

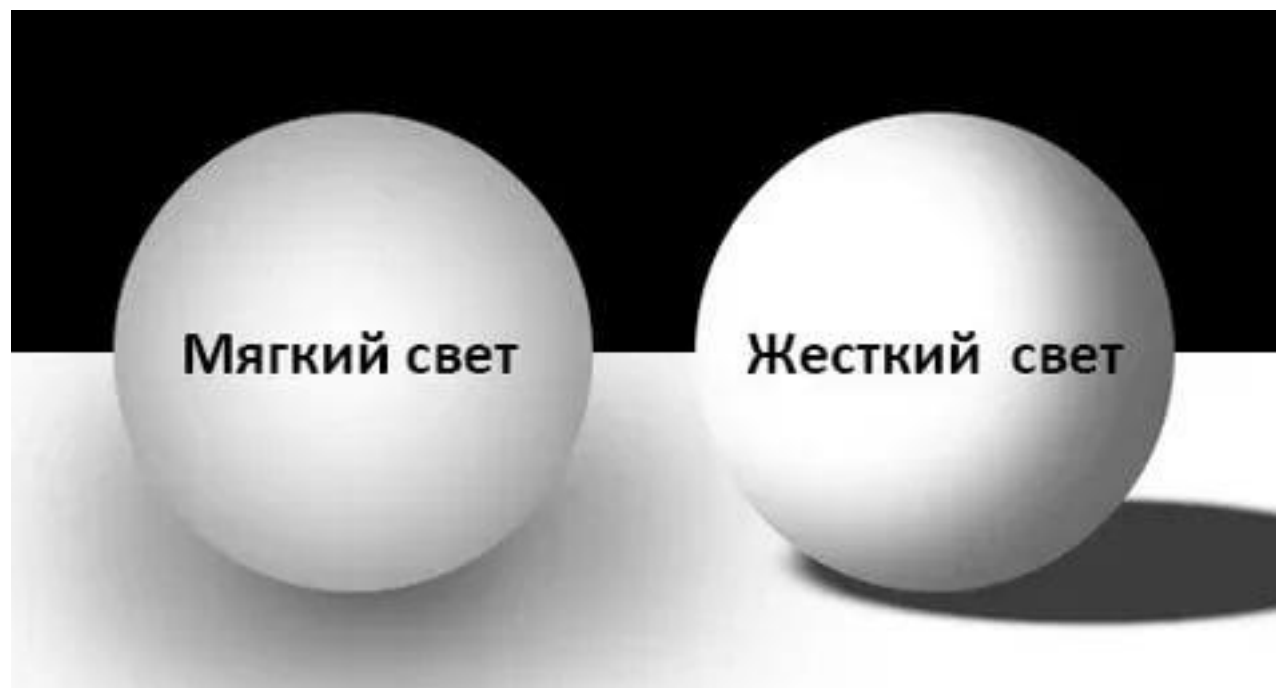
# Компьютерная графика

## Наложение теней в реальном времени

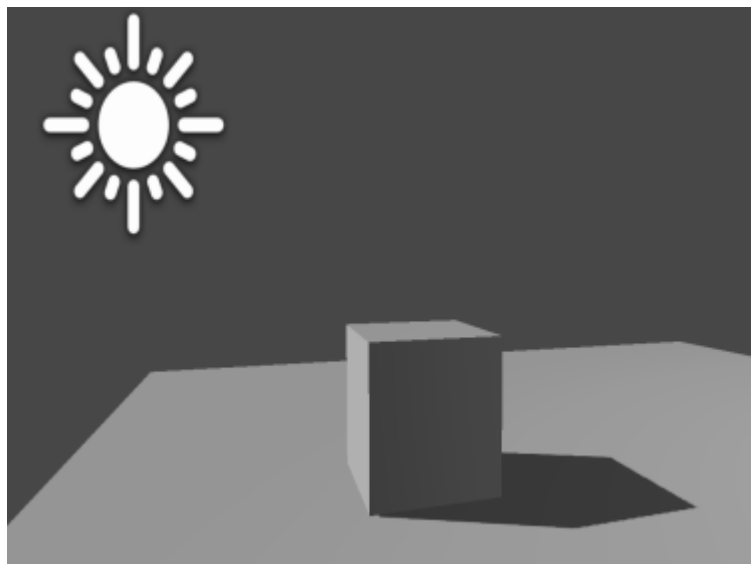
Лекция 9

Демяненко Я.М. ЮФУ 2023 MAGnUS

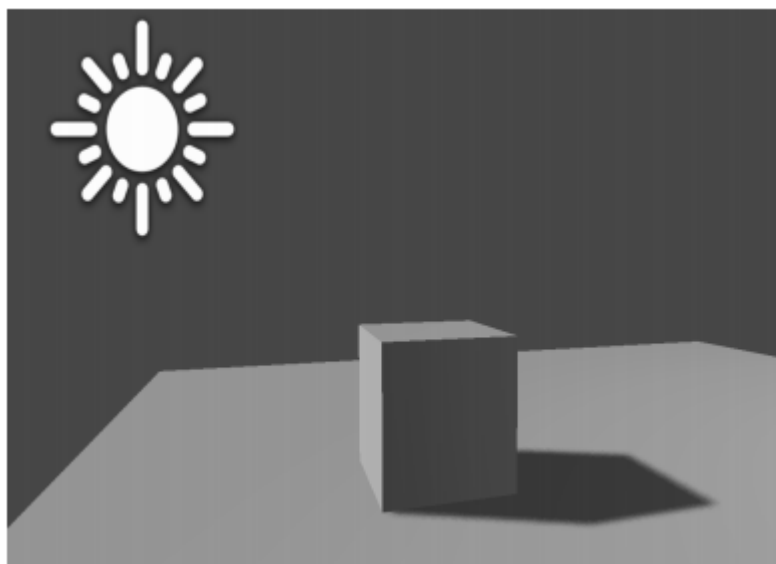
# Тени в жизни



# Типы теней



Жёсткие тени (hard shadows)



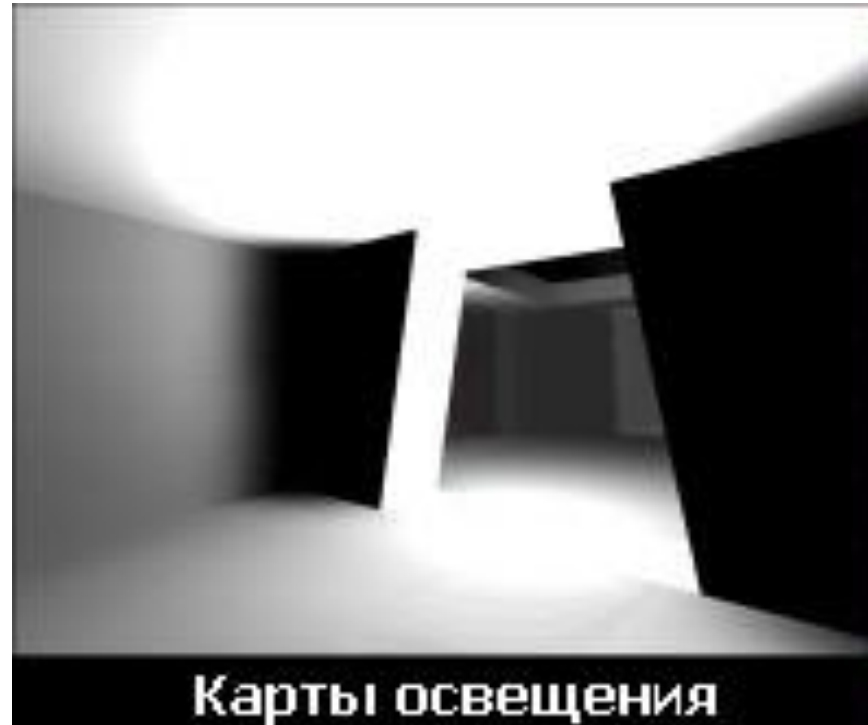
Мягкие тени (soft shadows)

# Методы рендеринга теней

- Карты освещения
- Вершинные тени
- Проектируемая геометрия
- Теневые объемы
- Теневые карты
- Теневые буферы
- Буферы ObjectID / приоритетов

# Карты освещения — lightmap

Метод освещения пространства, заключающийся в том, что создаётся текстура, содержащая информацию об освещённости трёхмерных моделей

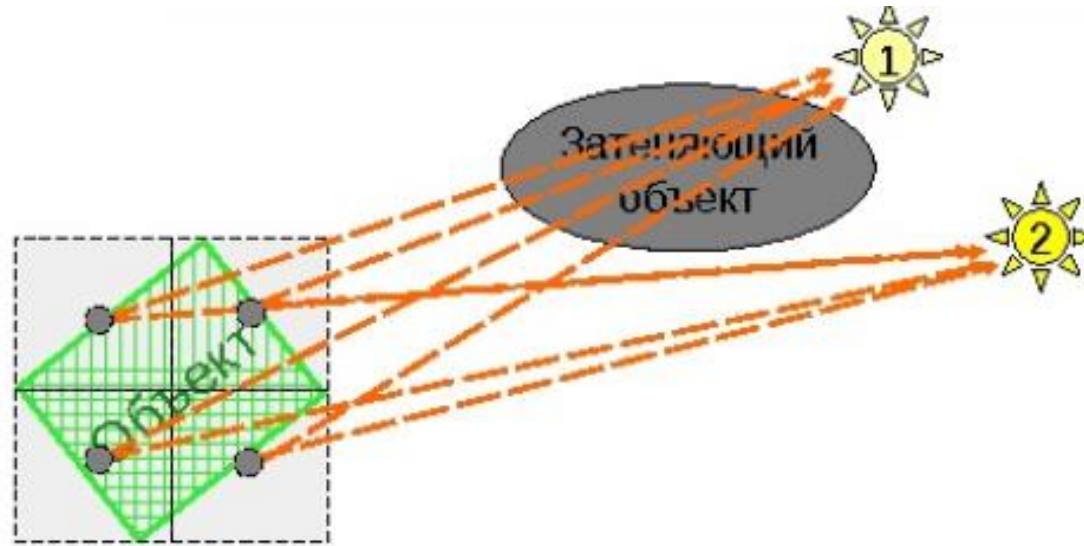


Поддерживается такими игровыми движками, как id Tech, Unreal Engine, Lithtech, GoldSrc, Xash3D, Source, X-Ray, Unity

# Использование карт освещения

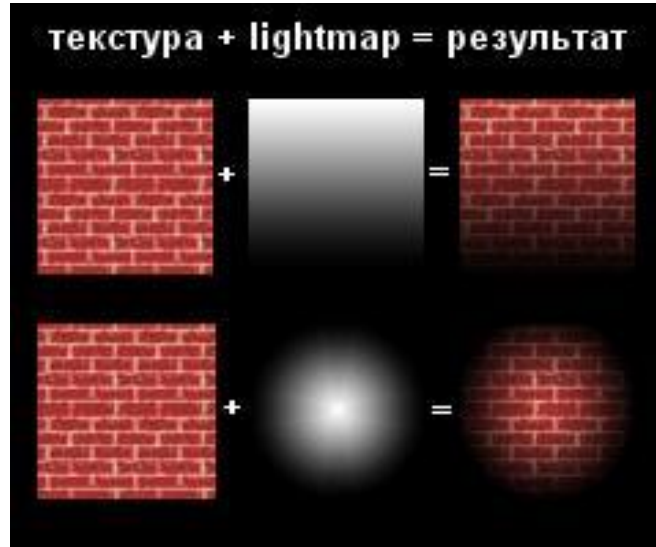
- Используются для расчёта освещения, в том числе теней
- Генерируется для статической геометрии до начала цикла рендеринга, и во время рендеринга в основном не изменяется
- Карты освещения для движущихся моделей создают только в случаях, если источник света присоединён к движущейся модели и результат её освещения не зависит от её перемещения и вращения.

# Разрешение карты освещения



- Для каждого полигона рассчитывается освещённость для каждого тексела его карты
- Каждый тексел карты освещения соответствует 4-32 текстелам текстуры

# Фильтрация и наложение текстур



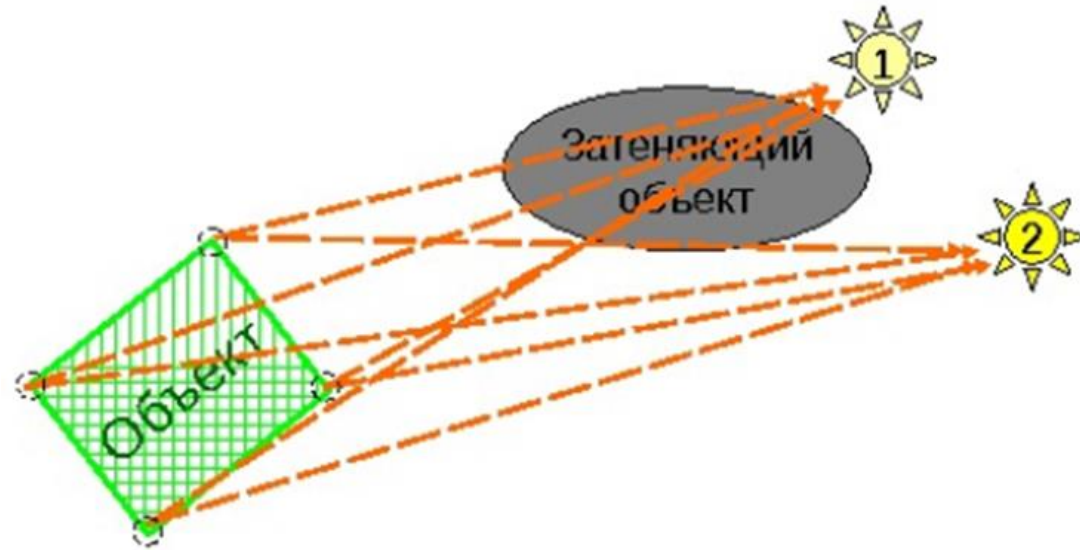
- В целях улучшения внешнего вида картинки, к картам освещения часто применяется билинейная фильтрация. Эти операции повторяются для каждого освещаемого полигона сцены.
- Во время рендеринга, карты освещения могут накладываться вторым проходом, с использованием альфа-блендинга. При наличии мультитекстурного оборудования можно накладывать текстуру и карту освещения за один проход.



# LightMap: Плюсы и минусы

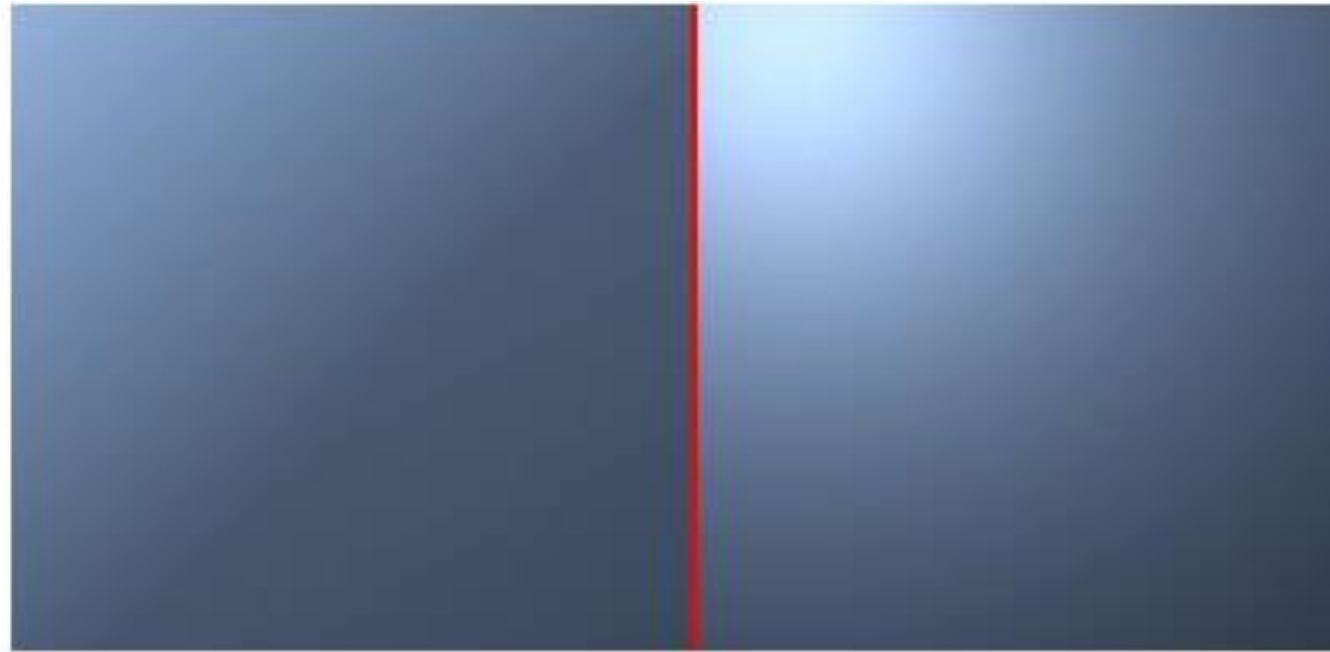
- + Самое высокое качество получаемого освещения
- + Не требует высокополигональных сцен
  
- Долгий расчет
- Занимает много видеопамяти
- Без упаковки LightMap'ов в одну текстуру из-за большого количества смен текстур долго
- При рендере занимает один текстурный юнит, который с успехом мог бы пойти на детализированную текстуру

# Вершинные тени

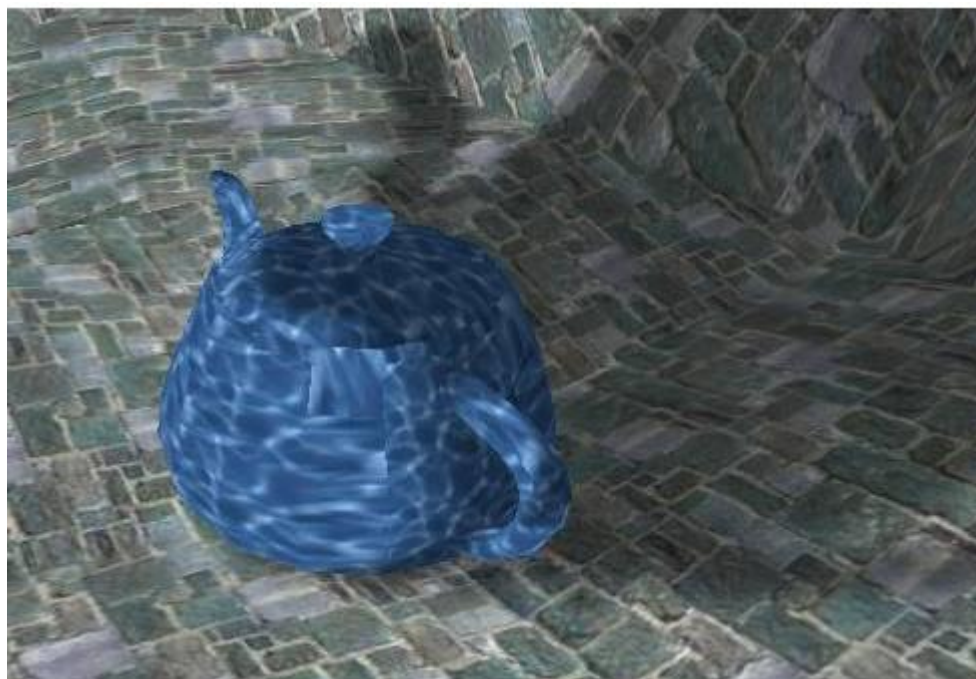


- Вершинное освещение, как правило, применяется для объектов, состоящих из большого количества маленьких полигонов
- Освещённость рассчитывается для каждой вершины полигональной сетки, и во время рендеринга интерполируется по полигону

# Vertex Lighting vs Lightmapping



# Вершинное освещение для высокополигональных моделей



# VertexLight: Плюсы и минусы

- + Быстрый расчёт для одного объекта
- + Требуется малое количество видеопамяти
- + Не занимает текстурный юнит
- + Работает быстрее, чем LightMap, так как нет необходимости менять текстуры
- + Может быть немного качественнее LightMap'ов из-за отсутствия текстурных фильтров
  
- Нужны высокополигональные объекты для качественного результата
- Тени в среднем все равно будут хуже, чем у LightMap'ов
- Расчёт может быть не таким уж и быстрым, только из-за того, что используются высокополигональные объекты

# Где Vertex Lighting и где Lightmapping?

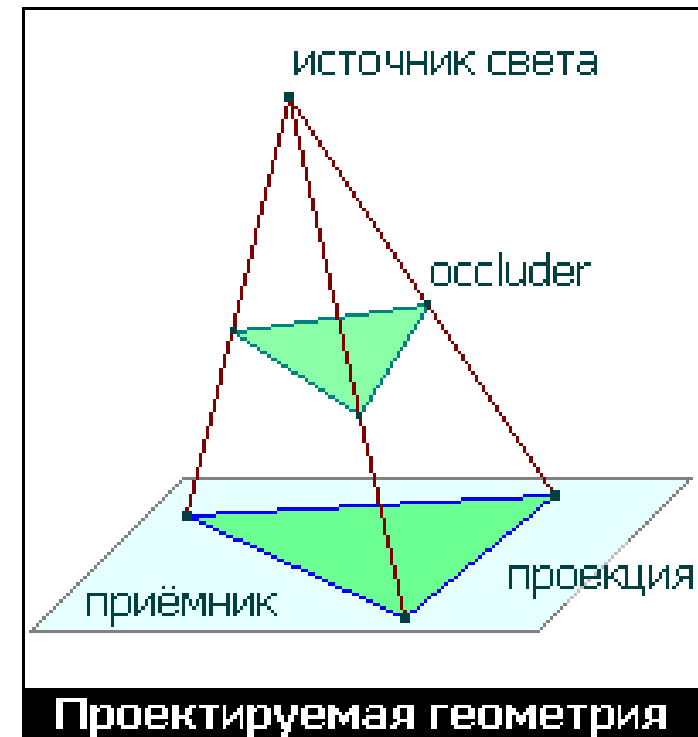


# Где хорошо Vertex Lighting ?



# Проектируемая геометрия

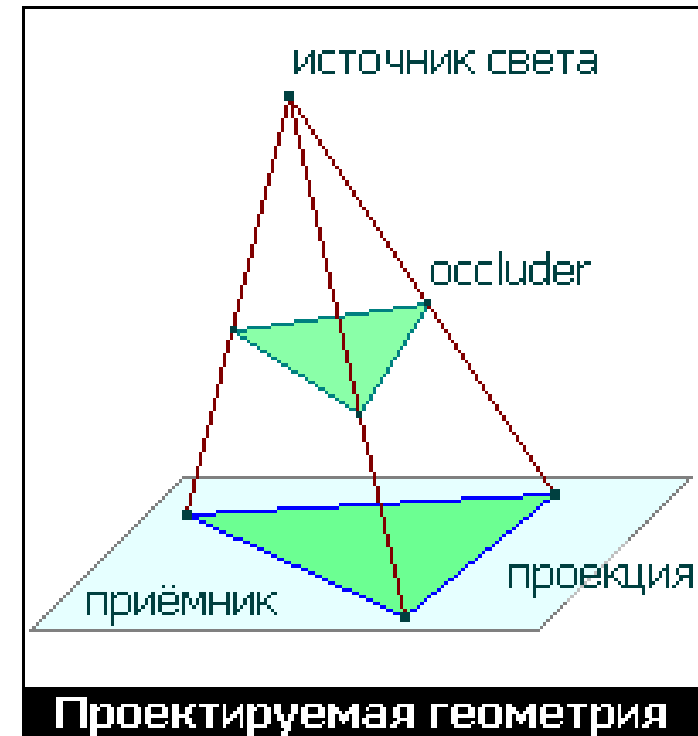
- Алгоритм относится к аналитическим
- Предполагается, что множество небольших occluder'ов отбрасывает тени на малое количество больших плоских объектов
- С помощью этого алгоритма нельзя рисовать все тени сцены, лишь самые важные





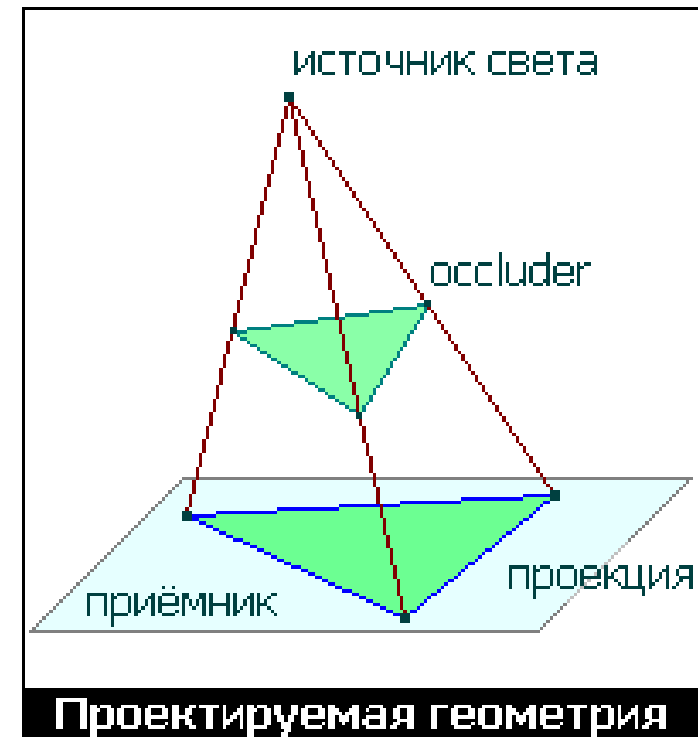
# Проектируемая геометрия: основная идея

- Основная идея алгоритма – проектирование объекта из точки положения источника света на приёмник теней
- Проекция получается применением трансформации модель/вид к оригинальному объекту:  $S = V * M * P$



# Occluder рисуется дважды:

1. один раз с нормальными матрицами  $V$  и  $M$  и оригинальными цветами и текстурами,
2. а второй раз - с использованием полученной матрицы  $S$ , цветом тени



# Особенности реализации

Проектирование осуществляется с помощью CPU, который должен также корректно обрабатывать случай выхода тени за границу приемника, т. е. отсекал тень по приёмнику

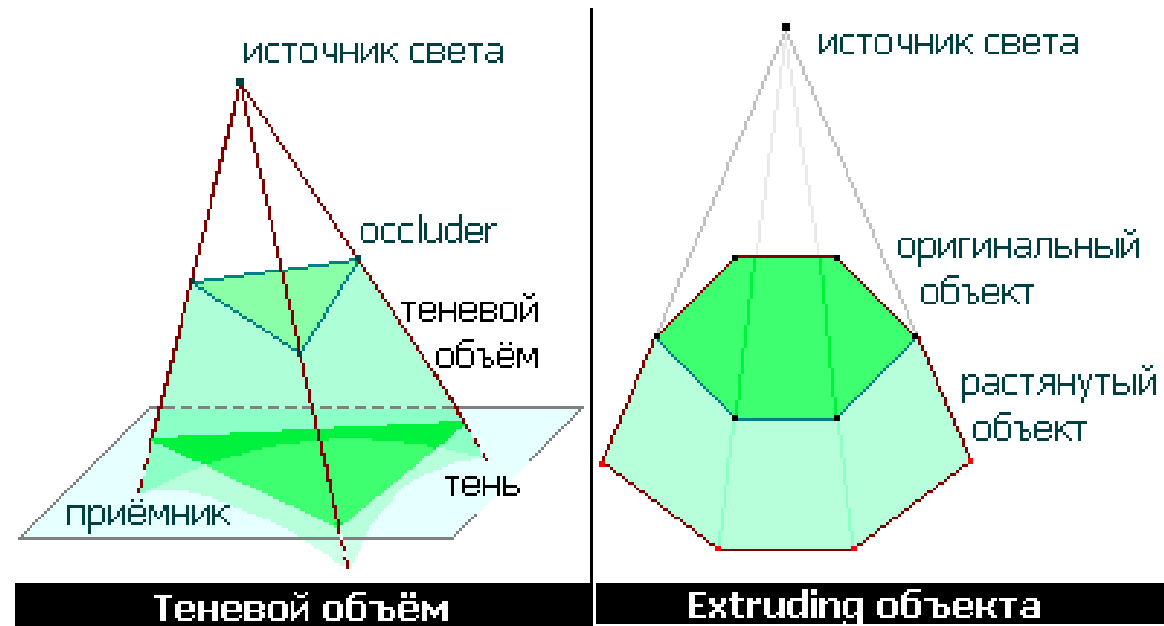
Вместо рисования проекции цветом тени, можно использовать alpha-blending для уменьшения яркости приёмника в областях тени, тогда сквозь тень видно текстуру приемника

В случае использования прозрачности возникает проблема двойного затенения: пересекающиеся спроектированные полигоны будут затенять приёмник 2 раза, что выглядит некорректно

Поэтому нужно искать минимальный силуэт объекта или избегать двойного наложения с помощью destination alpha или stencil-операций

Полигоны проекции нужно располагать не на самом приёмнике, а на некотором расстоянии от него, чтобы избежать ошибок точности z, либо опять использовать stencil

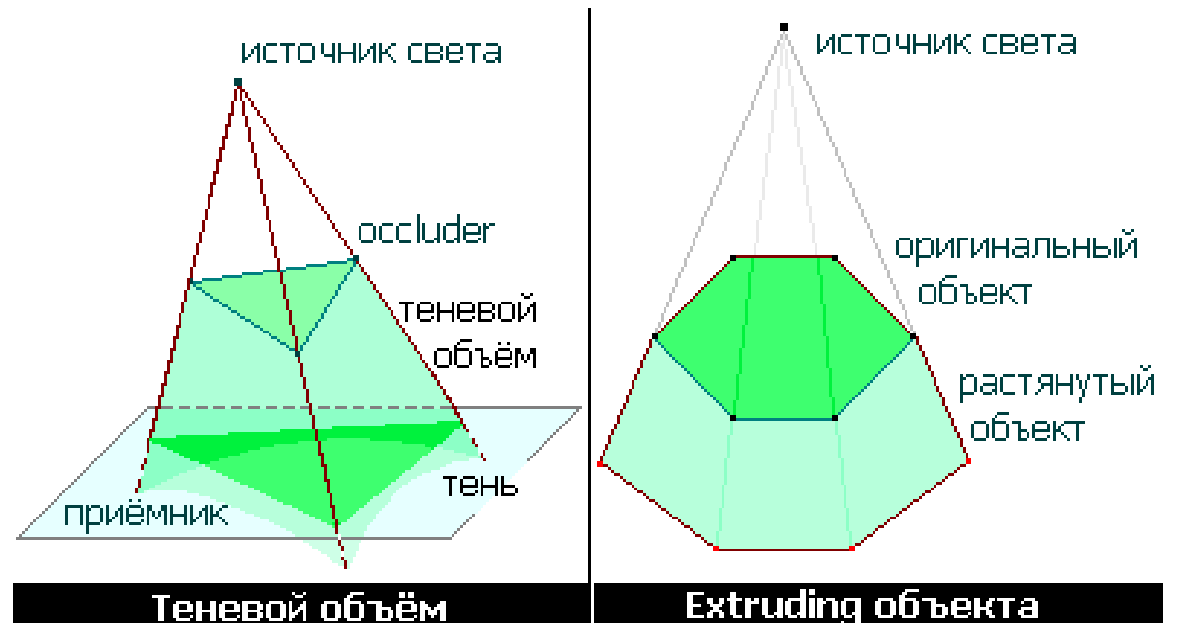
# Теневые объемы



- Основан на представлении затенённого пространства полигональным многогранником

# Теневые объемы: особенности

- Алгоритм работает достаточно эффективно только с использованием современного оборудования, поддерживающего z- и stencil-буферы
- До начала генерации теней сцена рисуется без учета источников света, генерирующих тени
- Перед началом обработки каждого источника света stencil-буфер очищается



# Проверка на попадание в сложный теневой объем

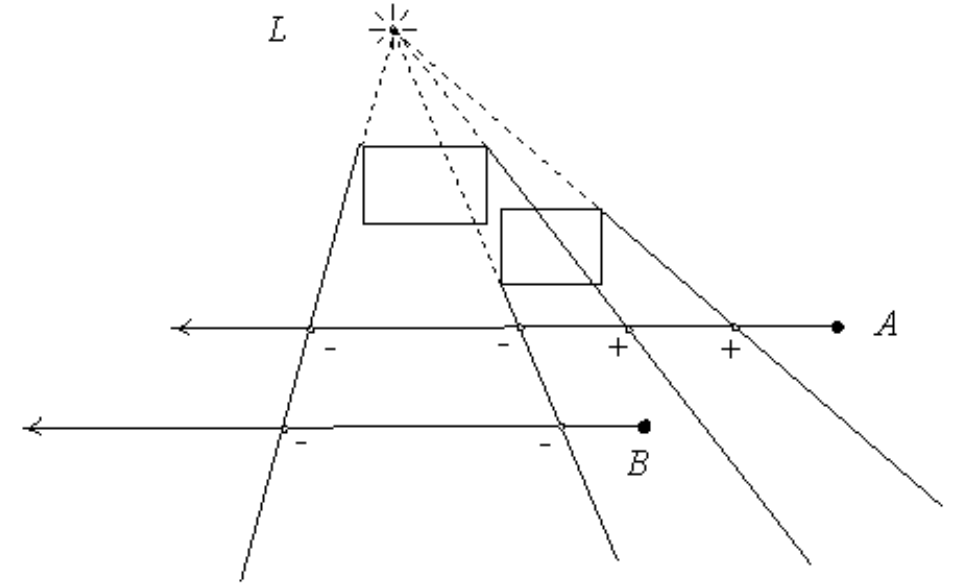
## а) Depth pass/Z-pass

Задние и передние грани теневого объёма рисуются в два прохода.

В каждом проходе различаются настройки рендера.

1. Отключим запись в буфер глубины и цвета
2. Включим отсечение задних граней. Установим операцию трафарета в инкрементирование при прохождении теста глубины. Нарисуем объём.
3. Включим отсечение передних граней. Установим операцию в уменьшение на единицу при прохождении теста глубины. Нарисуем объём.

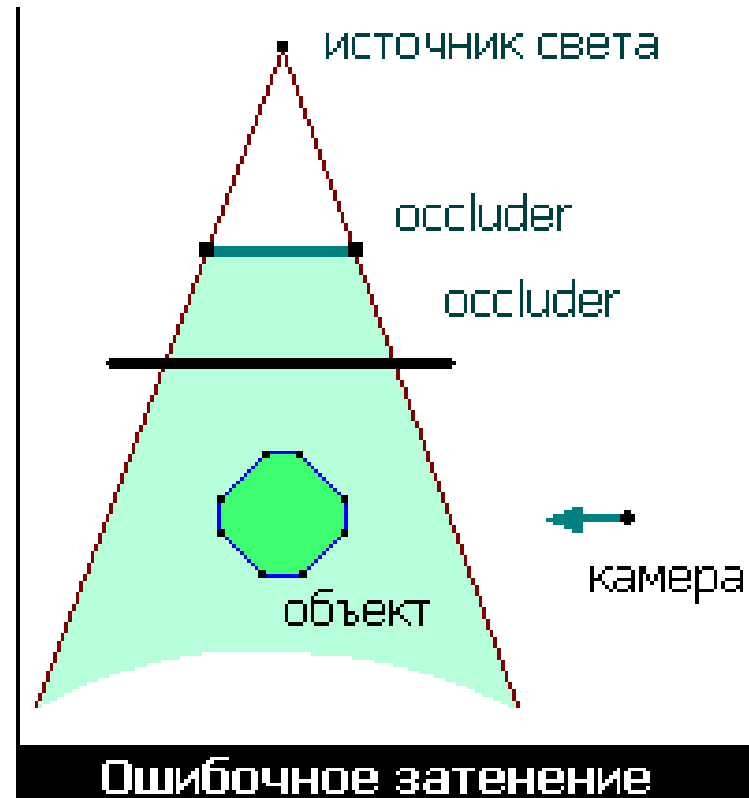
Все освещённые точки будут иметь значение 0 в буфере трафарета.



Метод даёт ошибку, когда камера находится внутри объёма. В этом случае камера не видит часть объёма, и все значения в буфере трафарета сдвигаются на -1, что приводит к инвертации теней.

Для обхода этой проблемы на картах NVidia было введено расширение `GL_NV_depth_clamp [spec]` с помощью которого можно отключить обрезание геометрии дальней и ближней плоскостями отсечения.

# Теневой объем: еще проблемы



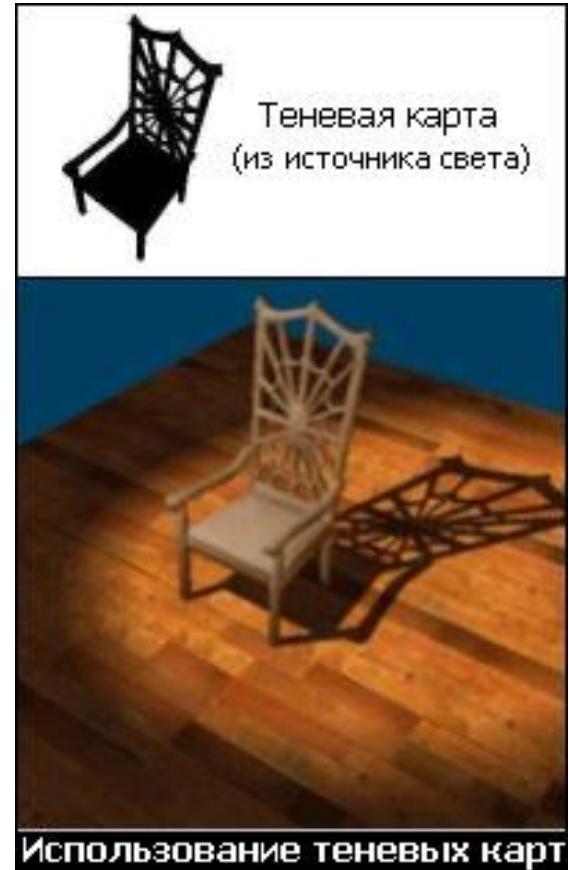
# Теневой объем: Плюсы и минусы

- + Намного большая по сравнению с проектируемой геометрией универсальность – производительность больше не зависит от сложности и количества приёмников тени, тень может падать на сколь угодно сложную поверхность
- Алгоритм слишком требователен к ресурсам, чтобы обеспечить генерацию всех теней сцены в реальном времени, к тому же размеры структур данных для любой достаточно сложной сцены могут достигать сотен мегабайт
- Производит тени с острыми краями
- Невыпуклые объекты не будут отбрасывать тени сами на себя (рука не затеняет торс)



# Теневые карты

Для каждой пары "источник света – объект" создаётся теневая карта, которую необходимо спроектировать на все приёмники теней



# Теневые карты

- Текстура очищается белым цветом, потом на нее из точки положения источника света рисуется shadow caster (черным)
- Для статических объектов и источников света это можно сделать перед началом цикла рендеринга, для движущихся придется делать в начале рисования каждого кадра
- Сначала объекты рисуются с обычными текстурами и текстурными координатами
- Затем каждая теневая карта устанавливается как активная текстура для объекта, проективные текстурные координаты генерируются, используя матрицу вида источника света, и объект рисуется еще раз в соответствующем альфа-режиме
- На оборудовании, которое поддерживает мультитекстурирование, можно рисовать объект и накладывать несколько теней за один проход

# Теневые карты: Плюсы и минусы

- + Алгоритм очень простой
- + Сравнительно быстрая генерация теней для небольшого количества объектов
- + Необходима только одна дополнительная прорисовка для каждого shadow caster'a
- + Алгоритм имеет проблемы с aliasing'ом, но допускает простое и дешевое их решение путем билинейной фильтрации
- Shadow caster'ы не отбрасывают тени сами на себя – если рассматривать объекты как приёмники теней, то они будут полностью затеняться своей же теневой картой. Поэтому на объекты нельзя проектировать их тени. Эта проблема существенна только для сложных, невыпуклых объектов
- Для каждой пары «объект - источник света» необходим свой render-to-texture, что ограничивает возможное количество таких пар

# Теневые буферы

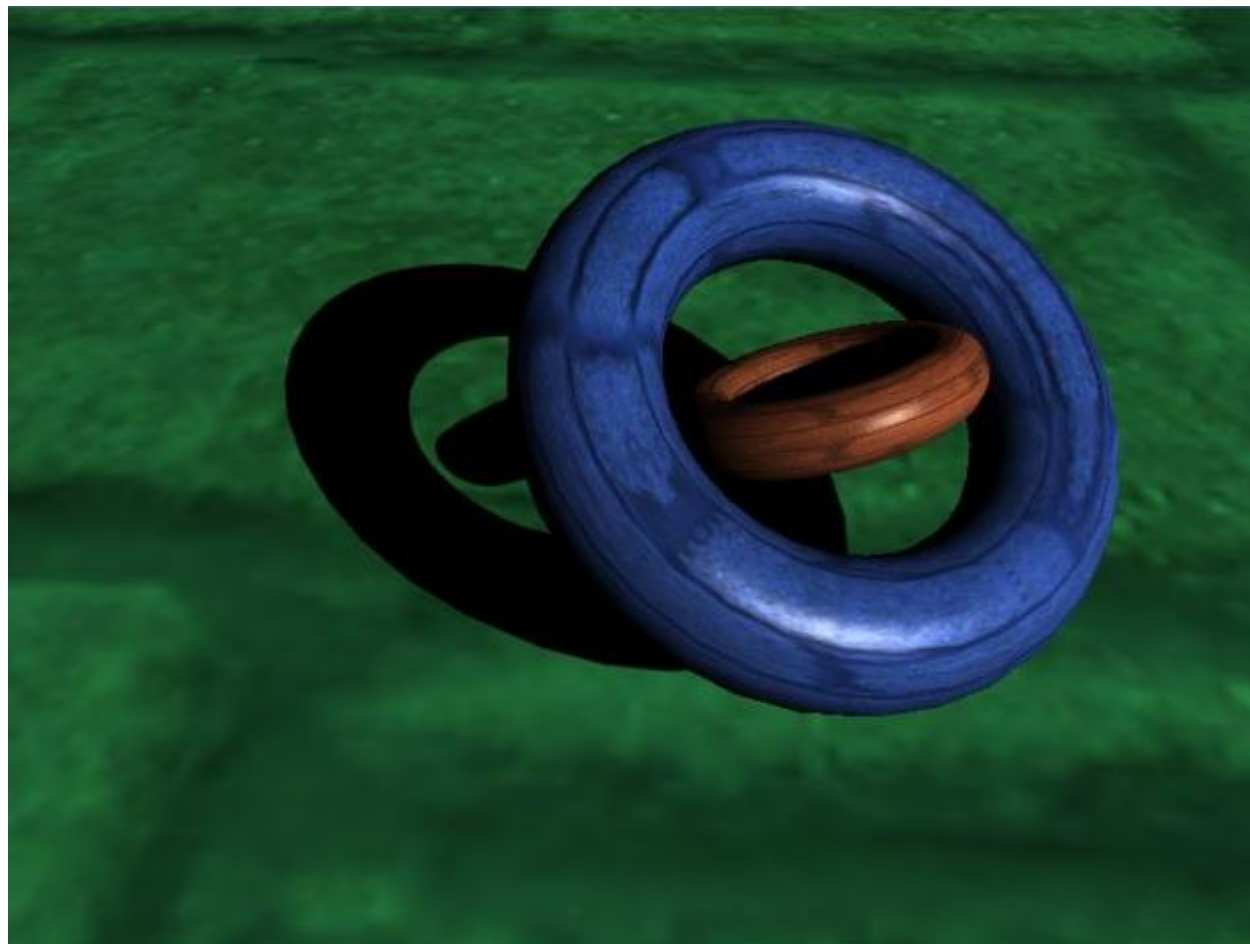


- Алгоритм базируется на том, что область, находящаяся в тени относительно источника света, не видна из точки его положения
- Для начала сцена рисуется из источника света во внеэкранный буфер
- В z-буфере (теневом буфере) записаны глубины всех ближайших к источнику света точек. Любая точка, находящаяся дальше соответствующей точки в теневом буфере, будет в тени

# Теневые буферы

- + Алгоритм не зависит от сложности геометрии сцены
- + Поддерживает самозатенение
- Алгоритм подвержен aliasing'у
- Для каждого источника света требуется отдельный рендеринг сцены
- Плохо работает для точечных источников
- Требуется аппаратной поддержки или много дополнительных проходов рендеринга

# Построение теней при помощи теневых буферов



# Основные идеи

- В основе этого метода лежит крайне простая идея, что освещенные фрагменты – это те фрагменты, которые видны из положения источника света.
- Поэтому, если мы расположим камеру в положении источника света и отрендерим сцену при помощи этой камеры, то мы получим все освещенные фрагменты.
- Видимость фрагмента источнику света в терминах буфера глубины означает, что данный фрагмент успешно проходит тест глубины при рендеринге из положения источника света.

# От буфера глубины к текстуре

- Буфер глубины поместить в специальную текстуру, которая будет использоваться для проверки видимости фрагментов из положения источника света
- Вычисление текстурных координат  $(s,t,r)$ , при котором  $(s,t)$  определяют проекцию соответствующего фрагмента на картинную плоскость при рендеринге из положения источника света, а третья компонента  $r$  является глубиной фрагмента относительно источника света
- При этом текстурные координаты  $(s,t,r)$  фактически являются координатами фрагмента в системе координат камеры, использованной для рендеринга из положения источника света
- Тогда для определения видимости источником света данного фрагмента достаточно просто сравнить значение глубины  $\text{depthMap}(s,t)$  и  $r$



# Что нужно?

- Построить две матрицы:
  - матрицу вида (view matrix)
  - матрицу проекции (projection matrix)
- Для направленного источника освещения используется ортогональная матрица проекции
- Для точечного источника освещения используется перспективная матрица проекции
- На каждый источник освещения необходим дополнительный проход рендера сцены

# Алгоритм

Запрещается запись в буфер кадра, нужно только содержимое буфера глубины

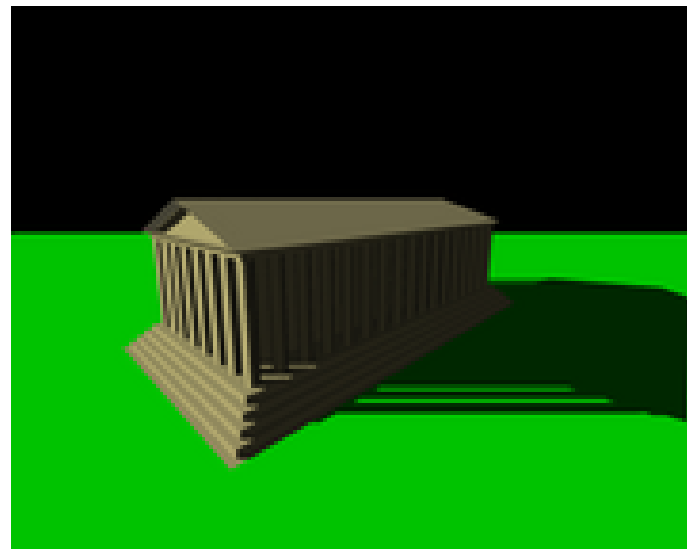
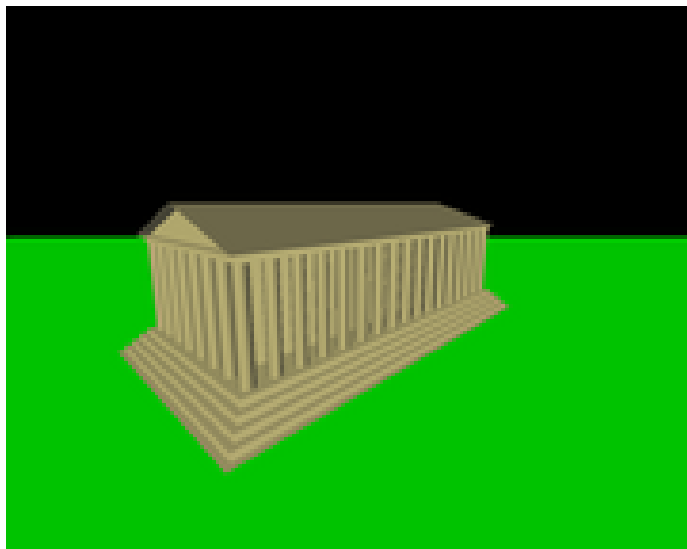
В положении источника света размещается виртуальная камера и производится рендеринг всей сцены при помощи этой камеры

Данные из буфера глубины копируются в специальную текстуру, называемую теневым буфером

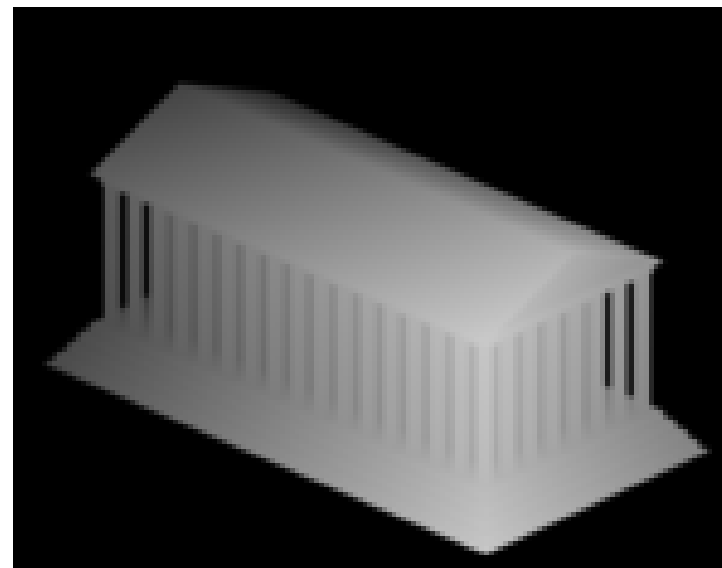
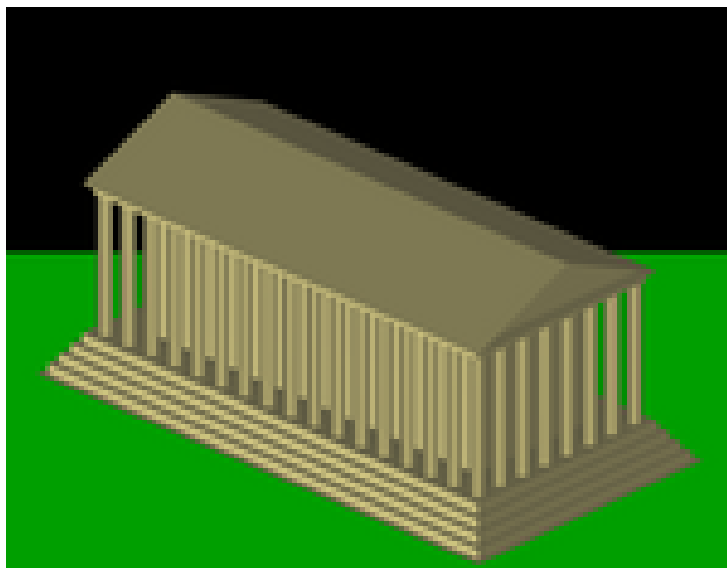
Проводится обычный рендеринг сцены из положения наблюдателя

Для каждого выводимого фрагмента вычисляются специальным образом текстурные координаты  $(s,t,r)$  для доступа к теневой карте

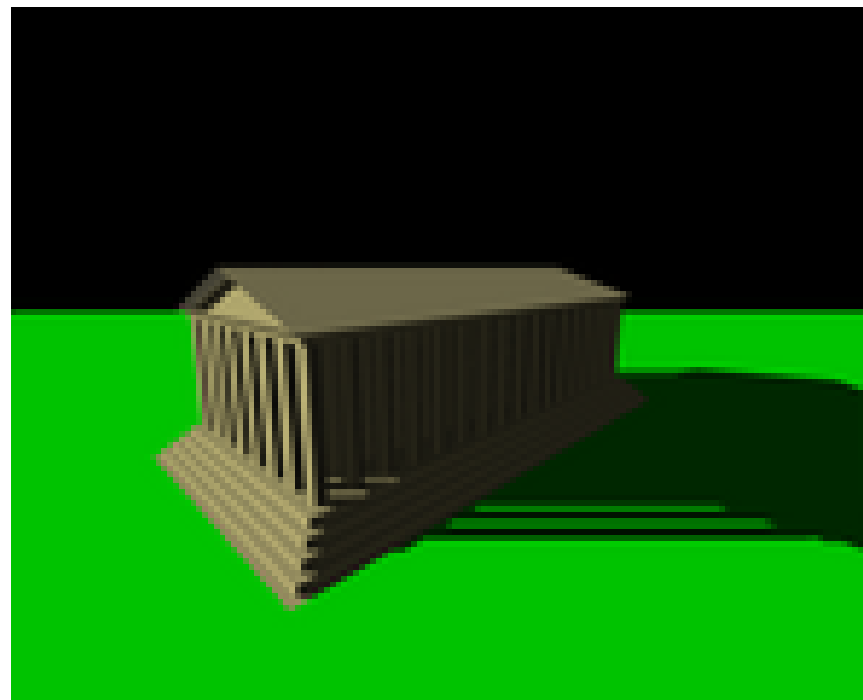
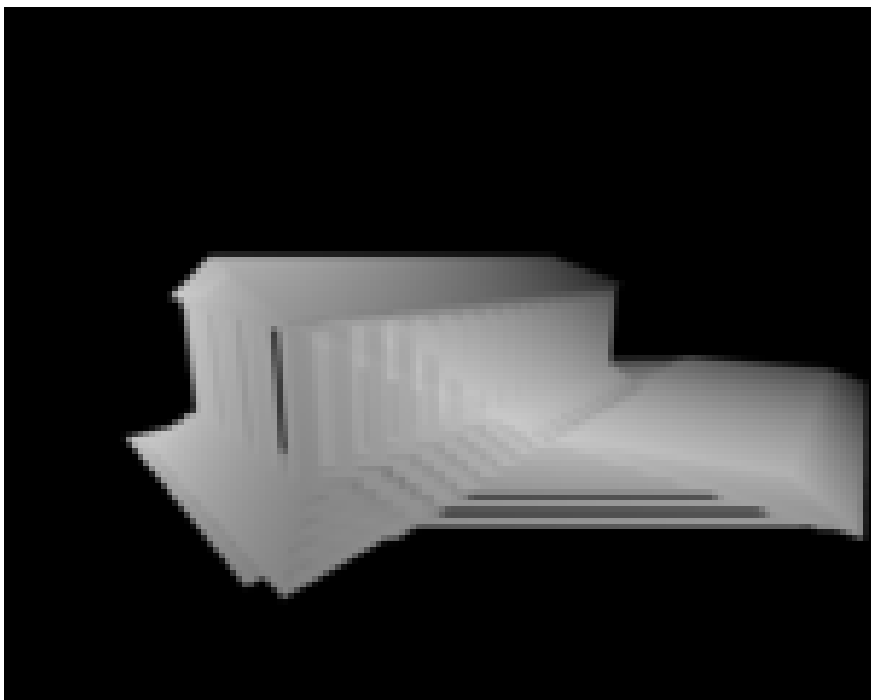
# Эффект наложения тени



# Создание теневого буфера



# Наложение теневого буфера



# Этапы реализации

- Создать текстуру для хранения буфера глубины
- Создать Framebuffer Object (FBO) и привязать к нему текстуру
- Настроить камеру и выполнить рендер сцены в созданный FBO

# Требования

- Специальный формат текстуры для хранения карты глубины (обладающий достаточной точностью)
- Быстрый способ вычисления  $\text{depthMap}(s,t) \leq r$

# Формат текстуры

//создание текстуры под depth данные

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width, height, 0, GL_DEPTH_COMPONENT,  
             GL_FLOAT, NULL);
```



# Быстрый способ вычисления

```
// Установить новый режим текстурирования для использования depthMap(s,t)<=r  
  
// pname GL_TEXTURE_COMPARE_MODE  
// param GL_COMPARE_REF_TO_TEXTURE  
  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);  
  
// Вернуться к нормальному режиму текстурирования  
  
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE );
```

# Типы экранных буферов

- Буфер **цвета**, в котором хранятся значения цвета фрагментов
- Буфер **глубины**, хранящий информацию о глубине фрагментов
- Буфер **трафарета**, позволяющий отбросить часть фрагментов согласно определенному условию.

Комбинация этих трех буферов зовется **кадровым буфером** (фреймбуфером) и хранится в определенной области памяти.

## Фреймбуфер



Фреймбуфер, сам по себе это только точка привязки, к нему привязываются так называемые рендер-текстуры.

Рендер-текстуры уже будут хранить непосредственно цветовую информацию.

# Кадровый буфер

- Все операции отрисовки, что мы выполняли до сих пор, исполнялись в рамках буферов, прикрепленных к базовому кадровому буферу.
- Базовый буфер кадра создается и настраивается в момент создания окна приложения.
- Создавая собственный кадровый буфер мы получаем дополнительное пространство куда можно направить рендер.

# Применение собственных кадровых буферов

- Тени (теневого буфер)
- Screen Space Ambient Occlusion (SSAO) – эффект самозатенения
- Эффекты зеркал
- Screen Space Reflections (SSR) – алгоритм который позволяет получить эффект отражения для плоских поверхностей
- Программная генерация текстур
- Постобработка
- Deferred Shading – отложенное освещение и затенение
- Depth of Field (DoF) – глубина резкости
- High Dynamic Range Compression (HDR) – технология расширения диапазона яркости кадра
- Bloom – своеобразный эффект свечения, заключается в размытии границ ярких объектов.
- Screen Space Subsurface Scattering (SSSS) – подповерхностное рассеивание

# Создание FBO

```
//FBO для рендера в него буфера глубины
GLuint depthFBO=0;
//переменная для получения состояния FBO
GLenum fboStatus;

//создаем FBO
glGenFramebuffers(1,&depthFBO);
//делаем созданный FBO текущим
glBindFramebuffer(GL_FRAMEBUFFER,depthFBO);

//отключаем вывод цвета в текущий FBO
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

# Привязка текстуры к FBO

```
//указываем для текущего FBO текстуру,  
//куда следует производить рендер глубины  
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTexture, 0);  
  
//проверим текущий FBO на корректность  
if ((fboStatus=glCheckFramebufferStatus  
    (GL_FRAMEBUFFER))!=GL_FRAMEBUFFER_COMPLETE){  
    LOG_ERROR(fboStatus);  
    return false;  
}  
  
//выполняем нужные действия  
...  
  
//возвращаем FBO по умолчанию  
glBindFramebuffer(GL_FRAMEBUFFER,0);
```