfenstein 3d 1992 года.

*-Пример и объяснение raycasting*

По сути это не совсем 3д, а псевдо-3д, поскольку сам геймплей и карта двумерные, которые просто переносятся в нечто похожее на 3д графику. Собственно, как оно работает: из камеры всё ещё двумерного игрока испускается много лучей, оттуда как раз и название, они врезаются в объекты, которые должны быть “видимыми”, потом на экране рисуются полосы высотой, зависящей от длины этих самых лучей и в итоге получается такое уменьшение объекта на расстоянии и в совокупности с уменьшением яркости тоже на расстоянии создаётся этот “3д эффект”. У технологии, как можно понять, есть ограничения в плане трёхмерности, например, все объекты получатся одинаковой высоты, ну потому что сами они двумерные, хотя разработчики doom и duke nukem 3d выкрутились и записывали в объекты просто координаты высоты, которые уже должны были считываться движком и сдвигать нужные линии на определённую высоту. Или вот ещё проблемка с чем-то кроме стен, поскольку опять же, трёхмерных данных никаких нет и в итоге те же npc это просто старые добрые двумерные спрайты бегающие про “3д” карте.

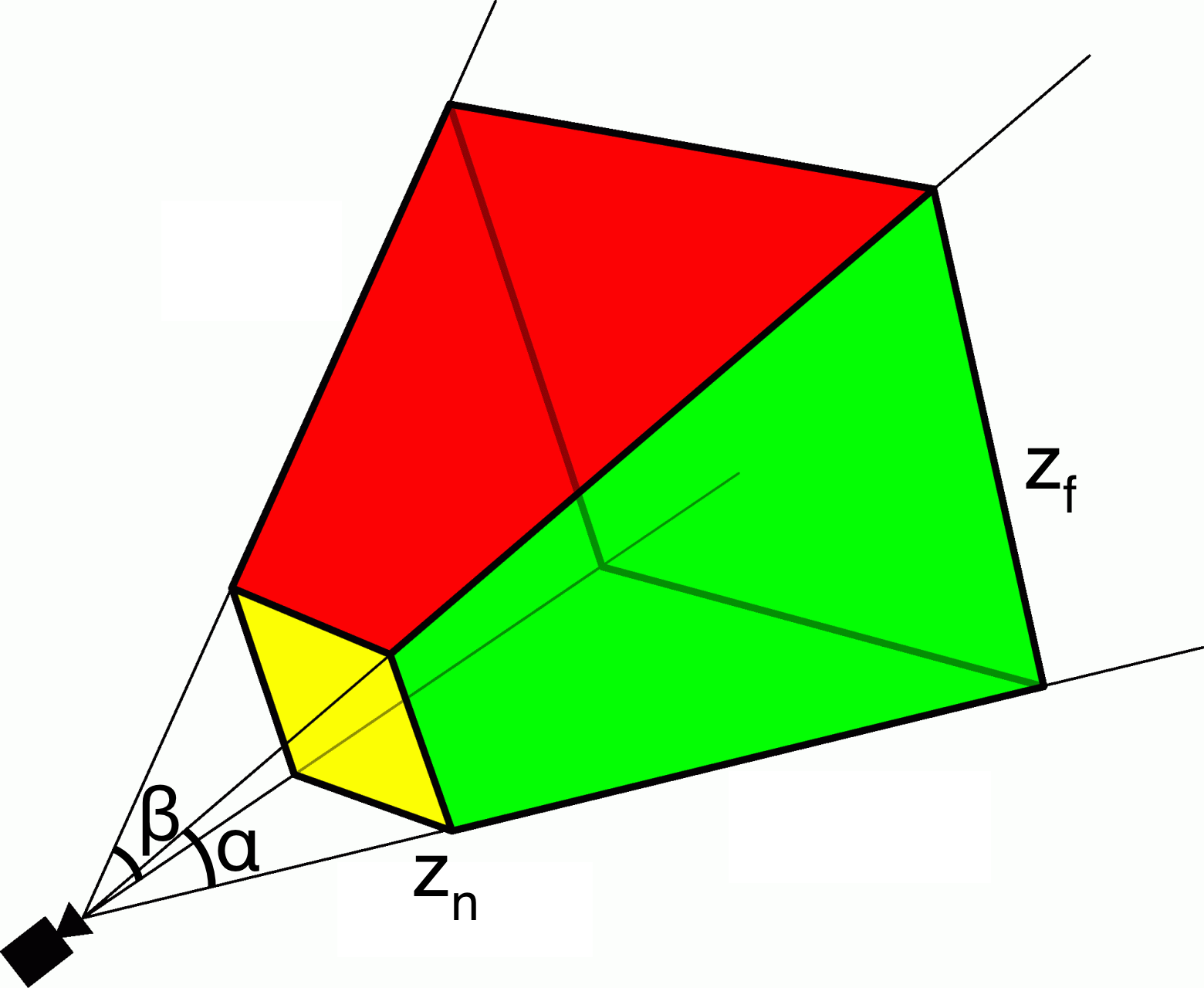
Но надо было разработчикам стремиться дальше, создать настоящую произвольную трёхмерную графику, а не что-то создающее её иллюзию. И поэтому дальше идёт способ, который используется и по сей день во всех популярных игровых движках.

*-Противники в Doom - всё ещё двумерные*

***3.2.2 Проекция***

Вкратце, это способ вычисления позиций трёхмерных координат вершин на плоскости и потом закрашивание областей между ними, но всё по порядку.

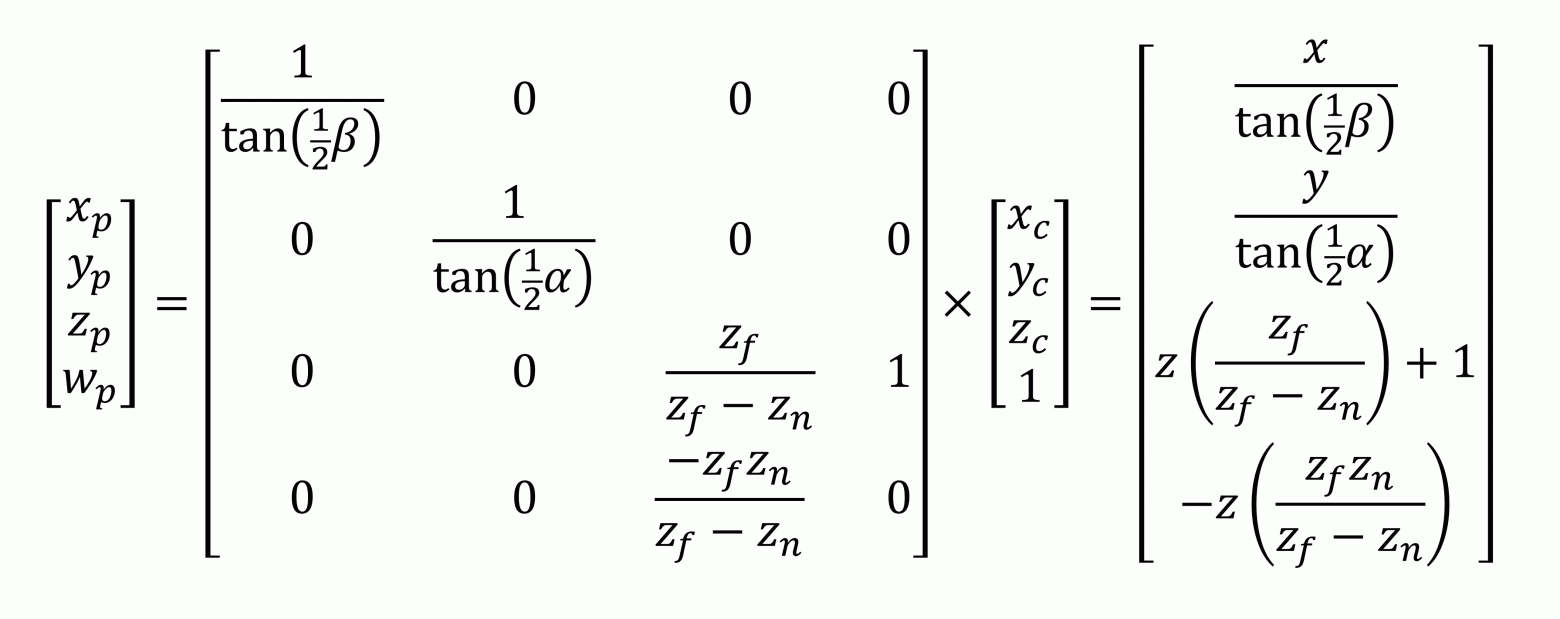
***3.2.2.1 Пирамида усечения (Frustum)***

Прежде чем производить вычисления с координатами вершин, нужно задать область видимости камеры.

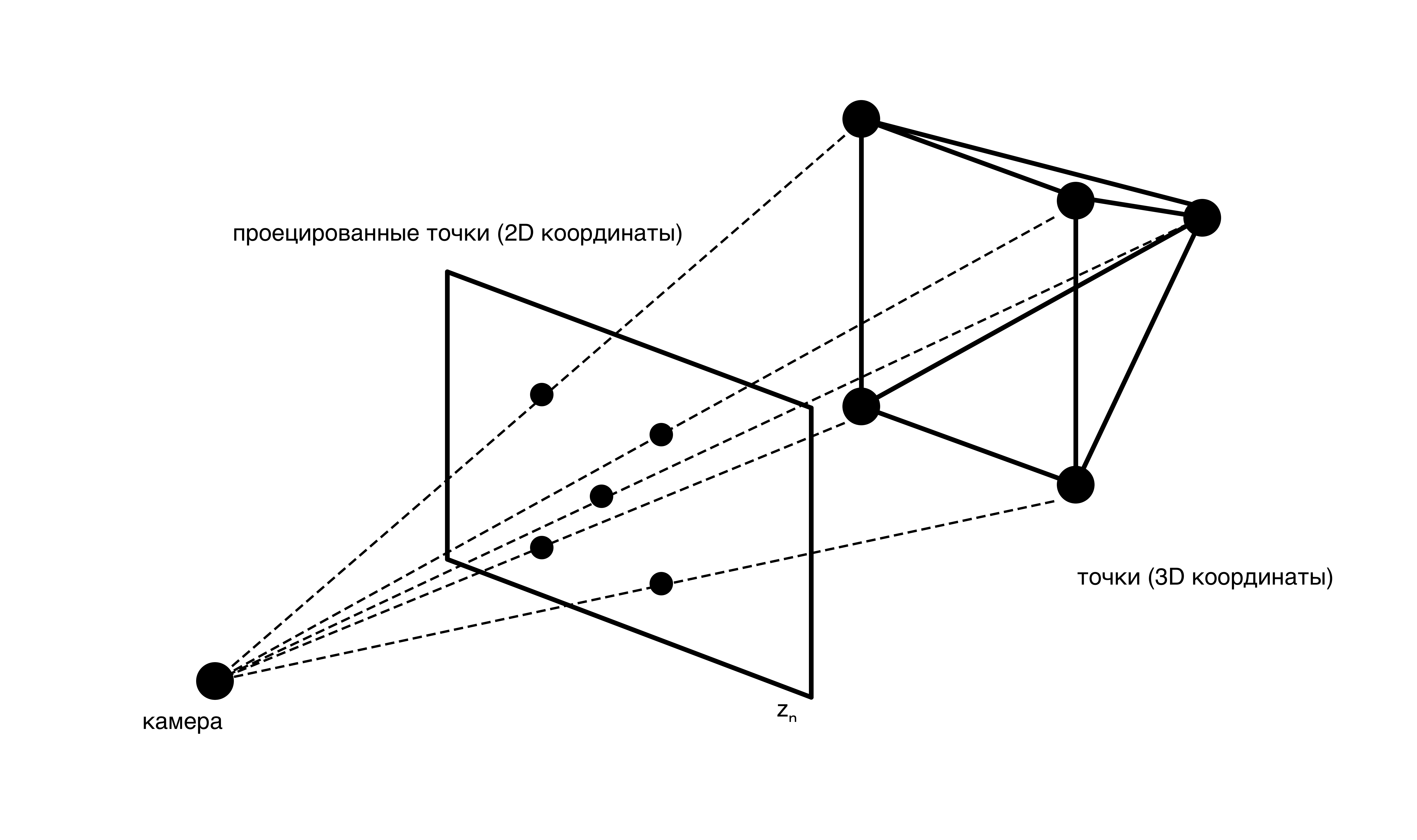
*-“Пирамида” области видимости камеры*

Для этого нужны значения ближней (zn) и дальней (zf) плоскостей, которые определяют насколько близко и далеко будут отрисовываться вершины, и также углы области (FOV): вертикальный (α) и горизонтальный (β). Также размер и соотношение сторон ближней и дальней плоскостей определяются соотношением сторон монитора.

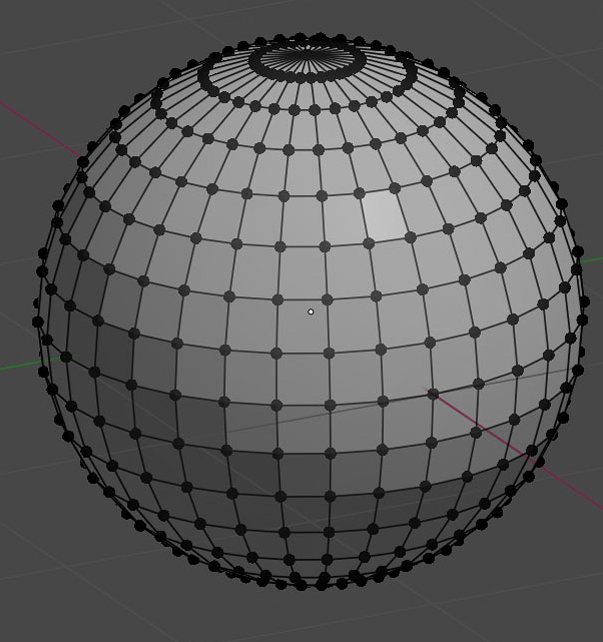
***3.2.2.2 Проекция вершин***

Теперь сами вершины. Способ проекции заключается в переносе трёхмерных координат вершин на ближнюю плоскость (zn) для создания эффекта перспективы на плоском мониторе с помощью перемножения матрицы с трёхмерными кооринатами на матрицу перспективного проецирования.

*-Перемножение матрицы вектора координат на матрицу перспективного проецирования*

Таким образом, на мониторе отображаются вершины того, что в будущем будет 3д объектами

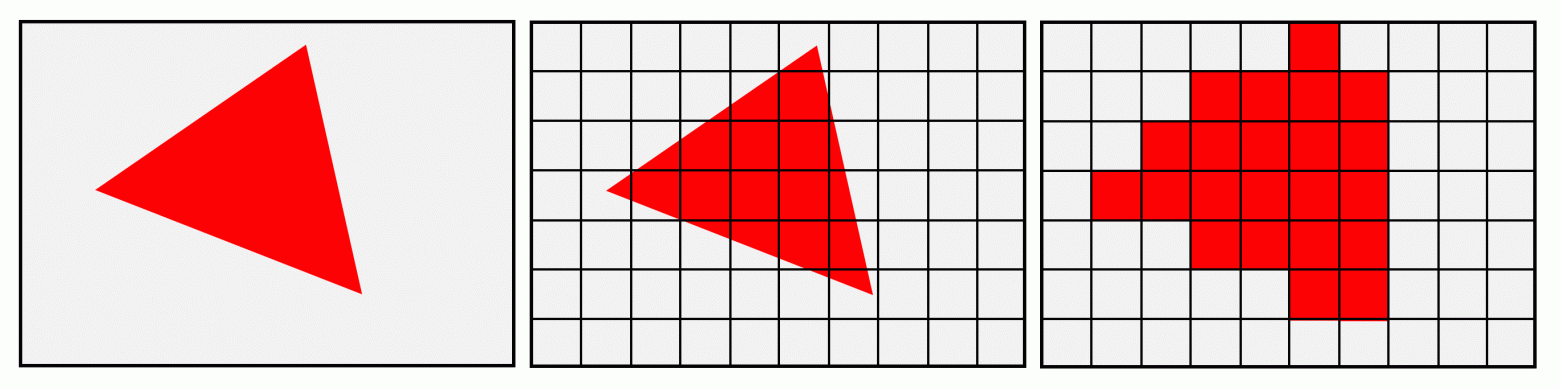
*-Вершины и их проекция на плоскость*

***3.2.3 Из чего состоит 3д модель***

*-3д модель с отображением вершин и рёбер*

Сейчас небольшое лирическое отступление по поводу вершин и из чего состоят трёхмерные объекты если вдруг кто не знал. По сути каждая модель - это набор треугольников (полигонов), которые в свою очередь образованы вершинами - теми самыми точками в пространстве, между которыми рисуются рёбра, а затем это пространство внутри треугольника закрашивается и получается моделька. Теперь обратно к основной теме.

***3.2.4 Растеризация (и ещё математики)***

Так вот затем между этими вершинами образуются рёбра, а потом уже с помощью другой технологии - растеризации, суть которой в закрашивании пикселей на мониторе, которые попадают в область каждого полигона, происходит магия и набор точек превращается в уже нечто похожее на трёхмерную модель. Вот и вся суть этих технологий.

*-Растеризация*

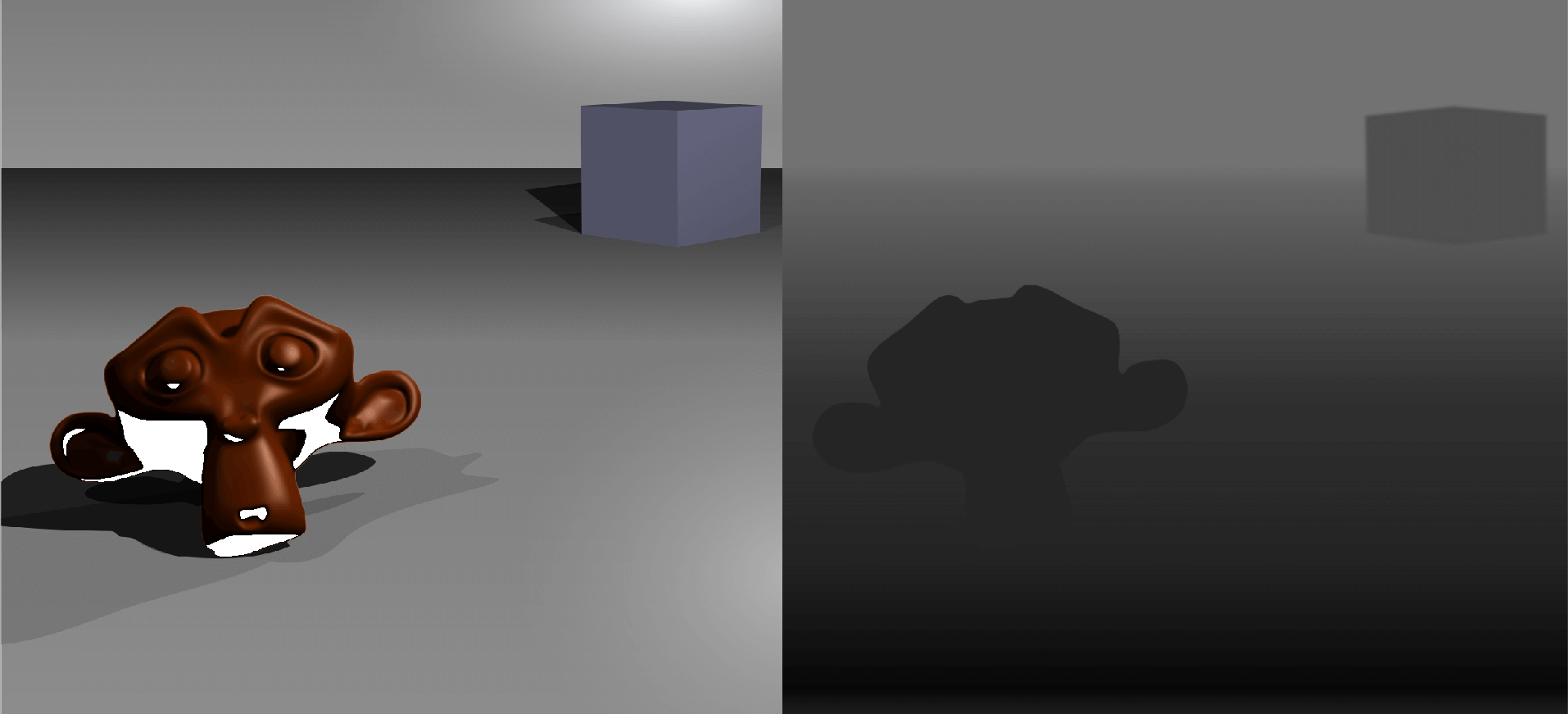
***3.2.5 Raytracing и есть ли в нём смысл***

Ну и немного про технологию трассировки лучей: на самом деле она не новая, существует аж с 60-х годов, а в 2018 году Nvidia представили просто аппаратное ускорение для неё, чтобы вычислениями трассировки занимались отдельно выделенные под это ядра чипа в видеокарте. Трассировка лучей - это технология, которая пускает от каждого пикселя на мониторе луч и полностью отслеживает его движение, параллельно изменяя свойство вроде цвета, пока не достигнет источника освещения. Учитывая, что такое происходит с каждым пикселем, а пикселей таких даже на стандартном 1080p мониторе более 2 миллионов, и вычислять все эти 2 миллиона лучей нужно будет хотя бы 60 раз в секунду для комфортной игры, трассировка становится очень сложной задачей для процессора, поэтому до 2018 никто в играх эту технологию и не думал использовать. И даже когда появилось аппаратное ускорение в видеокартах, использование трассировки в игре вообще везде приводит к падению частоты кадров, например как в Portal RTX, по сути, ремастера оригинального Portal с добавлением трассировки лучей. Проблема тут в том, что добавляется она туда вообще везде, оттуда и производительность, поэтому, как может быть уже понятно, это хороший вариант только для отдельных кадров, где есть время на отрисовку.

3.3 Почему именно полигоны и какие существуют другие решения

Итак, стало понятно, что вся графика в 3д игре состоит из полигонов. Но можно же как-нибудь обойтись без них и использовать с этими же технологиями что-нибудь другое? Действительно, для некоторых идей инди-разработчиков, на которых, как мы знаем, вся игровая индустрия держится, привычное представление 3д графики может не подойти. Ну и в таких случаях можно отойти от полигонов и посмотреть в сторону, например, вокселей, которые являются чем-то наподобие пикселей, только трёхмерных и заполняют всю модель такими кубиками, которые как и пиксели содержат в себе определённую информацию. Благодаря этому получается например более детализированный ландшафт в старых играх как в Comanche: Maximum Overkill. или вот уже относительно новые игры вроде Teardown, где благодаря вокселям достигается полная разрушаемость окружения. Воксели таким образом стали альтернативой полигонам, но предпочтение всё-же отдаётся последним, потому что плюс вокселей в заполнении модели кубиками и возможностью её полностью разобрать выливается в проблемы с производительностью и, соответственно, высокие системные требования таких игр, а именно полигоны потому что именно из этих треугольников можно создать вообще любую фигуру без лишних затрат в плане производительности.

***3.4 Z-буфер***

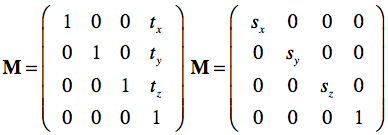
Ну и здесь всё же не так просто, не могут же все эти трёхмерные миры строиться всего лишь на парочке математических формул, ну так с этим тогда вообще любой справится. И действительно, не могут. Во-первых: нужно каким-то образом все полигоны сортировать, ведь какие-то из них должно быть видно, а остальные - нет. Для этого существует z-буфер - такой же массив, но только вместо цветов пикселей туда записываются значения глубины, которые потом используются в шейдерах для проверки видимости и перекрытия.

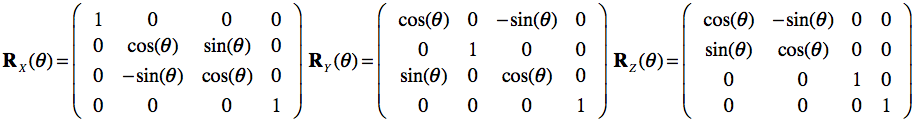
*-Сцена и z-буфер*

***3.5 Движение, вращение и преобразование***

Ну и хотелось бы добавить движения, статический, даже 3д, мир - конечно прикольно, но надоест быстро. Эти самые объекты нужно каким-то образом двигать, вращать и изменять размер. “Ну что же там сложного, угол указываешь, масштаб указываешь, ну всё, готово” казалось бы, но нужно не забывать, что “объект” - это не одно целое, это всё ещё набор вершин, которые нужно двигать так, чтобы получилось что-нибудь наподобие трансформации, и тут уже пригодятся и тригонометрия, и вектора, и матрицы, в общем всё весёлое из математики.

***3.5.1 Матрицы преобразования и зачем в векторах 4-ая ось (w)***

Для передвижения, масштабирования и вращения объекта нужно матрицу (да, и тут матрицы) вектора позиции каждой вершины перемножить с одной из матриц преобразования.

*-Матрицы перемещения и масштабирования (в случае с 1-й вектора складываются, во 2-й перемножаются)*

*-Матрицы вращения по x, y и z осям*

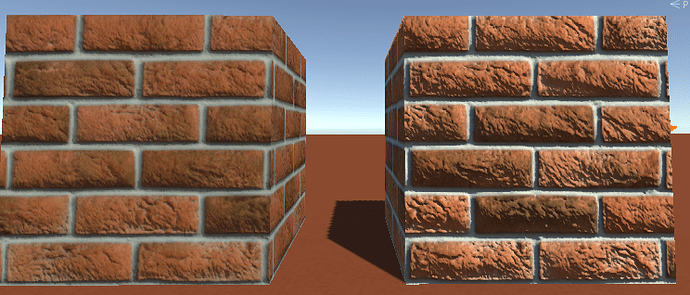
***3.6 Текстурирование***

Текстурирование - процесс наложения текстур на полигоны. У вершин модели есть параметр координаты UV, который отвечает за то, какому пикселю на текстуре соответствует вершина, и, если надо будет, текстура растянется, сожмётся или повторится.

***3.6.2 PBR***

Текстуры, конечно, делают картинку куда лучше, но они плоские. Например, на кирпичную стену свет будет падать как на обычную плоскость и это будет выглядеть скорее как бумажка с изображением кирпичной стены. Но вот же, в современных играх всё выглядит довольно объёмно, свет падает релистично, чуть ли не каждый камешек видно на поверхности, но каким-то образом не влияет на производительность. Дело в том, что вас обманывают, это фальшивая объёмность, потому что если бы это были реальные полигоны, то частота кадров в игре точно бы глаз не радовала. На самом деле, например та же кирпичная стена плоская, просто применяется так называемый PBR (Physics Based Rendering). Это по сути набор текстур, которые накладываются на объект как и все текстуры, но выполняют разные функции.

***3.6.2.1 Normal map***

Самая популярная из них - карта нормалей, в которой в цветовом коде записано то, куда направлена нормаль пикселя, то есть направление, куда должна была бы смотреть плоскость, будь это отдельные полигоны, а также якобы z координата пикселя, где r обозначает x, g - y, а b - z координату - насколько “в глубину” находится пиксель. Используются карты нормалей в основном для обозначения, как должны падать свет с тенями на текстуру или как должно выглядеть отражение на ней.

*-Пример кирпичной стены без карты нормалей и с ней*

***3.6.2.2 Height/Bump map***

Карта нормалей хоть и создаёт иллюзию объёма, но её не хватает, когда игрок движется и видно, что плоскость, извиняюсь за тофтологию, плоская. Эту проблему решает карта высот, на которой обозначено насколько сильно нужно сдвигать пиксели при перемещении игрока в нужную сторону.

*-Смещение пикселей с height map*

***3.6.2.3 Roughness map***

Карта шероховатости - текстура, в которой яркость пикселя обозначает “шероховатость” поверхности, от которой зависит то, как будет выглядеть отражение на ней.

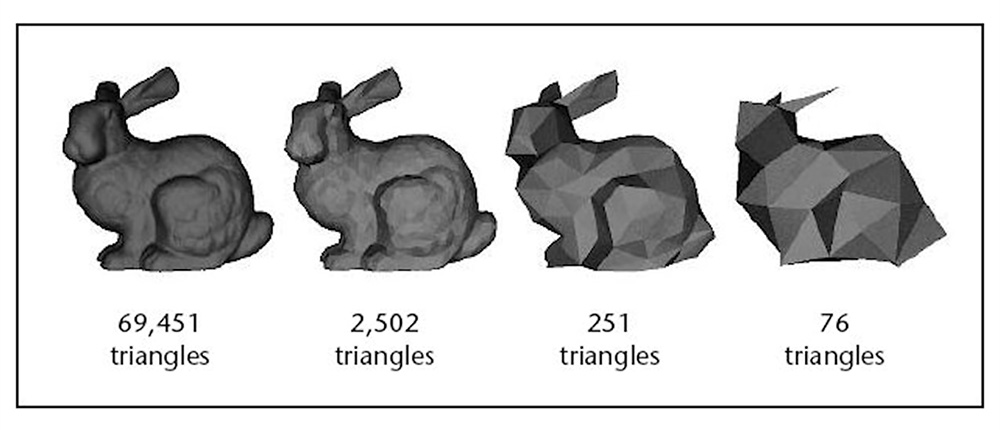
***3.6.2.4 Emission map***

Также, если можно заметить, некоторые части на объектах “светятся”, но при этом свет как таковой не отбрасывают. Это тоже имитация светящихся частей с помощью текстуры, где уже немного больше разнообразия в цветах: \*изображение 420\* можно помечать любым цветом, просто чем темнее пиксель на карте, тем тусклее будет светящаяся часть, т.е. несветящиеся части обозначены просто чёрным цветом, как на примере на изображении 420.

***3.7 Оптимизация (LOD, frustum и occlusion cuiling, back face cuiling, smooth shading)***

Такая проблема: графику нужно создать красивую, модельки детализированные, моделек таких красивых и детализированных на сцене много, ну и полигонов воответственно много. А если полигонов много, значит их все нужно проецировать, а значит вычислять для каждого кадра все эти ранее описанные математические операции для абсолютно каждой вершины на сцене, что очень сильно нагрузит компьютер и соответственно из игры получится уже слайд-шоу. И для решения этого придумали несколько способов оптимизации.

***3.7.1 LOD***

Вот, услышав описание проблемы, и вырисовывается самое простое решение: почему бы не просто уменьшить количество вершин на сцене, а значит количество полигонов, а значит и детализацию моделей, а значит они будут низкокачественными и графика не будет выглядеть так красиво и технологично, как задумали разработчики. Вроде бы не вариант, но именно так и сделали, но с некоторыми правками. Просто, если подумать, игрок не видит в каждом кадре всей картины мира, и всех этих детализированных моделек, и просто на моделях, которые находятся дальше от игрока, и на которые он особо внимание не обращает, можно спокойно, в пределах нормы разумеется, урезать детализацию, а те, которые находятся близко, оставить такими же. Ну и чтобы переход от детализированных к недетализированным моделькам был более плавным и не резал глаз, создаётся сразу несколько уровней с разной детализацией - от самой высокой, которую прямо перед собой будет видеть игрок, до самой низкой, которую тоже будет видеть игрок, но далеко, где разница в деталях при грамотном использовании не будет особо заметна.

*-Модель с разными уровнями детализации*

Таким образом получается и красивые модели сохранить, и оптимизацию улучшить.

***3.7.2 Frustum cuiling***

Как уже ранее сказано, игрок не видит всех полигонов на сцене, больше скажу, не видит и большей их части, особенно где-нибудь в зданиях и лабиринтах. Но они всё-же компьютером учитываются, проецируются и производительность снижают. И как раз Frustum cuiling - это “отсечение” вершин, которые выходят за те самые пределы области видимости игрока из двух плоскостей и двух углов, про которую уже упоминалось ранее при разборе проекции.

***3.7.3 Occlusion cuiling***

Теперь такая ситуация: игрок находится где-нибудь в здании, где большая часть объектов, не отсечённых frustum cuiling’ом, не видно, потому что они все за стеной. Нехорошо бы такое оставлять просто так, когда можно и их отсечь и ещё больше повысить производительность. Это называется уже occlusion cuiling, когда объекты не рендерятся не только когда выходят из области видимости, но и когда они перекрываются другими объектами и рендерить их нет смысла.

***3.7.4 Back face cuiling***

И даже если отсечь объекты, которые находятся за другими объектами, то все равно может остаться геометрия, невидимая игроку - обратная сторона объекта. Это уже back face cuiling - отсечение полигонов, нормали (направление) которых направлены в сторону от игрока. Правда, из-за неправильного использования в виде неправильного направления нормалей некоторых полигонов могут появляться “дыры” в объектах, а если полностью “перевернуть” нормали всех граней объекта, то можно сделать эффект обводки.

***3.8 Как получается делать красивое и иногда незатратное освещение***

***3.8.1 Realtime lighting***

Из названия понятно, что это освещение в реальном времени. При нём постоянно высчитывается “освещение” полигона в зависимости от дальности от источника света. Также такие источники позволяют генерировать тени, как статические, так и динамические, от движущихся объектов.

***3.8.2 Baked lighting и lightmaps***

Только есть одна проблема в realtime lighting - производительность. И, понятно дело, она будет проседать из-за вычислений падения света и динамических теней каждый кадр и ещё хорошо если источников освещения немного. Но каким-то образом при этом удалось создать красивый свет с мягкими тенями и нестыдным глобальным освещением в Uncharted 4, создававшейся для Playstation 4, которая, в свою очередь, подобные вычисления в реальном времени не потянет.

*-Освещение в Uncharted 4*

Ключевая фраза - “в реальном времени”, ведь большинство локаций, где используется этот красивый и реалистичный свет - полностью статичные, а значит, полностью исчезает надобность в просчёте освещения и теней каждый кадр. Поэтому существует так называемое “запекание освещения” - это когда данные о свете и тенях просчитываются всего один раз ещё на этапе разработки и записываются в статичные текстуры под названием lightmaps, которые отдельным слоем накладываются на объекты и создают ощущение освещения, которого на самом деле нет. И, кстати, это будет выглядеть даже лучше, чем освещение в реальном времени, потому что запечённый свет просчитывается с помощью трассировки лучей, оттуда и мягкие тени, и глобальное освещение.

*-Тень от шкафа после перемещения осталась на том же месте*

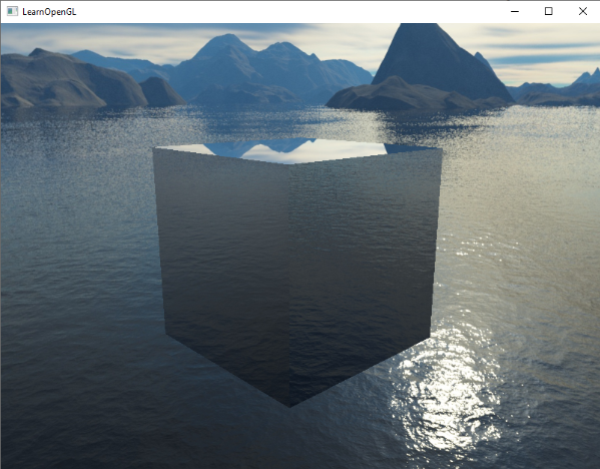
Но работает такое решение правда только на стаитческих объектах, поскольку, опять же, это просто текстуры, и если что-нибудь сдвинуть, то свет все равно останется такой же. Такой косяк есть и в ранее упомянутом Uncharted 4, где на одной из локаций при передвижении шкафа тень от него остаётся на том же самом месте.

***3.9 Отражения***

Отражения также играют не последнюю роль в создании технологичной и красивой картинки, проблема лишь в том, как их отображать при всё ещё растеризации, и здесь есть несколько способов:

***3.9.1 Cubemaps***

Самый примитивный и незатратный - это кубические карты. Всё просто - кубическая карта - это 6 текстур с изображением “окружения”, которые правильно накладываются на объект. Но такой метод не позволяет делать динамические отражения т.е. например, игрока видно не будет.

*-Кубическая карта окружения*

*-Наложение этой карты на куб*

***3.9.2 Планарные***

На этот случай существуют так называемые планарные отражения. Это на самом деле другая камера, которая рендерит ту же самую сцену с другого ракурса и выводит на текстуру. Это уже бьёт по производительности, поэтому как правило они специально в низком разрешении.

*-Планарные отражение в зеркале в Grand Theft Auto V*

***3.9.3 SSR***

SSR (Screen Space Reflections) - Похоже на трассировку лучей, отражает все объекты, но только которые находятся в пределах видимости игрока.

*-Края не отражаются в воде, потому что они вне поля видимости*

***3.9.4 RT***

Отражения, основанные на трассировке лучей. Полностью физически корректные, отражают абсолютно все объекты на сцене.

***3.9.5 Комбинация разных технологий отражений***

Конечно, использование трассировки лучей будет бить по производительности, поэтому её и не только часто комбинируют с другими методами, например, в окнах больших зданий часто используются просто кубические карты. Также они часто используются, чтобы заполнить просторанство, которое не отразило SSR.

**5 Выводы и заключение**

В заключении могу сказать, что графика в современных играх - это не одна какая-то технология, а комбинация множества разных, которые при грамотном использовании разработчиками могут выдать реалистичную и технологически продвинутую картинку даже на слабых устройствах при использовании обмана и некоторых хитростей, и даже при условии существования множества современных игровых движков и графических библиотек определённо есть смысл в понимании того, как всё это работает, чтобы понять как всеми этими технологиями пользоваться.

**Источники**

<https://habr.com/ru/articles/480338/> - Как работает рендеринг 3D-игр: растеризация и трассировка лучей

<https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html> - Spatial Transformation Matrices

<https://habr.com/ru/articles/499540/> - Как работает рендеринг 3D-игр: текстурирование и фильтрация текстур

<https://3dstudio.co/ru/3d-texture-mapping/> - 3D Texture Maps Fundamentals