МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МУРМАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра ЦТМиЭ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНОЙ, САМОСТОЯТЕЛЬНОЙ И РАСЧЕТНО-ГРАФИЧЕСКОЙ РАБОТЫ

По дисциплине: Интерактивные графические системы
название дисциплины

для направления (специальности) _____
код направления (специальности)

«Информатика и вычислительная техника»
наименование направления подготовки

Мурманск
2021

Составитель – Ершов Павел Сергеевич, старший преподаватель кафедры ЦТМиЭ

Методические указания к выполнению расчетно-графической работы рассмотрены и одобрены на заседании кафедры-разработчика: ЦТМиЭ 21.06.2021, пр.№12.

Рецензент – Романовская Ю.В., доцент, к.ф.-м.н.

ОГЛАВЛЕНИЕ

Современный OpenGL / OpenGL ES

OpenGL – это программный интерфейс к графической аппаратуре. Этот интерфейс состоит приблизительно из 250 отдельных команд (около 200 команд в самой OpenGL и еще 50 в библиотеке утилит), которые используются для указания объектов и операций, которые необходимо выполнить, чтобы получить интерактивное приложение, работающее с трехмерной графикой.

OpenGL ES (OpenGL for Embedded Systems — OpenGL для встраиваемых систем) — подмножество графического интерфейса OpenGL, разработанное специально для встраиваемых систем — мобильных телефонов, карманных компьютеров, игровых консолей. OpenGL ES определяется и продвигается консорциумом Khronos Group, в который входят производители программного и аппаратного обеспечения, заинтересованные в открытом API для графики и мультимедиа.

Библиотека OpenGL разработана как обобщенный, независимый интерфейс, который может быть реализован для различного аппаратного обеспечения. По этой причине сама OpenGL не включает функций для создания окон или для захвата пользовательского ввода; для этих операций вы должны использовать средства той операционной системы, в которой вы работаете. По тем же причинам в OpenGL нет высокоуровневых функций для описания моделей трехмерных объектов. Такие команды позволили бы вам описывать относительно сложные фигуры, такие как автомобили, части человеческого тела или молекулы. При использовании библиотеки OpenGL вы должны строить необходимые модели при помощи небольшого набора геометрических примитивов – точек, линий и многоугольников (полигонов).

Преимущества OpenGL

1. Кроссплатформенность
2. Легкость интеграции, написан на чистом C
3. Открыт и бесплатен
4. Высокая скорость работы

Последовательность действий при работе с OpenGL

1. Создание контекста отрисовки, окошка и прочего
2. Инициализация исходных данных (вершины, цвета, текстуры и тд)
3. Инициализация программы для отрисовки (шейдер)
4. Загрузка и инициализация текстур
5. Включение необходимых состояний для отрисовки
6. Вызов отрисовки

Ключевые составляющие современного OpenGL

1. Машина состояний OpenGL
2. Основные понятия 3х и 2х мерной графики
3. VertexBufferObjects (VBO) / VertexAttributesObject VAO
4. Матрицы трансформаций
5. Шейдеры

Машина состояний OpenGL ES

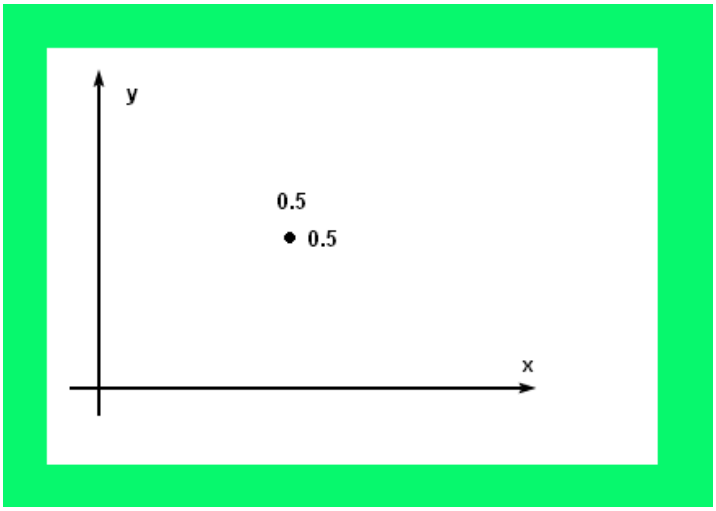
OpenGL – это машина состояния. Вы задаете различные переменные состояния, и они остаются в действии, сохраняя свое состояние, до тех пор, пока вы же их не измените.

Многие переменные относятся к возможностям OpenGL, которые можно включать или выключать командами **glEnable()** или **glDisable()**.

Каждая переменная состояния имеет свое значение по умолчанию, и в любой момент вы можете опросить систему на предмет ее текущего значения. Обычно, чтобы это сделать, используется одна из следующих 6 команд: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, **glIsEnabled()**.

Основные понятия 3х и 2х мерной графики

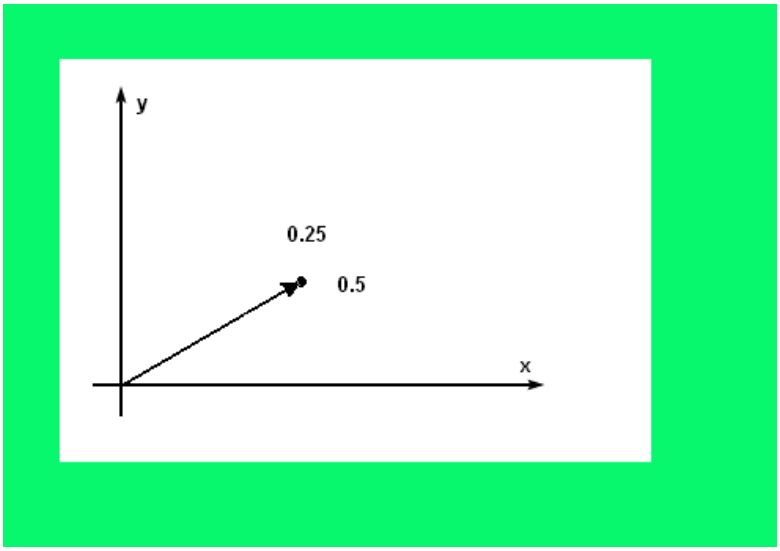
Прежде чем приступать к дальнейшему изучению OpenGL необходимо ознакомится с тем, какими данными мы будем в дальнейшем оперировать. Конечно, вы уже знаете что одна вершина модели характеризуется тремся координатами XYZ, однако для того чтобы работать в OpenGL этого недостаточно.



Вектор - Отрезок произвольной длинны, называется *вектором.* Вектор может быть представлен двумя способами:

Заданием *компонент* вектора.

Заданием *направления* и *модуля* вектора



Если вкратце, то компоненты вектора – это просто координаты конечной точки вектора. Задание трехмерного вектора в виде компонент математически выглядит как:

Модулем вектора A называется длинна вектора, она равна по абсолютной величине квадратному корню из суммы квадратов компонент вектора. Любой вектор характеризуется направлением, которое задается одним или более углов между выбранными осями координат (ортами) и модулем этого вектора.

В трехмерной графике как правило используют вершины состоящие из 4х компонентов, [x y z w]**, где** w - коэффициент, определяющий вектор это или направление

Гомогенные координаты

Изначально предполагается, что вершина расположена по координатам (x, y, z). Давайте-ка добавим еще одну координату – w. Отныне вершины у нас будут по координатам **(x, y, z, w)**

Вскоре вы поймете, что к чему, но пока примите это как данность:

- Если **(w == 1)**, тогда вектор **(x,y,z,1)** – **это позиция в пространстве**
- Если **(w == 0)**, тогда вектор **(x,y,z,0)** – **это направление**.

Запомните это как аксиому без доказательств!!!

И что это нам дает? Ну, для вращения ничего. Если вы вращаете точку или направление, то получите один и тот же результат. Но если вы вращаете перемещение(когда вы двигаете точку в определенном направлении), то все кардинально меняется. А что значит «переместить направление»? Ничего особенного. (направление вроде как не перемещается?)

Гомогенные координаты позволяют нам оперировать единым матаппаратом для обоих случаев.

Матрицы трансформаций

Проще всего представить матрицу, как массив чисел, со строго определенным количеством строк и столбцов.

Однако в трехмерной графике мы будем использовать только матрицы 4×4, которые позволят нам трансформировать наши вершины (x, y, z, w). Трансформированная вершина является результатом умножения матрицы на саму вершину

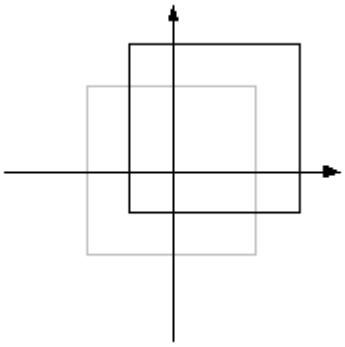
Матрицы позволяют осуществлять следующие операции над вершинами:

- поворот
- перемещение
- скейл
- перспективные преобразования

Компоненты матрицы

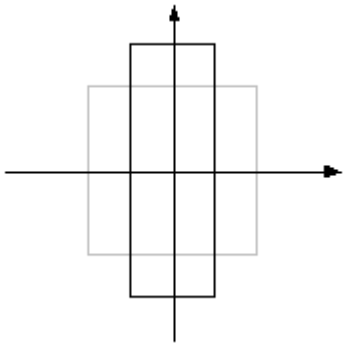
- матрица параллельного переноса:

```
[ 1  0  0  x ]
[ 0  1  0  y ]
[ 0  0  1  z ]
[ 0  0  0  1 ]
```



- матрица растяжения/сжатия:

```
[ z  0  0  0 ]
[ 0  y  0  0 ]
[ 0  0  x  0 ]
[ 0  0  0  1 ]
```

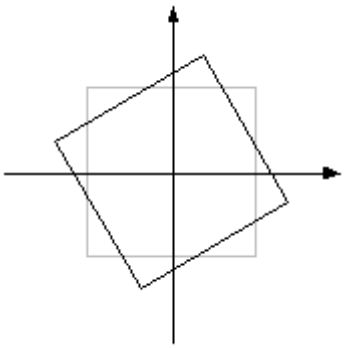


- матрица поворота вокруг оси x, y, z (порядок сверху-вниз):

```
[ 1     0       0     0 ]
[ 0   cos α   sin α   0 ]
[ 0  -sin α   cos α   0 ]
[ 0     0       0     1 ]
```

```
[ cos α   0  -sin α   0 ]
[   0     1     0     0 ]
[ sin α   0   cos α   0 ]
[   0     0     0     1 ]
```

```
[ cos α   sin α   0   0 ]
[-sin α   cos α   0   0 ]
[   0       0     1   0 ]
[   0       0     0   1 ]
```



Пример умножения точки на матрицу

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

Все не так страшно как выглядит. Укажите пальцем левой руки на **a**, а пальцем правой руки на **x**. Это будет **ax**. Переместите левый палец на следующее число **b**, а правый палец вниз на следующее число – **y**. У нас получилось **by**. Еще раз – **cz**. И еще раз – **dw**. Теперь суммируем все получившиеся числа – **ax+by+cz+dw**. Мы получили наш новый **x**. Повторите то же самое для каждой строки и вы получите новый вектор **(x,y,z,w)**.

Так же есть еще одна разновидность матриц - матрица проекции, она будет рассмотрена позже.

Пример с матрицей перемещений

Матрица перемещения, это, наверное, самая простая матрица из всех. Вот она:

```
[ 1   0   0   x ]
[ 0   1   0   y ]
[ 0   0   1   z ]
[ 0   0   0   1 ]
```

Тут **X, Y, Z** – это значения, которые мы хотим добавить к нашей позиции вершины.

Итак, если нам нужно переместить вектор (10,10,10,1) на 10 пунктов, по позиции X, то:

$$\begin{bmatrix}1&0&0&10\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} * \begin{bmatrix}10\\10\\10\\1\end{bmatrix} = \begin{bmatrix}1*10+0*10+0*10+10*1\\0*10+1*10+0*10+0*1\\0*10+0*10+1*10+0*1\\0*10+0*10+0*10+1*1\end{bmatrix} = \begin{bmatrix}10+0+0+10\\0+10+0+0\\0+0+10+0\\0+0+0+1\end{bmatrix} = \begin{bmatrix}20\\10\\10\\1\end{bmatrix}$$

…И у нас получится (20,10,10,1) в гомогенном векторе. Как вы, я надеюсь, помните, 1 значит, что вектор представляет собой позицию, а не направление.

А теперь давайте попробуем таким же образом трансформировать направление (0,0,-1,0):

$$\begin{bmatrix}1&0&0&10\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} * \begin{bmatrix}0\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1*0+0*0+0*-1+10*0\\0*0+1*0+0*-1+0*0\\0*0+0*0+1*-1+0*0\\0*0+0*0+0*-1+1*0\end{bmatrix} = \begin{bmatrix}1+0+0+0\\0+1+0+0\\0+0-1+0\\0+0+0+0\end{bmatrix} = \begin{bmatrix}0\\0\\-1\\0\end{bmatrix}$$

И в итоге у нас получился тот же вектор (0,0,-1,0). Как я и говорил, двигать направление не имеет смысла.

Пример с матрицей масштабирования

Матрица масштабирования так же достаточно проста:

```
[ z   0   0   0 ]
[ 0   y   0   0 ]
[ 0   0   x   0 ]
[ 0   0   0   1 ]
```

Поэтому если вам хочется увеличить вектор(позицию или направление, не важно) в два раза по всем направлениям:

$$\begin{bmatrix}2&0&0&0\\0&2&0&0\\0&0&2&0\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}x\\y\\z\\w\end{bmatrix} = \begin{bmatrix}2*x+0*y+0*z+0*w\\0*x+2*y+0*z+0*w\\0*x+0*y+2*z+0*w\\0*x+0*y+0*z+1*w\end{bmatrix} = \begin{bmatrix}2*x+0+0+0\\0+2*y+0+0\\0+0+2*z+0\\0+0+0+1*w\end{bmatrix} = \begin{bmatrix}2*x\\2*y\\2*z\\w\end{bmatrix}$$

А координата w не поменялась. Если вы спросите: «А что такое масштабирование направления?». Полезно не часто, но иногда полезно.

Совмещенные трансформации

Теперь мы знаем как вращать, перемещать и масштабировать наши вектора. Хорошо бы узнать, как объединить все это. Это делается просто умножением матриц друг на друга.

TransformedVector = TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;

И снова порядок!!! Сначала нужно изменить размер, потом прокрутить и лишь потом сдвинуть.

Если мы будем применять трансформации в другом порядке, то не получим такой же результат. Вот попробуйте:

- Сделайте шаг вперед(не свалите компьютер со стола) и повернитесь влево
- Повернитесь влево и сделайте один шаг вперед.

Да, нужно всегда помнить про порядок действий при управлении, например, игровым персонажем. Сначала, если нужно, делаем масштабирование, потом выставьте направление(вращение), а потом перемещайте. Давайте разберем небольшой пример(я убрал вращение для облегчения расчетов):

Не правильный способ:

- Перемещаем корабль на (10,0,0). Его центр теперь на 10 по X от центра.
- Увеличиваем размер нашего корабля в 2 раза. Каждая координата умножается на 2 относительно центра который далеко… И в итоге у нас получается корабль необходимого размера но по позиции 2*10=20. Что не совсем то, чего мы хотели.

Правильный способ:

- Увеличиваем размер корабля в 2 раза. Теперь у нас есть большой корабль расположенный по центру.
- Перемещаем корабль. Размер корабля не изменился и он расположен в нужном месте.

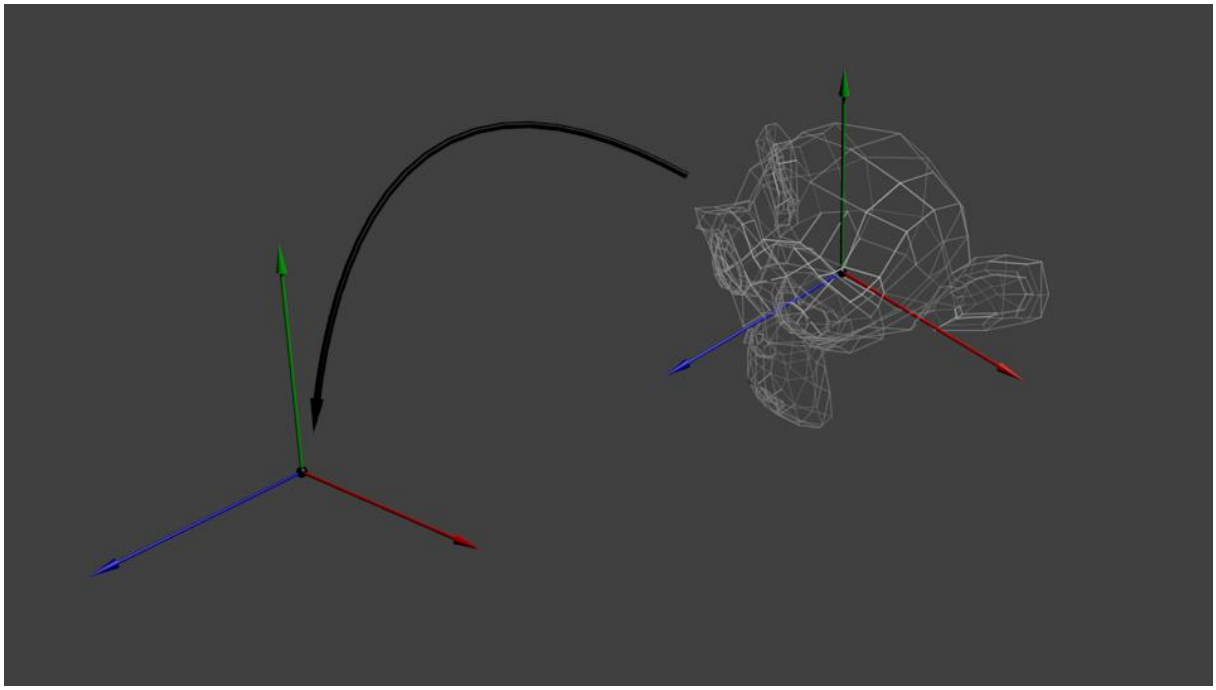
Работа с матрицами в OpenGL

Главная особенность в трехмерной графике и в играх в частности заключается в том, что на самом деле когда мы перемещаемся и крутимся по игровому миру, мир крутится вокруг нас! Почему - будет ясно из описания матриц

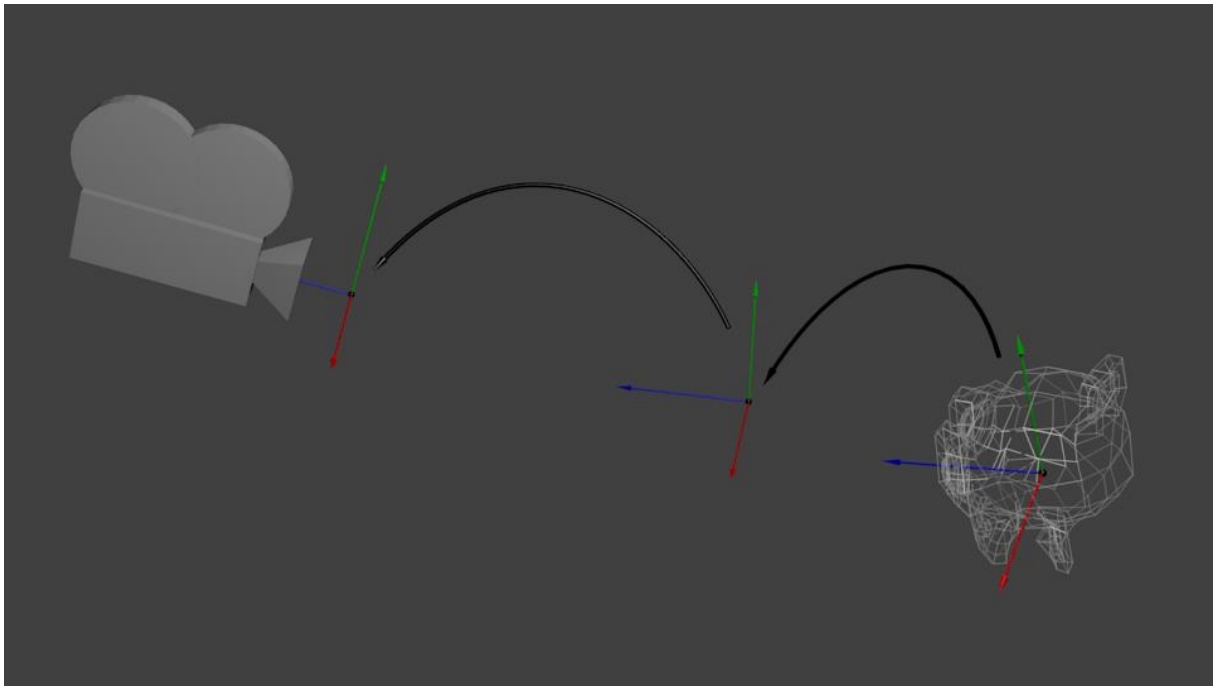
Представим, что наш мир - это трехмерная игра в духе ходим туда-сюда, что-то делаем - тогда на таком примере можно разобраться с матрицами

Как правило при отрисовке используется 3 матрицы, это матрицы

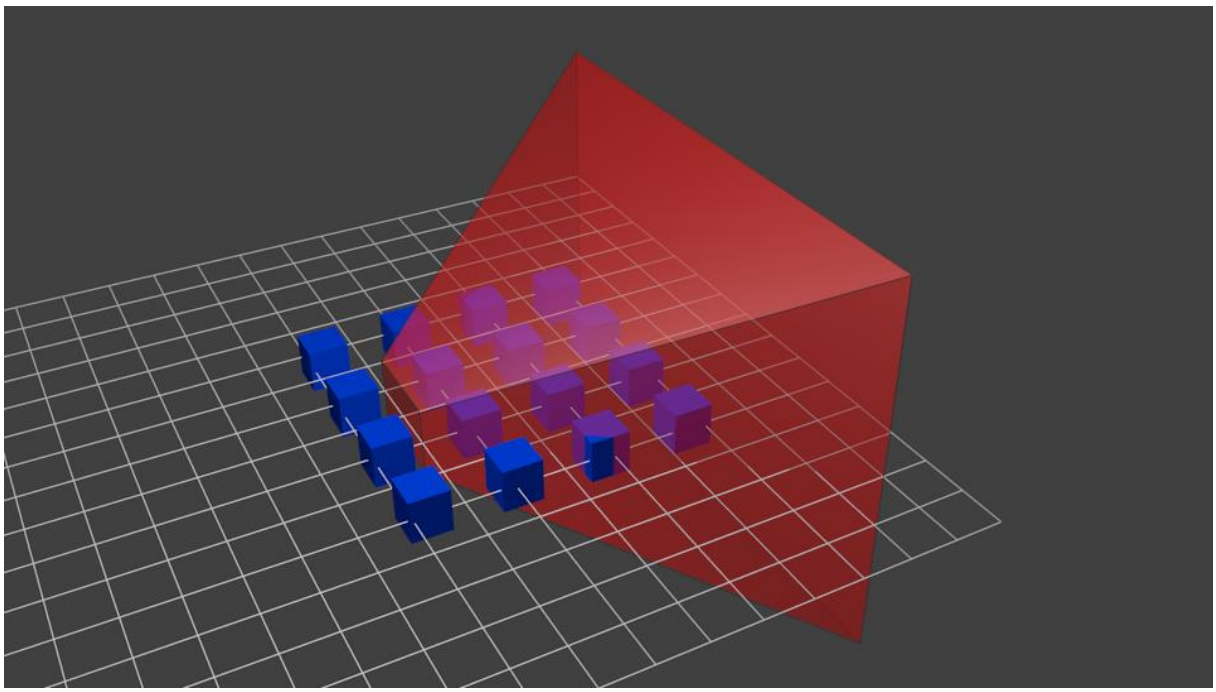
1. **Матрица модели** - представляет собой трансформацию нашей модели в мире, например, перемещение и поворот стула относительно нашего помещения

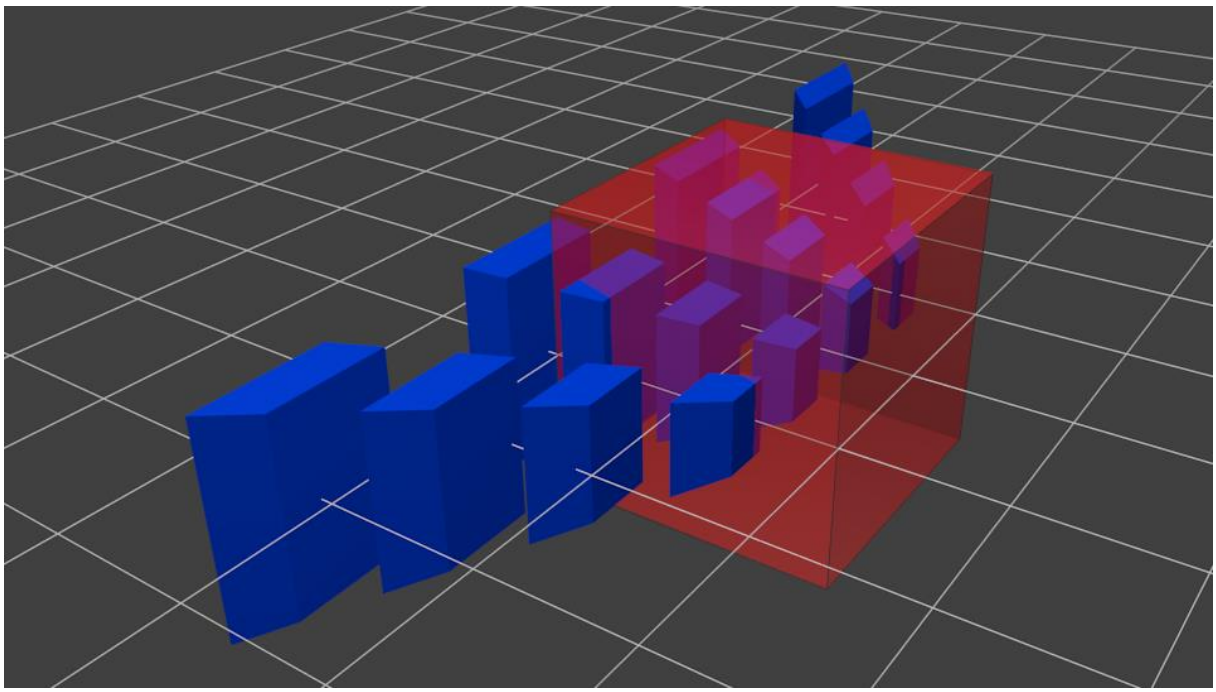


2. **Матрица вида** - простыми словами - это матрица трансформации, которая описывает трансформацию мира относительно нас находящихся в центре координат (мира). По сути когда мы ходим по игровому миру, то мир смещается относительно нас и создается эффект, что мы ходим по игровому пространству **ПРИМЕР ЕЩЕ КАКОЙ-НИБУДЬ**



3. **Матрица проекции -** матрица проекции предназначена для того, что из координат нашего мира, например [-1000, 1000], перейти к координатам OpenGL [-1,1], а так же для реализации перспективной проекции, если она нам нужна





Vertex Buffer Objects (VBO) / Vertex Attributes Object VAO

Одним из самых "узких" мест в работе с современным GPU является передача ему данных - текстур, вершин, нормалей и т.п. Для повышения быстродействия следует уменьшить количество запросов на передачу данных и передавать их как можно большими частями.

VBO (Vertex Buffer Objects) – технология, позволяющая хранить координаты вершин совместно с их атрибутами в видеопамяти.

В отличии от использования блока **glBegin/glEnd**, при котором на каждом кадре вся геометрия передаетсяGPU по очень медленной шине **PCIEx**, при использовании **VBO** все геометрия загружается в видеопамять только один раз, на этапе инициализации, после чего мы просто ссылаемся на эти данные.

1. Это **во-первых** позволят существенно разгрузить шину для более важных задач, **во-вторых** - это приводит к существенному повышению производительности, так как GPU может незамедлительно приступать к рендерингу, не дожидаясь пока будут получены данные от CPU.

2. Ну и пропускная способность видеопамяти в десятки раз выше пропускной способности PCIEx (4Гб/сек против 148Гб/сек), благодаря чему все операции копирования в видеопамяти происходят существенно быстрее чем при загрузке данных со стороны CPU. Используя приведенные выше особенности можно существенно повысить как производительность так и гибкость рендера, но обо всем по-порядку.

Использование вершинных буфером в этих командах позволяет GPU более эффективно работать с памятью, повышая быстродействие приложения.

Шейдеры

Шейдер представляет собой небольшую программку написанную на С подобном языке которая описывает как работать с определенным примитивом, например, вершиной (вершинный шейдер) или отдельным фрагментом (для простоты можно воспринимать фрагмент как отдельный пиксель).

В некотором роде это шаг назад, поскольку большая часть 3D функционала ранее была представлена в виде фиксированных функций конвейера и требовала от разработчика только указать конфигурационные параметры (параметры света, значения вращения и т.д).

Теперь же это должно быть реализовано программистом через шейдеры.

Может показаться, что это только усложняет написание приложение, но по факту - это дает огромную гибкость при работе с графикой

Типы данных шейдера

Скалярные:
float, int, bool

Вектора с плавающей запятой:
float – одномерный вектор
vec2 - двух
vec3 - трех
vec4 – четырехмерный вектор соответственно

Целочисленные вектора:
int - одно
ivec2 - двух
ivec3 - трех
ivec4 -четырехмерный вектор

Boolean-вектора: (разрядность по аналогии)

bool
bvec2
bvec3
bvec4

Матрицы:
mat2 – матрица 2x2
mat3 – матрица 3x3
mat4 – матрица 4x4

Пример:
float u_Time; // скаляр с плавающей запятой
vec4 a_Position; // четырехмерный вектор с плавающей запятой
mat4 m_Projection; // матрица 4x4
ivec2 r_Offset; // целочисленный двухмерный вектор

Типы шейдеров

Существует 2 основных типа шейдера без которых не отрисовать какой-то примитив на экране, данные шейдеры обязательно работают в паре:

1. **Вершинный шейдер**
2. **Фрагментный шейдер**

Коммуникация приложения и шейдера

Для связи с нашей программой в GLSL используются специальные типы переменных:
1. Attribute (Вершинные атрибуты)
Используются ТОЛЬКО в вершинном шейдере.
Могут включать в себя такие данные как позицию вершины, нормали, текстурные координаты или любые другие данные, характерные для КОНКРЕТНОЙ ВЕРШИНЫ.
2. Uniforms (Униформы).
Используются как в вершинном так и фрагментной шейдерах.
Могут включать в себя любые данные ЕДИНЫЕ ДЛЯ ВСЕГО ОТРИСУЕМОГО МЕША.
Например время или положение и поворот всей модели, матрицы преобразований или матрицу вида и т.д.
3. Varying
Используются ТОЛЬКО для связи между вершинным и фрагментным шейдером.
Являются исходящими данными из вершинного шейдера.
На входе в фрагментный шейдер представляют из себя интерполированное значение между вершинами примитива.
4. Sampler (текстура)
Могут использоваться как в фрагментном так и в вершинном.
Не все GPU поддерживают работу в текстурами в вершинном шейдере.

Вершинный шейдер

Представляет собой код, который описывает как у нас происходит трансформация каждой отдельной вершины нашего примитива. В данном шейдере происходит

перемножение координаты вершины на матрицы трансформации для получения координаты вершины в пространстве OpenGL

Пример вершинного шейдера шейдера

```
attribute vec4 aPointCoord; // attribute - аттрибут вершины
attribute vec3 aColor; // attribute - аттрибут вершины
uniform mat4 uModelViewProjection; // uniform - данные из приложения
varying vec3 vColor; // varying - параметры, передаваемые в
фрагментный шейдер

void main() {
      gl_Position = uModelViewProjection * aTriangleCoord;
      vColor = aColor;
}
```

Фрагментный шейдер

Фрагментный шейдер - код, который обрабатывает каждый полученный на предыдущих стадиях ГК фрагмент (не пиксель). Обработка может включать такие стадии, как получение данных из текстуры, просчет освещения, просчет смешивания. Обязательной работой для фрагментного шейдера является запись цвета фрагмента во встроенную переменную **gl_FragColor**. В случае отбрасывания фрагмента, никакие расчеты дальше с ним производиться не будут, и фрагмент уже не попадет в буфер кадра.

Представим, что мы рисуем треугольник. Теперь когда вершинный шейдер обрабатывает положение 3х точкек на экране, он заполнит пространство между ними, чтобы создать треугольник. Для каждого фрагмента (или пикселя) между 3мя точками будет вызываться фрагментный шейдер.

```
varying vec3 vColor;
uniform float uTime;
void main() {
      gl_FragColor = vColor * sin(uTime);
}
```

Важной особенностью является тот факт, то данные, которые получены из вершинного шейдера (обозначены как **varying**) интерполируются от вершины к вершине

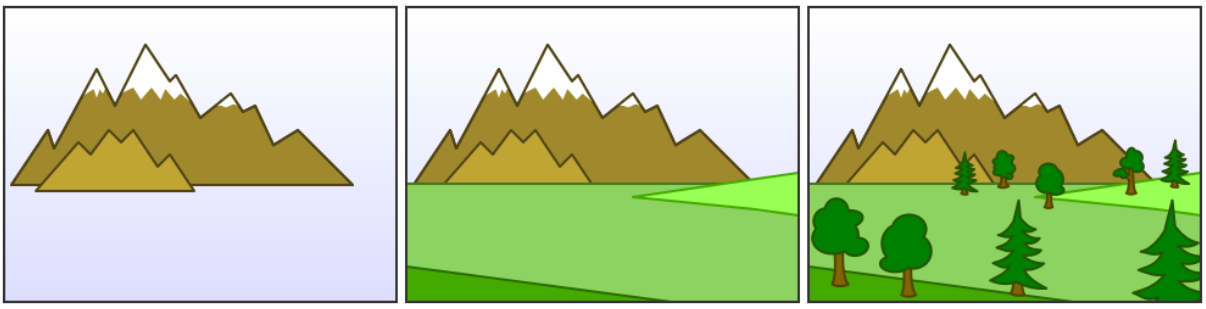
Проблема порядка отрисовки

Алгоритм художника

Алгоритм художника — простейший программный вариант решения «проблемы видимости» в трехмерной компьютерной графике.

Название «алгоритм художника» относится к технике, используемой многими живописцами: сначала рисуются наиболее удалённые части сцены, потом части которые ближе. Постепенно ближние части начинают перекрывать отдаленные части более удалённых объектов. Задача программиста при реализации алгоритма художника —

отсортировать все полигоны по удалённости от наблюдателя и начать выводить, начиная с более дальних.

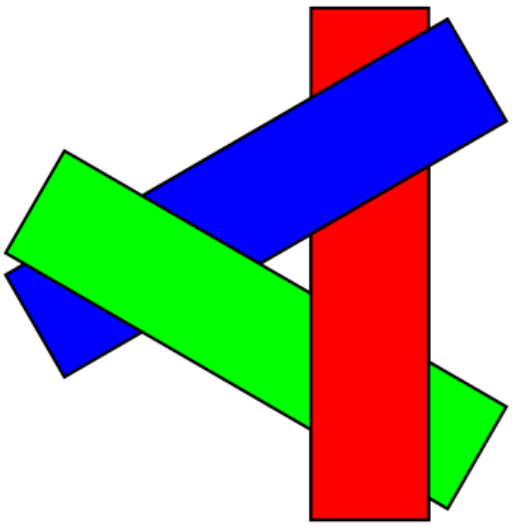


Проблемы алгоритма

Алгоритм не позволяет получить корректную картину в случае взаимноперекрывающихся полигонов. В этом случае, как показано на рисунке справа, полигоны A, B и C накладываются друг на друга таким образом, что невозможно определить, в каком порядке их следует рисовать. В этом случае, следует разбить конфликтный полигон на несколько меньших, например алгоритмом Ньюэлла, предложенным в 1972 году.

Второй распространённой проблемой является то, что система прорисовывает также области, которые впоследствии будут перекрыты, на что тратится лишнее процессорное время.

Эти недостатки привели к разработке метода Z-буфера, который можно рассматривать как развитие алгоритма художника.



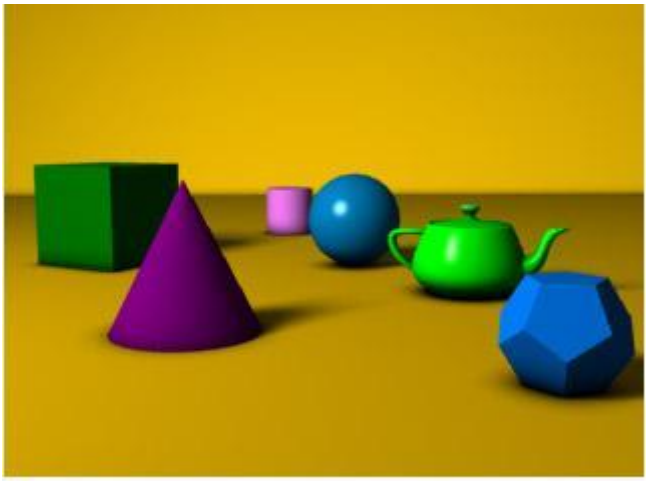
Z-буфер (Буффер глубины)

Буффер глубины предназначен для того, чтобы мы могли выполнять отрисовку трехмерных непрозрачных полигонов в произвольном порядке, при этом, при отрисовке пиксели с меньшим значением Z будут отбрасываться.

Z-буфер представляет собой двумерный массив (Текстуру), каждый элемент которого соответствует пикселю на экране. Когда видеокарта рисует пиксель, его удалённость просчитывается и записывается в ячейку Z-буфера. Если пиксели двух рисуемых объектов перекрываются, то их значения глубины сравниваются (значение

рисуемого пикселя и значение в буффере глубины), и рисуется тот, который ближе, а его значение удалённости сохраняется в буфер.

Получаемое при этом графическое изображение носит название z-depth карта, представляющая собой полутоновое графическое изображение, каждый пиксель которого может принимать до 256 значений серого. По ним определяется удалённость от зрителя того или иного объекта трехмерной сцены.



Простая трёхмерная сцена

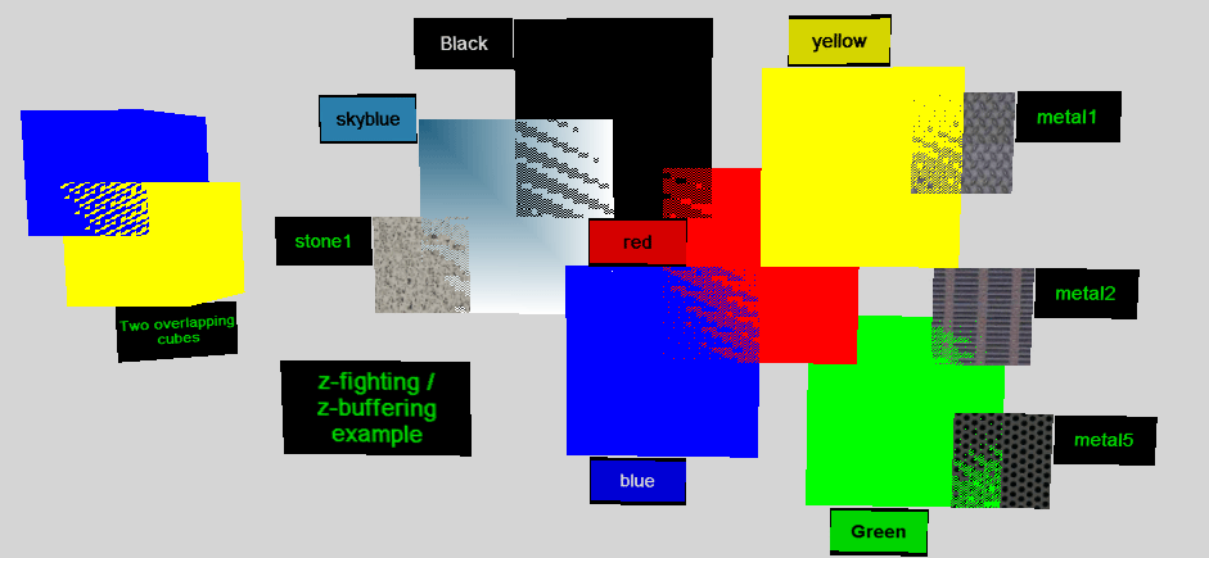
Представление в Z-буфере

Алгоритм буффера глубины очень эффективен и практически не имеет недостатков, если реализуется аппаратно. Программно же существуют другие методы, способные конкурировать с ним: Z-сортировка («алгоритм художника») и двоичное разбиение пространства (BSP), но они также имеют свои достоинства и недостатки. **Основной недостаток** Z-буферизации состоит в потреблении большого объёма памяти: в работе используется так называемый буфер глубины или Z-буфер.

Разрядность буфера глубины оказывает сильное влияние на качество визуализации: использование 16-битного буфера может привести к геометрическим искажениям, например, эффекту «борьбы», если два объекта находятся близко друг к другу. 24, 32-разрядные буферы хорошо справляются со своей задачей. 8-битные почти никогда не используются из-за низкой точности.

Z-Fight

Если два объекта имеют близкую Z-координату, иногда, в зависимости от точки обзора, показывается то один, то другой, то оба полосатым узором. Это называется Z-конфликт (англ. Z fighting). Чаще всего конфликты присущи спецэффектам (декалям), накладывающимся на основную текстуру, например, дырам от пуль.



Решаются Z-конфликты сдвигом по Z (по видимой плоскости глубины) одного объекта относительно другого на величину, превышающую погрешность Z-буфера. То есть нужно немного сдвинуть один объект ближе после выполнения всех трансформаций.
В OpenGL это происходит с помощью функции

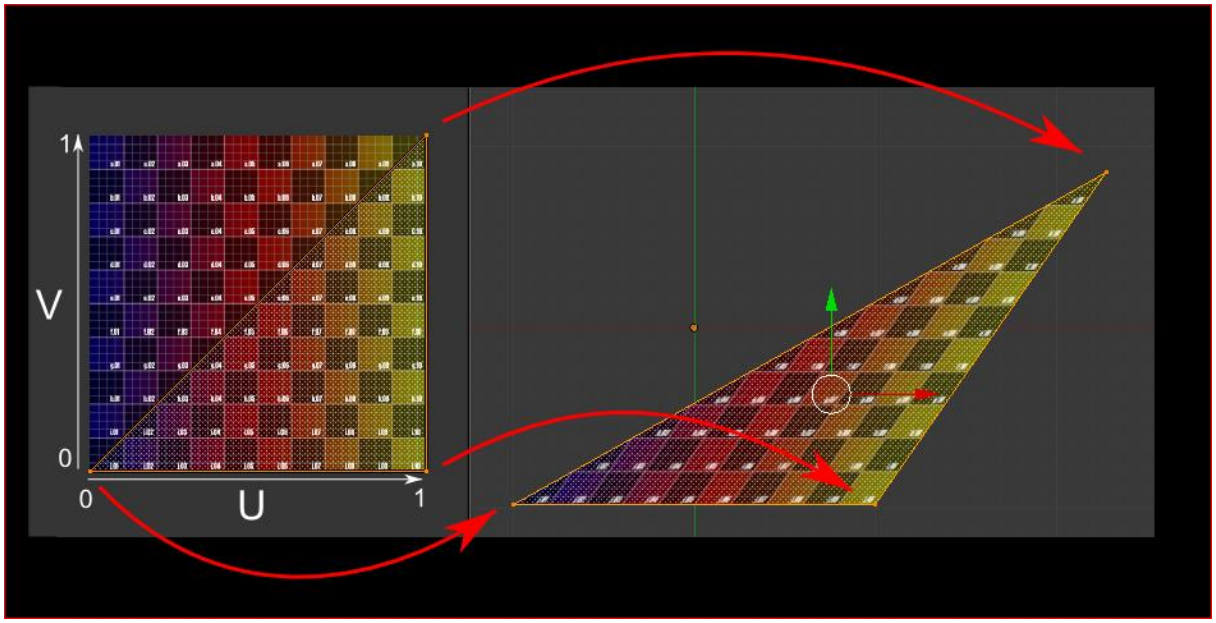
```
void glPolygonOffset( GLfloat factor, GLfloat units);
```

Текстурирование

UV координаты

При текстурировании меша, нужно каким-то образом объяснить OpenGL какая часть текстуры будет использоваться для заливки каждого из треугольников. Это делается с помощью UV координат.

Каждая вершина может иметь не только позицию, а еще цвет и еще, например, два float значения – координаты U и V. Эти координаты используются для доступа к текстуре следующим образом:



Важной особенностью является то, что при работе с текстурами независимо от их размера текстурные координаты должны быть **от 0 до 1.**

Возможен вариант также с повторением текстуры, задать режим можно с помощью функции

Отброс пикселей

Так же при отрисовке какого-то меша с текстурой имеется возможность отбрасывать определенные пиксели по определенным условиям. Часто используется это при отрисовке травы в 3х-мерных играх.

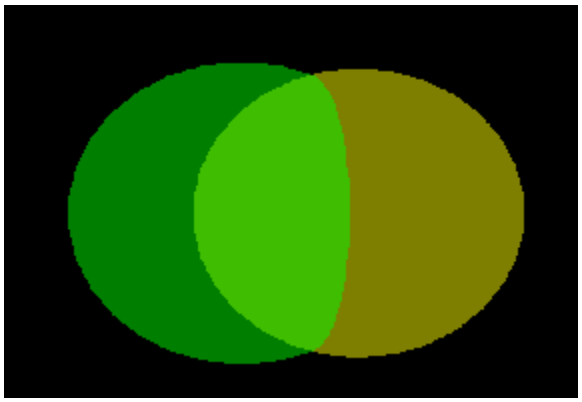
Когда рисуется трава 3хмерных играх, то рисуется просто текстурированный прямоугольник с тектурой травы, который всегда обращен в камеру. Однако при включеном тесте глубины складываться такие прямоугольники будут некрасиво. Однако, если отбрасывать пксели, который имеют значаение прозрачности меньше определенного значения, то не будет происходить отрисовка и запись в буффер глубины отброшеных пикселей.

Делается это очень просто в пиксельном шейдере с помощью ключевого слова `discard`.

```
if(color.alpha < 0.1){
 discard;
}
```

Смешивание цветов

Для создания полупрозрачных объектов используется смешивание. Ведь если объект является полупрозрачным, то он не полностью закрывает объекты, находящиеся за ним, как будто вы смотрите через стекло или мутное стекло. Значение прозрачности в этом случае отвечает за то, насколько через стекло хорошо видно.
Рассмотрим на примере, у нас есть 2 объекта: один жёлтый, другой зелёный. Причём объекты являются полупрозрачными. Смешанный цвет получается путём смешивания жёлтого и зелёного цвета.



Результирующий цвет смешивания рассчитывается по формуле:

```
resultColor = frontColor*srcAlpha + backColor*(1 - srcAlpha)
```

`backColor` - цвет фона, на который накладывается зелёный круг, то есть это жёлтый цвет (1.0, 1.0, 0.0).
`frontColor` - цвет круга, который накладывается, то есть это зелёный (0.0, 1.0, 0.0)
`srcAlpha` - прозрачность круга, который накладывается, то есть 0.5.
`backColor` - результирующий цвет (0.5, 1.0, 0.0).

Для использования смешивания его необходимо включить, используя функцию:

`glEnable(GL_BLEND);`
По умолчанию смешивание выключено.
Для задания коэффициентов функции прозрачности обычно используются функции:
`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);` - данное задание режима как раз соответствует формуле, которая указана сверху.
По умолчанию это значение `GL_ONE, GL_ONE`.

Освещение

Нормали Треугольников

Нормаль к плоскости — это единичный вектор который направлен перпендикулярно к этой плоскости.

Нормаль к треугольнику — это единичный вектор направленный перпендикулярно к треугольнику. Нормаль очень просто рассчитывается с помощью векторного произведения двух сторон треугольника(если вы помните, векторное произведение двух векторов дает нам перпендикулярный вектор к обоим) и нормализованный: его длина устанавливается в единицу.
Вот псевдокод вычисления нормали:

```
треугольник( v1, v2, v3 )
сторона1 = v2-v1
сторона2 = v3-v1
нормаль = вектПроизведение(сторона1, сторона2).нормализировать()
```

Вершинная Нормаль

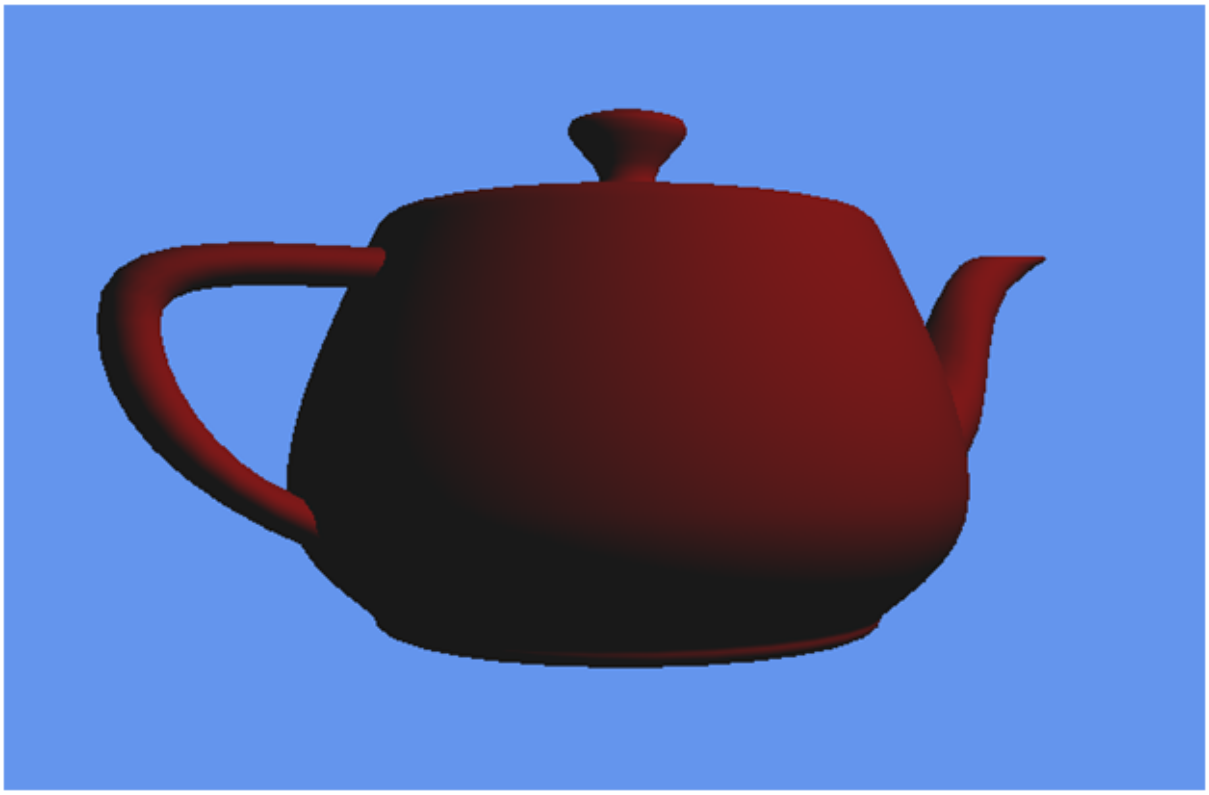
Это нормаль введенная для удобства вычислений. Это комбинированная нормаль от нормалей окружающих данную вершину треугольников. Это очень удобно, так как в вершинных шейдерах мы имеем дело с вершинами, а не с треугольниками. В любом случае в OpenGL у мы почти никогда и не имеем дела с треугольниками.

Использование нормалей вершин в OpenGL

Использовать нормали в OpenGL очень просто. Нормаль — это просто атрибут вершины, точно так же, как и позиция, цвет или UV координаты...Тоесть ничего нового учить не придется…

Модель освещения Ламберта

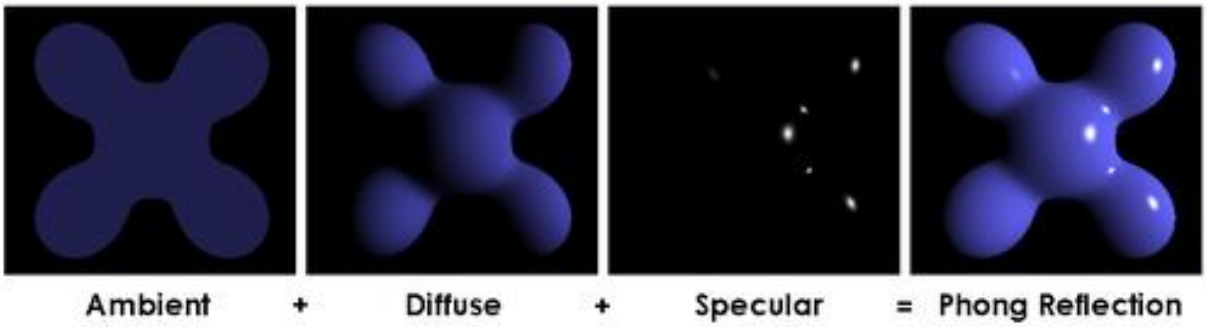
Простейшая модель освещения - чисто диффузное освещение. Считается, что свет падающий в точку, одинакового рассеивается по всем направлением полупространства. Таким образом, освещенность в точке определяется только плотностью света в точке поверхности, а она линейно зависит от косинуса угла падения света к отдельной вершине.



Модель освещения по Фонгу

Основной смысл модели в том, что итоговое освещение объекта складывается из трех компонентов, так же данная модель включает в себя освещение по Ламберту в качестве диффузной составляющей:

- Фоновой свет (ambient)
- Рассеянный свет (diffuse)
- Отраженный свет (specular)



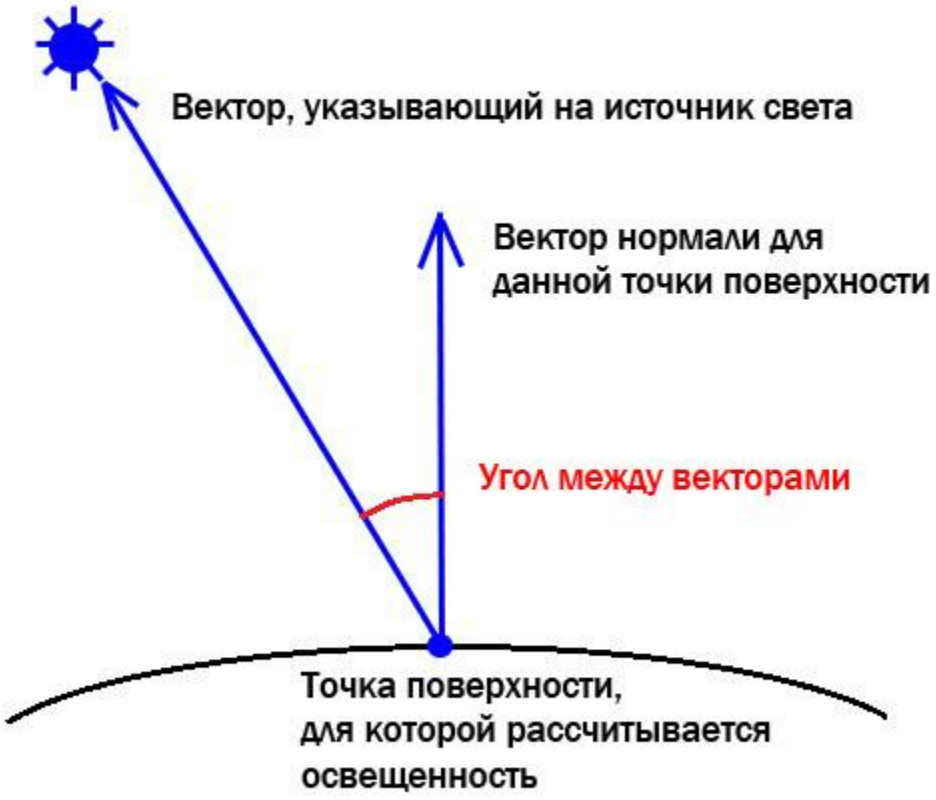
Фоновое (ambient) освещение.

Фоновое освещение освещает объекты одинаково со всех сторон. Оно не зависит от положения источника света и глаза наблюдателя. Задается константой.

Диффузное (diffuse) или рассеянное освещение.

Яркость объекта, освещенного диффузным светом, зависит от положения объекта и от положения источника света. Диффузный свет отражается от поверхности одинаково во все стороны. Поэтому положение глаза наблюдателя на диффузное освещение не влияет. Яркость диффузного освещения определяют по фактору Ламберта. Вычисляется косинус угла между вектором нормали и вектором, указывающим из точки на источник света. Чем этот угол меньше, тем ярче освещена точка. Если **угол = 0**, получаем

максимальную яркость. Если **угол=90** градусов - яркость будет равна нулю.



Косинус угла между векторами равен **скалярному произведению** двух векторов единичной длины. Для калярного произведения существует специальная функция библиотеки GLM + функция доступна в шейдерах - **dot.** В фрагментном шейдере у нас уже есть нормализованный вектор нормали для данного пикселя:

```
vNormalViewSpace = normalize(vec3(uNormalMat * normalVec4));
     // так как это направление, учитывается поворот только
```

Также во фрагментном шейдере мы имеем интерполированное для каждого пикселя значение координат точки поверхности `vPosViewSpace` в координатах камеры.

```
varying vec3 vPosViewSpace;
```

Передадим координаты источника света как униформу во фрагментный шейдер (в координатном пространстве камеры):

```
uniform vec3 uLightPosViewSpace;
```

Теперь мы можем вычислить вектор из точки поверхности на источник света и нормализовать его:

```
vec3 fromTexelToLightDir = normalize(uLightPosViewSpace -
                        vPosViewSpace);
```

Вычислим скалярное произведение вектора нормали `vNormalViewSpace` и вектора `fromTexelToLightDir`:

```
float diffusePower = diffuseCoef * max(dot(vNormalViewSpace,
                fromTexelToLightDir), 0.0);
```

В общем случае скалярное произведение может быть отрицательным, если угол между векторами больше 90 градусов. Отрицательные значения нам нужно отсечь.

Умножим полученное значение на некоторый коэффициент диффузного освещения k_diffuse и получим яркость диффузного освещения пикселя

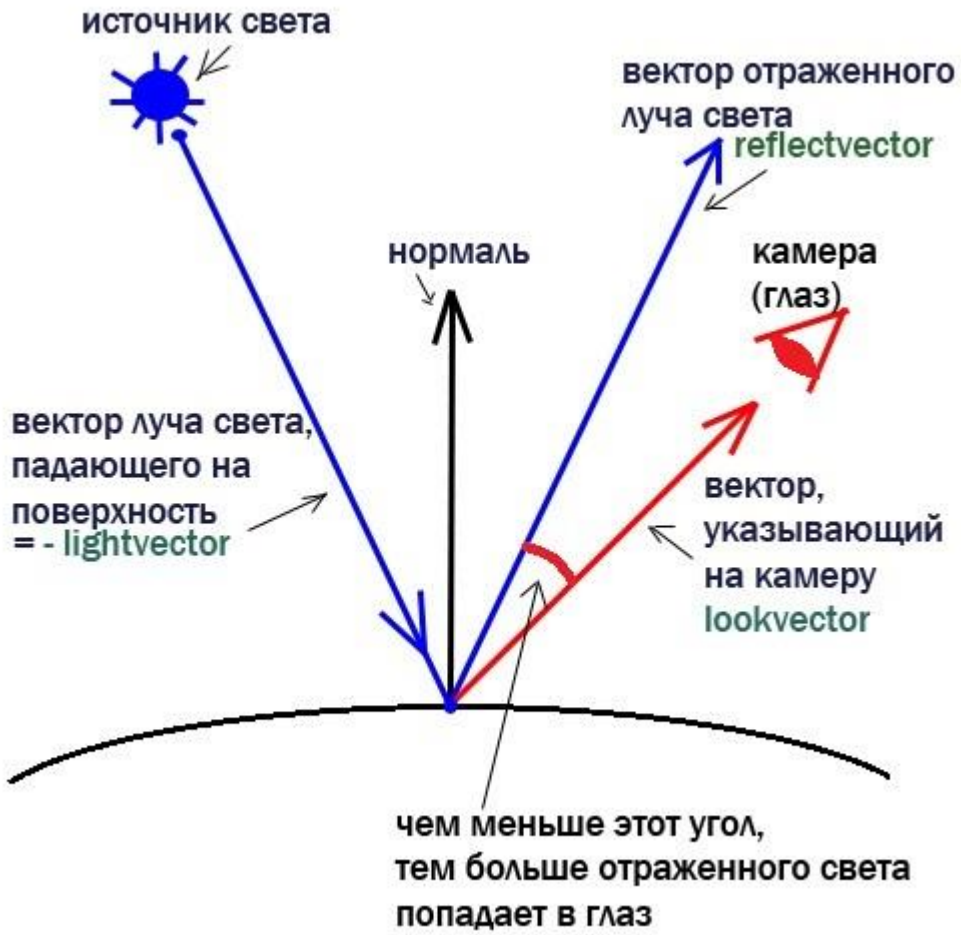
Кстати, число с плавающей точкой в шейдерах нужно указывать как 0.0. Если указать просто 0, то будет ошибка и шейдер не скомпилируется.

Зеркальное (specular) или бликовое освещение.

Диффузное освещение не зависит от положения глаза наблюдателя (камеры). Зеркальное освещение определяется долей отраженной световой энергии попавшей в камеру.

Поэтому яркость зеркального освещения зависит не только от положения источника света, но также и от положения камеры.

Вычисляем вектор отраженного луча света от точки, далее находим косинус угла между отраженным вектором и направлением на камеру. Чем меньше угол, тем больше косинус и тем больше света попадет в камеру. Максимальная яркость достигается при угле равном нулю, минимальная при угле 90 градусов. Смотрите на рисунок:



При расчете диффузного освещения мы определили вектор единичной длины, проходящий из освещаемой точки к источнику света и назвали его `fromTexelToLightDir`. Очевидно, что вектор падающего луча света нужно провести от источника света к точке на поверхности, т.е. просто поменять знак на минус. Для вычисления отраженного вектора в GLSL существует специальная функция reflect:

```
vec3 texelLightReflectionDir = normalize(reflect(-fromTexelToLightDir,
                                vNormalViewSpace));
```

Теперь вычислим вектор, указывающий из точки освещения на камеру и нормализуем его, в координатном пространстве камеры - это просто отрицательный вектор направления к пикселю:

```
vec3 fromTexelToEyesDir = normalize(-vPosViewSpace);
```

Далее нам нужно вычислить косинус угла между отраженным вектором и направлением на камеру. Это скалярное произведение двух единичных векторов:

```
float specularDot = max(dot(texelLightReflectionDir,
                    fromTexelToEyesDir), 0.0);
```

Отсекаем отрицательные значения скалярного произведения при помощи функции max.

При отражении света на поверхности появляются блики. Размер блика можно регулировать при помощи параметра блеска. Вычисленное значение скалярного произведения нужно возвести в степень блеска. Для возведения в степень в GLSL предусмотрена функция `pow` Обычно значение блеска выбирают в размере несколько десятков. При увеличении блеска размер блика уменьшается, но яркость его увеличивается. И наоборот, чем меньше блеск, тем больше размер блика, но яркость его становится меньше. Пусть, для примера, блеск будет равен 40. Возведем полученное скалярное произведение в степень 40:

```
float specularPower = 0.0;
// проверка, так как 0 в любой степени - это 1.0
if(specularDot > 0.0){
   specularPower = pow(specularDot, specularShinnes);
   specularPower = clamp(specularPower, 0.0, 1.0);
}
```

Для того, чтобы получить цвет пикселя с учетом освещения нужно сложить фоновую диффузную и зеркальную части освещения `ambientCoef + diffusePower + specularPower`

```
float lightPower = ambientCoef + diffusePower + specularPower;
```

http://andmonahov.blogspot.ru/2012/10/opengl-es-20.html
http://eax.me/opengl-lighting/
http://triplepointfive.github.io/ogltutor/

Создание теней в трехмерном пространстве

Тени - вероятно, самый важный визуальный эффект, влияющий на реалистичность картинки, генерируемой 3D-движком. Тени помогают ощутить глубину сцены и оценить взаимное расположение объектов. Без теней трехмерное изображение кажется плоским и не натуральным.

Для статической геометрии проблема генерации теней решена давно: начиная с Quake, 3D-игры реализуют статическое освещение, как правило, основанное на картах освещения (*будет рассказано далее*). Этот подход дает реалистичные, мягкие тени, но не для динамических объектов. Существует довольно много методов рендеринга теней в реальном времени, каждый имеет свои преимущества и недостатки. Выбор алгоритма зависит от нужд и специфики графического приложения.

Карты освещения (Lightmaps)

В том или ином виде карты освещения используются практически во всех современных 3D-приложениях. Этот метод применяется для создания не только теней, но и всего освещения сцены. Освещение генерируется для статической геометрии до начала цикла рендеринга, и во время рендеринга в основном не изменяется.

Карты освещения - это небольшие текстуры, которые генерируются на этапе создания сцены. Почти всегда карты освещения выравниваются с обычными текстурами полигонов, и каждый пиксель карты соответствует 4-32 пикселям текстуры. Размеры карты определяются размерами минимального, ограничивающего полигон прямоугольника, стороны которого параллельны текстурным векторам. (См. рисунок)

При создании сцены в редакторе карта заполняется черными пикселями. Далее, для каждого пикселя карты освещения находятся трехмерные координаты точки на полигоне. Для этой точки необходимо построить список всех источников света, которые влияют на ее освещение: вектора из данной точки до источников света проверяются на пересечение с геометрией уровня, и если пересечение имеет место - то этот источник света не освещает точку (относительно него точка в тени). Остальные источники увеличивают значение пикселя лайтмэпа на величину, зависящую от используемой модели освещения и положения источника света относительно точки. В целях улучшения внешнего вида картинки, к картам освещения часто применяется билинейная фильтрация. Эти операции повторяются для каждого освещаемого полигона сцены.

Во время рендеринга, карты освещения могут накладываться вторым проходом, с использованием альфа-блендинга.

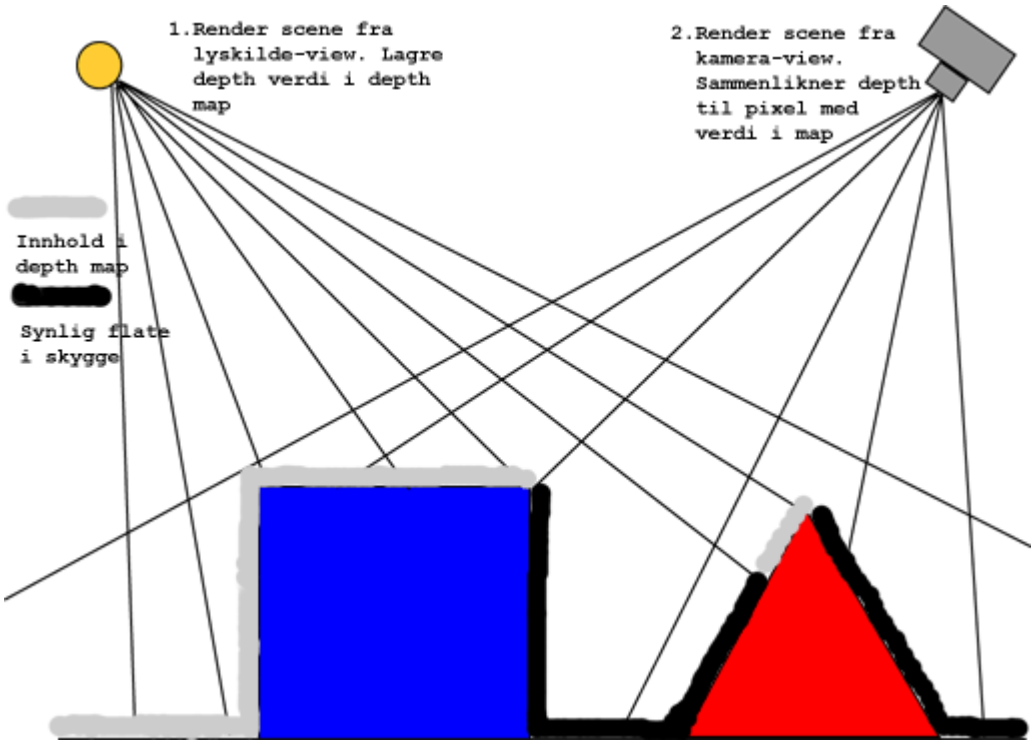
Алгоритм работает только со статической геометрией, карты освещения занимают довольно большой объем памяти, но во время рендеринга алгоритм чрезвычайно эффективен.

Shadow mapping

Тень является очень важной частью для восприятия трехмерной сцены. Один из самых распространенных алгоритмов создания динамических теней - карта теней.

Когда дело доходит до отрисовки главный вопрос, который нужно вычислить - находится ли пиксель в тени? Если переформулировать вопрос, то звучать будет он следующим образом - будет ли луч света, идущей из источника до пикселя, встречать на своем пути препятствия или нет? Если да - вероятно, что пиксель в тени (предполагая, что объект не прозрачный…), и если нет - пиксель не в тени.

Если переместить камеру в позицию света и нарисовать сцену, то можно с помощью буффера глубины определить расстояние до конкретного пикселя и узнать какие пиксели ближе всего к источнику света. Тогда можно сказать, что пиксель, который не прошел тест глубины находится в тени. Только пиксели, которые проходят тест глубины - оказываются на свету. Они будут единственными, кто получит свет, поскольку их ничто не закрывает. Вот идея карты теней в 2 словах.



Таким образом глубины поможет нам определить, находится ли пиксель в тени или нет, но не все так просто. Камера и свет обычно находятся в разных точках, а тест глубины обычно используется для решения проблем отображения в позиции камеры.

Решение - рендерить сцену 2 раза. Первый с позиции света, причем итог рендера не пойдет в буфер цвета. Вместо этого значение глубины ближайшего пикселя рендерится в отдельный буфер глубины (вместо того, который автоматически создается при рисовании). Во втором проходе сцена рендерится как обычно, в позиции камеры. Буфер глубины, который мы сами создали, привязывается к фрагментному шейдеру. Для каждого пикселя мы получаем соответствующее значение глубины из буфера. Мы уже подсчитали глубину для этого пикселя с позиции света. Временами эти значения глубины равны. Это тот случай, когда пиксель был ближе к свету, поэтому значение его глубины

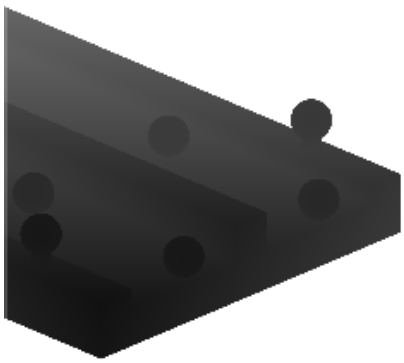
попало в буфер. Тогда мы считаем, что пиксель на свету, а значит его цвет находится как обычно. Если значения глубины различны (вычисленное значение меньше, чем в буффере), то значит, что другой пиксель перекрыл наш во время первого рендера. Тогда мы учтем этот факт во время нахождения цвета добавив эффект затенения.

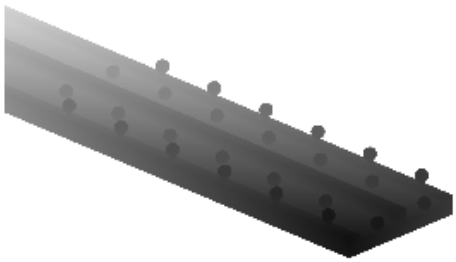
Преимущества и недостатки ShadowMapping

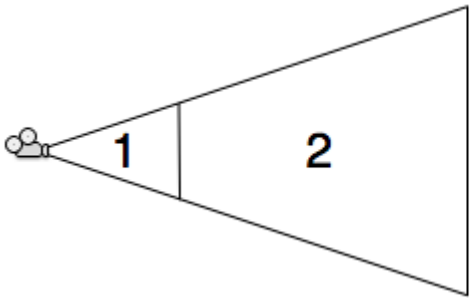
Является достаточно простым и быстрым алгоритмом, который реализуется достаточно легко. К недостаткам алгоритма можно отнести высокое потребление памяти при необходимости обеспечить высокое качество теней (так как необходимо создавать достаточно большую текстуру для карты теней).

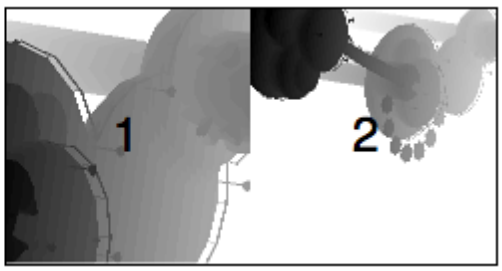
Parallel Split Shadow Mapping (Cascade shadow mapping)

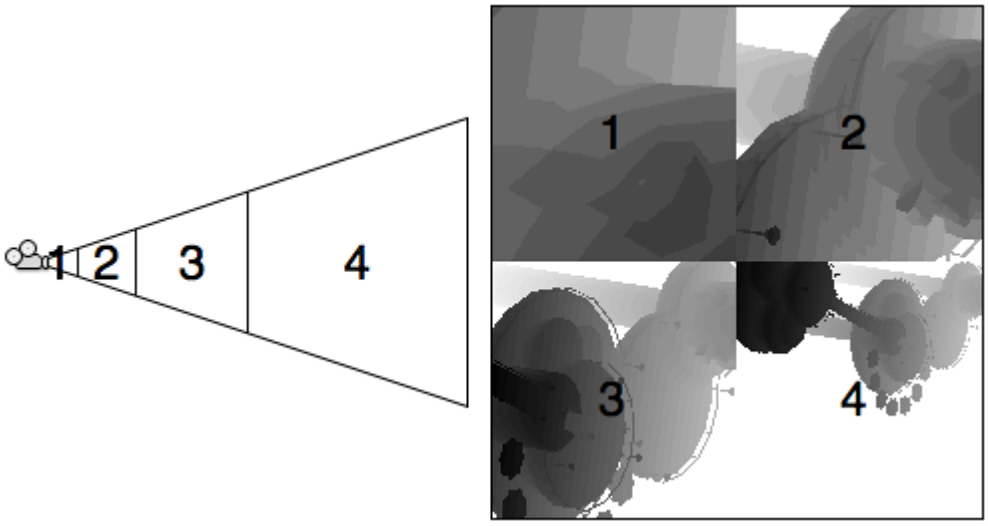
Для того, чтобы избавиться от выделения больших текстур под теневые карты используются различные вариации алгоритма, например **Parallel Split Shadow Mapping (Cascade shadow mapping)**. Идея данного алгоритма заключается в том, что создается несколько буфферов для карты теней, затем происходит рендеринг сцены в каждую из текстур, но с разным расстоянием до источника света. Затем, при отрисовке трехмерной сцены - вблизи игрока используется карта теней, отрендереная с самого близкого расстояния (тем самым достигается высокое качество теней вблизи), а при рендеринге отдаленных объектов используется карта теней, отрендеренная с самого большого расстояния (как результат - в дали нам не очень важно высокое качество теней).






1
2


1
2


1
2
3
4
1
2
3
4


Powered by
TV3D

Другие реализации алгоритмов
http://www.dailytelefrag.com/articles/print.php?id=19

Реализация теней на OpenGL

При отрисовке с использованием OpenGL, отрисовка происходит в определенный буффер кадра. Буффер кадра раскладывается на буфер цвета (который отображается на экран) и буфер глубины (а еще на несколько для дополнительных буфферов при необходимости).

Когда вызывается инициализация окна отрисовки в коде, то создается стандартный буфер кадра с указанными параметрами инициализации. Он управляется оконной системой и не может быть удален OpenGL.

Но приложение может создать отдельный, свой собственный буфер кадра. Он может быть использован для различных методов под управлением приложения.

OpenGL позволяет создавать FBO (буффер кадра), который инкапсулирует внутреннее состояние буфера кадра.

Однажды создав и правильно настроив мы можем менять буфер просто привязав другой. Однако только стандартный буфер может быть использован для отображения чего-либо на экран. Буфер кадра, созданный приложением может быть использован только для ”рендера помимо экрана“. Это может быть промежуточный рендер (как у нас для карты теней), результат которого будет использован для ”настоящего" рендера, который попадет на экран.

Сам по себе буфер кадра легко заполнить. Чтобы его можно было использовать, мы должны прикрепить одну или несколько текстур. Текстура содержит пространство буфера кадра. OpenGL определяет следующие точки крепления:

- **COLOR_ATTACHMENT<i>** - текстура, которая будет прикреплена сюда, будет получать цвет, который выходит из фрагментного шейдера. Окончание ‘i’ означает, что может быть прикреплено сразу несколько текстур; у фрагментного шейдера есть

механизм, который позволяет рендерить сразу в несколько буферов цвета одновременно (Относится к современным версиям OpenGL - 3.0+).

- **DEPTH_ATTACHMENT** - эта текстура будет получать результат теста глубины.
- **STENCIL_ATTACHMENT** - текстура будет использована в качестве трафарета (стенсила). Он ограничивает область растеризации и может быть использован во многих ситуациях.
- **DEPTH_STENCIL_ATTACHMENT** - просто комбинация из двух предыдущих так как они довольно часто используются вместе.

Для карты теней потребуется только буфер глубины. Так мы создаем FBO. Аналогично текстурам и буферам мы указываем адрес массива типа GLuints и его размер.

glGenFramebuffers(1, &m_fbo)

Далее мы создаем текстуру, которая будет служить в качестве карты теней. В целом это обычная 2D текстура с некоторыми отличиями, что бы она подходила для нашей цели:

- Внутренний формат - GL_DEPTH_COMPONENT. Это отличается от предыдущих использовании этой функции, в которой формат всегда был одним из типов цвета (например GL_RGB). GL_DEPTH_COMPONENT представляет единственное вещественное число, обозначающее глубину.
- Последний параметр glTexImage2D - 0. Это значит, что мы не поставляем данных для инициализации буфера. Это имеет смысл, зная что буфер будет хранить значение глубины каждого кадра, и каждый кадр будет немного отличаться. Когда бы мы не начали новый кадр, мы будем использовать glClear() для очистки нашего буфера. Вот вся инициализация в данной ситуации.
- Мы сообщаем OpenGL, что в случае выхода координат текстуры за пределы, мы их сжимаем до отрезка [0,1]. Это может произойти в случае, когда окно проекции с позиции камеры вмещает больше, чем окно в позиции света. Для избежания артефактов, таких как влияния тени саму на себя, мы сжимаем координаты текстуры.

```
glGenTextures(1, &m_shadowMap);
glBindTexture(GL_TEXTURE_2D, m_shadowMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, WindowWidth,
     WindowHeight, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

После того, как создали FBO, текстуру и настроили ее так, что бы она подходила для карты теней. Теперь нам требуется прикрепить текстуру к FBO.
Первое, что мы должны сделать, это привязать FBO. Это сделает его “текущим” и все последующие операции над FBO будут применены к нему. Эта функция принимает указатель на FBO и желаемую цель.

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, m_fbo);
```

Она может быть **GL_DRAW_FRAMEBUFFER** или **GL_READ_FRAMEBUFFER**. Мы используем последний когда хотим считать с буфера кадра через glReadPixels (не в этом уроке). Так как мы хотим рендерить в буфер, то мы выбираем **GL_DRAW_FRAMEBUFFER**.

```
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_TEXTURE_2D, m_shadowMap, 0);
```

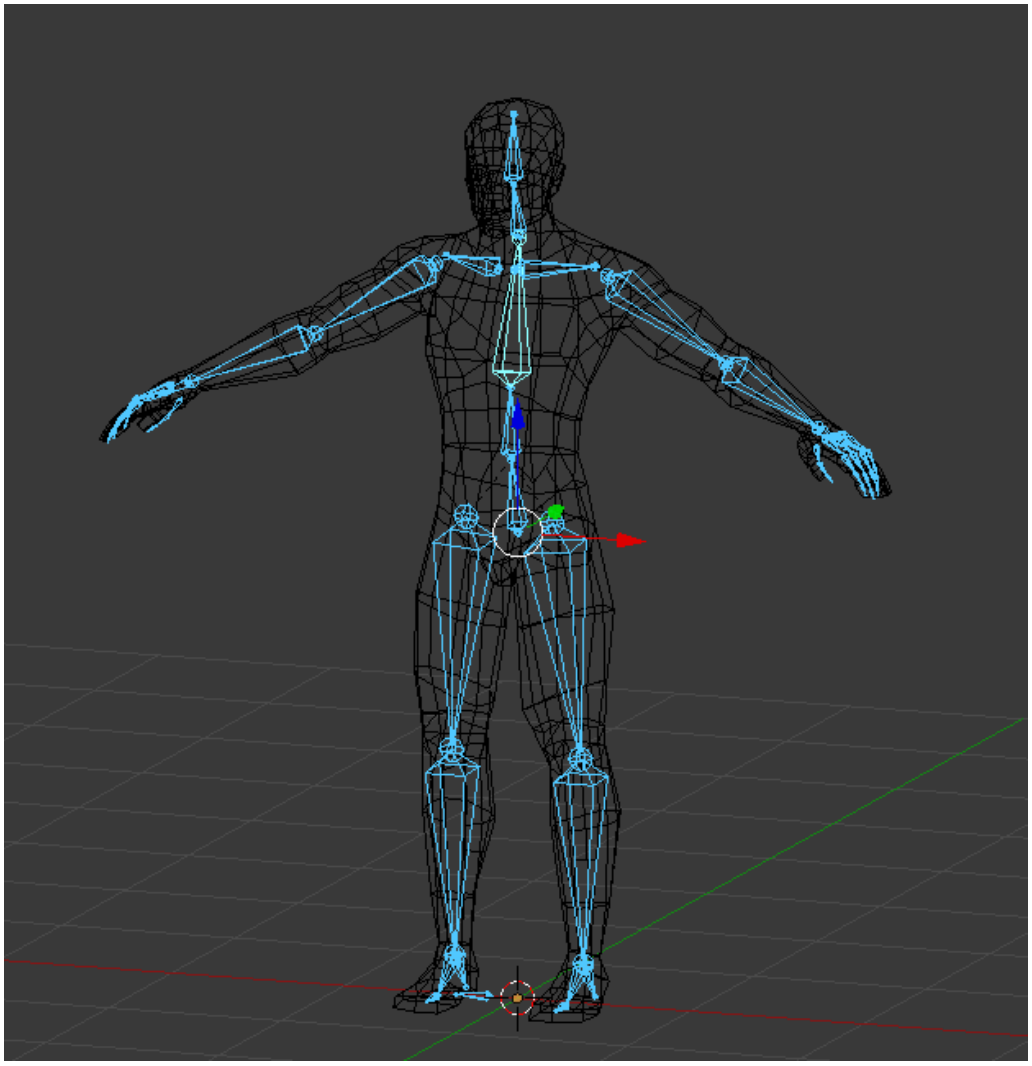
Здесь мы прикрепляем текстуру для карты теней к FBO. Последний параметр указывает на то, какой слой мипмапа использовать. Мипмап - это характеристика отображения текстуры, показывающая различные разрешения, начиная с наивысшего с мипмап равным 0 и уменьшая до 1-N. Комбинация мипмапов текстур - трехлинейный фильтр, дающий наилучший результат через комбинацию текселей с соседних уровней мипмапа (когда не один не подходит полностью). Здесь мы используем только 1 мипмап, поэтому ставим 0. Мы даем указатель на карту теней как 4 параметр.

```
glDrawBuffer(GL_NONE);
```

Продолжение описания тут
https://triplepointfive.github.io/ogltutor/tutorials/tutorial23.html

Еще темы для лекций
https://triplepointfive.github.io/ogltutor/

Скелетная анимация



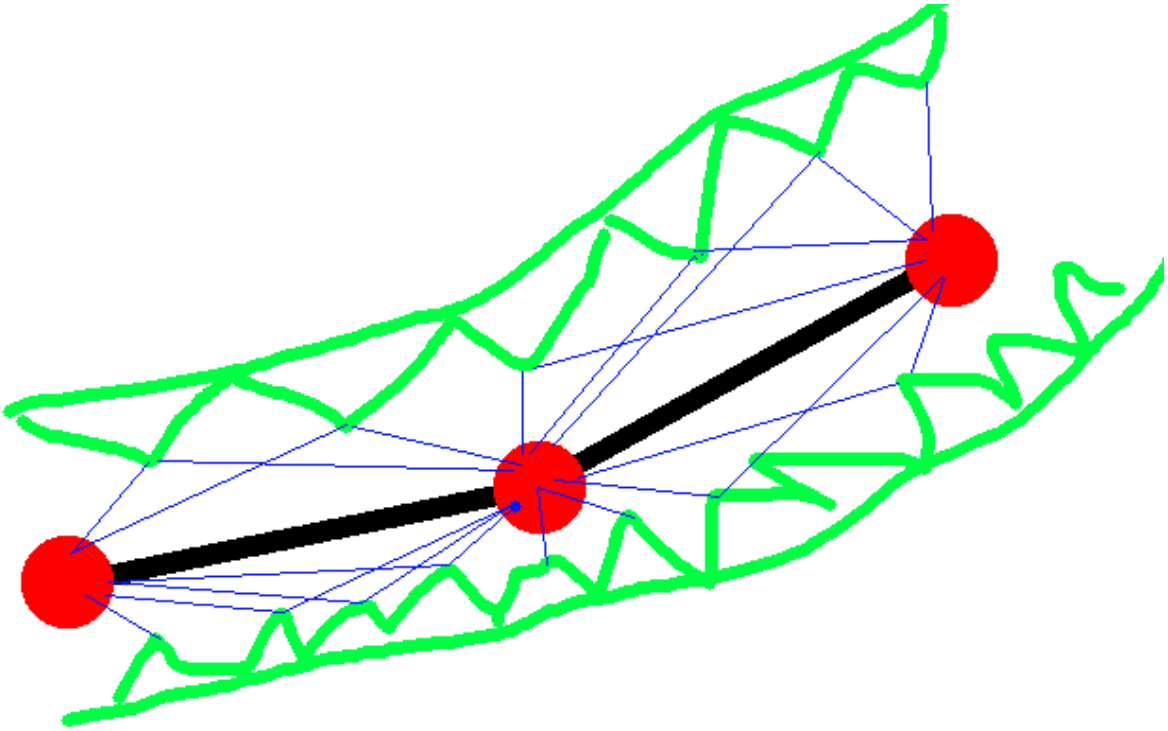
Заключается в том, что 3D моделер создаёт скелет, представляющий собой как правило древообразную структуру костей, в которой каждая последующая кость «привязана» к предыдущей, то есть повторяет за ней движения и повороты с учётом иерархии в скелете. Далее каждая вершина модели «привязывается» к какой-либо кости скелета (либо к нескольким, в разной пропорции). Таким образом, при движении отдельной кости двигаются с определенным коэффициентом и все вершины, привязанные к ней.

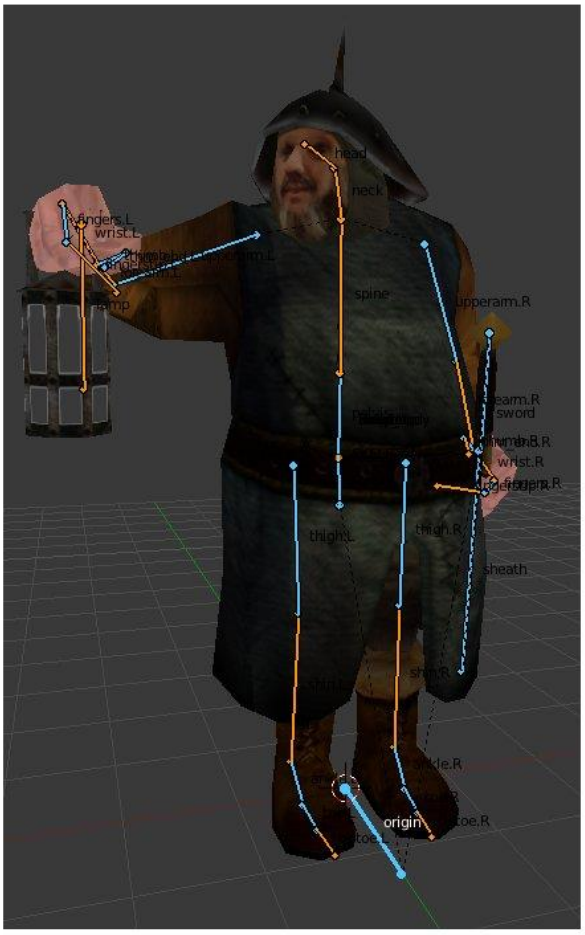
Благодаря этому задача 3D моделера сильно упрощается, потому что отпадает необходимость анимировать отдельно каждую вершину модели, а достаточно лишь задавать положение и поворот костей скелета.

Скелетная анимация состоит из 2 частей.
Первую выполняет человек, который моделирует анимацию, а вторую - программист (точнее код, который он пишет).

Первая часть создается в ПО для моделирования и называется Риггинг (Rigging). В этом этапе 3D моделер создает скелет из костей внутри меша. Меш в данном случае служит “кожей” объекта (будь это человек, монстр или кто-то еще), а кости будут использоваться для движения меша таким образом, чтобы происходила имитация движения в реальном мире. Для этого каждая вершина привязывается к одной или более костей. Когда вершина присоединена устанавливается вес, который задает силу влияния кости на вершину. Хорошей практикой является установка суммарного веса для вершины

равным 1. Например, если вершина расположена между 2 костями, то вероятно, что нужно разделить вес по 0.5 между отдельными “костями”, потому что ожидается одинаковое воздействие на вершину. Хотя, если вершина полностью во влиянии 1 кости, то вес будет равен 1 (что означает полный контроль над движением вершины).





Часто используется иерархическая структура костей для скелетной анимации. Это значит, что кости имеют потомков / родителей тем самым создавая дерево костей. Каждая кость имеет родителя, кроме корневой кости. В случае, например, человеческого тела хорошим выбором будет позвоничник, у которого дети ноги и плечи, а пальцы еще на уровень ниже. Когда движется родительская кость, то движутся и потомки, но если движется потомок, то на родителя это не влияет (мы можем двигать пальцами свободно от руки, но при движении руки пальцы следуют за ней). С практической точки зрения это значит, что когда мы хотим переместить кость, то нам требуется скомбинировать преобразования и для всех родительских костей, которые ведут от корневой кости.

Реализация костевой анимации в OpenGL

Чаще всего кости представляют собой обычные матрицы трансформации, которые передаются в процессе анимации перемножаются в соответствии с иерархией дерева костей, затем передаются в качестве “Юниформа” в вершинный шейдер в виде массива.

Также в качестве аттрибута каждой отдельной вершины добавляются индексы массива костей и коэффициенты влияния кости на трансформацию вершины (как правило, не более четырех костей влияют на каждую отдельную вершину модели).

В процессе анимации каждый раз происходит пересчет матриц трансформации костей и последующая отрисовка модели.

Принцип работы OpenGL на низком уровне

Алгоритм Бразенхема для построения линий линий

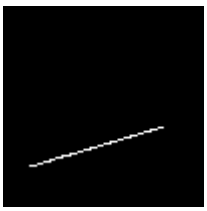
Алгоритм Брезенхема - это алгоритм, с помощью которого можно определить, какие точки нужно закрасить на экране либо в изображении, чтобы получить изображение прямой линии между двумя заданными точками.

Для рисования прямых отрезков на плоскости с использованием алгоритма Брезенхема запишем уравнение прямой в общем виде

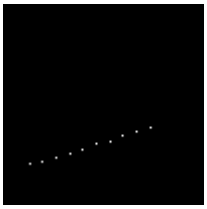
$$y = kx + b$$

Можно непосредственно попробовать интерпретировать данную формулу при рисовании линий, тогда код будет выглядеть следующим образом

```
void line(int x0, int y0, int x1, int y1, ImageBuffer &image, Color color){
    for (float t = 0.0; t < 1.0; t += 0.01) {
        int x = x0*(1.0-t) + x1*t;
        int y = y0*(1.0-t) + y1*t;
        image.set(x, y, color);
    }
}
```



Однако данный подход имеет определенные недостатки, а именно его невысокая скорость и сложность выбора константы шага `t = 0.01`. Если взять значение этой константы меньше с целью увеличения производительности, то на некоторых отрезках будут появляться дыры.

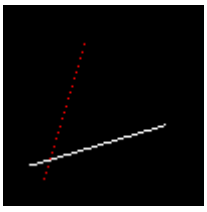


Логично предположить, что шаг нужно вычислить на основании количества пикселей, которые надо нарисовать.

```
void line(int x0, int y0, int x1, int y1, ImageBuffer &image, Color color){
    for (int x=x0; x<=x1; x++) {
        float t = (x-x0)/(float)(x1-x0);
        int y = y0*(1.0-t) + y1*t;
        image.set(x, y, color);
    }
}
```

Однако проблема данного кода в том, что чем вертикальнее наша линия, то тем хуже она прорисовывается, **например**:

```
line(13, 20, 80, 40, image, white);
line(20, 13, 40, 80, image, red);
line(80, 40, 13, 20, image, red);
```



Выяснится, что одна линия хороша, вторая с дырками, а третьей вовсе нет. Решением данной проблемы является использование цикла по значению Y если линия является больше вертикальной, чем горизонтальной. Это определяемся в помощью длины проекций на оси.

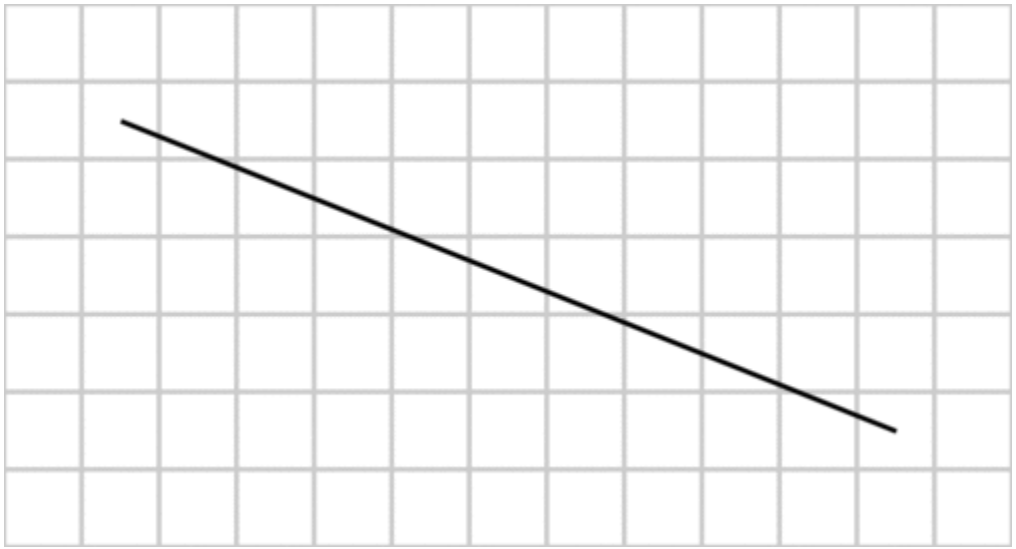
```
void line(int x0, int y0, int x1, int y1, ImageBuffer &image, Color color){
    bool isVertical = false;
    // Определяем какая из проекций на оси длинее
    if (std::abs(x0-x1) < std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        isVertical = true;
    }
    // Корректируем значения X, чтобы X0 был меньше X0
    if (x0 > x1) { // make it left-to-right
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    // цикл заполнения пикселей
    for (int x=x0; x<=x1; x++) {
        float t = (x-x0)/(float)(x1-x0);
        int y = y0*(1.0-t) + y1*t;
        if (isVertical) {
            image.set(y, x, color);
```

```
        } else {
            image.set(x, y, color);
        }
    }
}
```

Однако данный код все равно не идеален. Его можно оптимизировать вынеся вычисление шага за цикл, а также можно оптимизировать заполнение пикселей. Можно ввести переменную `error`, которая даёт нам дистанцию до идеальной прямой от нашего текущего пикселя `(x, y)`. Каждый раз, как `error` превышает один пиксель, мы увеличиваем (уменьшаем) y на единицу, и на единицу же уменьшаем ошибку.

Финальный код для рисования линии выглядит следующим образом

```
void line(int x0, int y0, int x1, int y1, ImageBuffer &image, Color color){
    bool isVertical = false;
    // Определяем какая из проекций на оси длинее
    if (std::abs(x0-x1)<std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        isVertical = true;
    }
    //  Корректируем значения X, чтобы X0 был меньше X1
    if (x0>x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    // вычисляем шаг
    int dx = x1-x0;
    int dy = y1-y0;
    // вычисление погрешности каждого шага
    float derror = std::abs(dy/float(dx));
    float error = 0;
    // Заполнение пикселей
    int y = y0;
    for (int x=x0; x<=x1; x++) {
        if (isVertical) {
            image.set(y, x, color);
        } else {
            image.set(x, y, color);
        }
        // суммирование ошибки
        error += derror;

        if (error > 0.5) {
            y += ((y1>y0) ? 1 : -1);
            error -= 1.0;
        }
    }
}
```



Алгоритм рисования треугольника

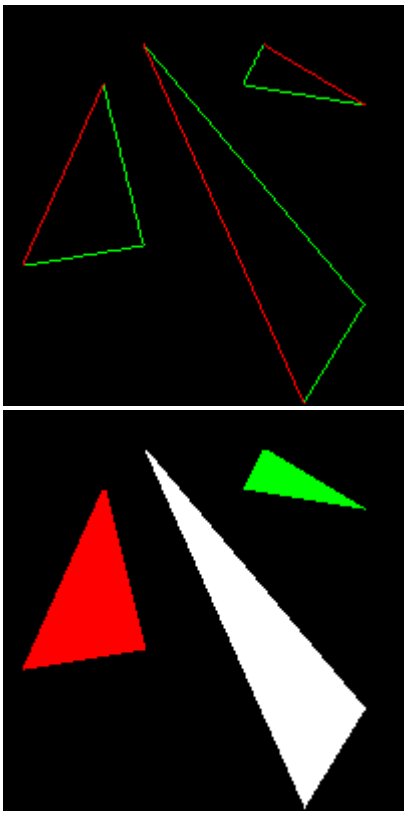
Для отрисовки треугольника без заливки достаточно просто нарисовать три линии, здесь все достаточно понятно. Если же необходимо нарисовать залитый треугольник, то следует применить один из алгоритмов заливки пикселей.
Одним из возможных вариантов заливки треугольника - это алгоритм **line sweeping.**

Данный алгоритм по своей сути достаточно прост:

- Сортируем вершины треугольника по их y-координате
- Растеризуем параллельно левую и правую границы треугольника
- Отрисовываем горизонтальный отрезок между каждым левым и правым пикселем границы

```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, ImageBuffer &image, Color color) {
    // проверка нулевого треугольника
    if (t0.y==t1.y && t0.y==t2.y) {
       return;
    }

    // сортируем вершины сверху вниз
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);

    // суммарная высота
    int total_height = t2.y-t0.y;

    // циклом идем сверху вниз
    for (int i=0; i < total_height; i++) {
        // определяем в какой половине треугольника мы находимся
        // разделение происходит по средней точке
       bool second_half = (i > t1.y-t0.y) || (t1.y == t0.y);
```

```
        // определяем высоту текущей части треугольника
        int segment_height = second_half ? t2.y-t1.y : t1.y-t0.y;

        // ничего не делаем при нулевой высоте сегмента
        if(segment_height == 0){
           break;
        }

        // выисляем коэффициенты альфа и бэта
        float alpha = (float)i/ total_height;
        float beta  = (float)(i- (second_half?(t1.y-t0.y) : 0))/segment_height;

        // Вычисляем левую и правые точки на сторонах треугольника
        Vec2i A =               t0 + (t2-t0)*alpha;
        Vec2i B = second_half ? t1 + (t2-t1)*beta : t0 + (t1-t0)*beta;

        // A должно быть меньше, чем B
        if (A.x > B.x) std::swap(A, B);

        // Заполняем пиксели от левой стороны треугольника к правой
        for (int j=A.x; j <= B.x; j++) {
            // attention, due to int casts t0.y+i != A.y
            image.set(j, t0.y+i, color);
        }
    }
}
```



Принцип освещения

Смысл освещения трехмерной модели заключается в том, что чем перпендикулярнее полигон к источнику света, тем больше он освещен. Нулевая освещенность получается если полигон является параллельным источнику света.

С математической точки зрения - интенсивность освещённости равна скалярному произведению вектора света и нормали к данному треугольнику.

Нормаль к треугольнику может быть посчитана просто как векторное произведение двух его рёбер. (Например, по правилу правой руки)

```
for (int i = 0; i < triangles; i++) {
        const Triangle& triangle = triangles[i];

        // временные массивы для хранения координат треугольника в экранном и в
мировом пространстве
        vec2i screen_coords[3];
        vec3 world_coords[3];

        // идем по вершинам треугольника
        for (int j=0; j<3; j++) {
            vec3 v = triangle.getVert(j);
            // приводим к размеру изображения
            screen_coords[j] = vec2i((v.x+1.0)*width/2.0, (v.y+1.0)*height/2.0);
            world_coords[j]  = v;
        }

        // Вычисляем нормаль
        vec3 vec1 = (world_coords[2]-world_coords[0]);
        vec3 vec2 = (world_coords[1]-world_coords[0]);
        vec3 normal = vec1 ^ vec2;
        normal = normalize(normal);

        // интенсивность - это скалярное произведени вектора к свету и вектора
нормали
        light_dir = normalize(light_dir);
        float intensity = n*light_dir;

        if (intensity>0) {
            triangle(screen_coords[0], screen_coords[1], screen_coords[2],
                          image, Color(intensity));
              }
          }
```

ЗАДАНИЕ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Изучить альтернативные API для рендеринга, такие как DirectX, Metal, Vulkan. Попытаться реализовать практические задания и РГЗ на аналогичном API.

ЗАДАНИЕ ДЛЯ РАСЧЕТНО-ГРАФИЧЕСКОЙ РАБОТЫ

Целью расчетно-графической работы является получение опыта в оценке качества графического пользовательского интерфейса программного средства.

Задание:

1. Нарисовать цветной треугольник
2. Нарисовать кубик и покрутить его
3. Затекстурировать кубик
4. Простая шейдерная анимация у кубика (смещение координат текстур со временем). Для продвинутых - делание дыр в кубике по маске, в стиле создания 3D травы.
5. Создание вращающейся модели солнечной системы с анимацией вращения

ТРЕБОВАНИЯ К ОТЧЕТУ О ВЫПОЛНЕНИИ РАСЧЕТНО-ГРАФИЧЕСКОЙ РАБОТЫ

Отчет по выполнению расчетно-графической работы должен содержать:

1. Описание функционала рассматриваемого программного средства в виде списка функций.

2. Демонстрация работающего программного продукта

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ И ИНТЕРНЕТ-РЕСУРСОВ

1. Д. Вольф — Open GL 4. Язык шейдеров. Книга рецептов (2015)
2. Д. Гинсбург — OpenGL ES 3.0. Руководство разработчика (2014)
3. В. Порев — Компьютерная графика (2002)
4. П. Ширли — Основы компьютерной графики (2009)
5. Э. Ангел — Интерактивная компьютерная графика

Приложение 1. ОБРАЗЕЦ ТИТУЛЬНОГО ЛИСТА

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МУРМАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра
Математики, информационных систем и программного обеспечения

О Т Ч Е Т

о выполнении расчетно-графической работы

по дисциплине

«ИНТЕРАКТИВНЫЕ ГРАФИЧЕСКИЕ СИСТЕМЫ»

Выполнил:
студент группы _____
Фамилия И.О.

Проверил:
Старший преподаватель
кафедры МИС и ПО
Фамилия И.О.

Мурманск

20__