Компьютерная графика и визуализация в реальном времени

Deferred shading

Алексей Романов

План

- *Forward render*
- *Deferred render*
- Упаковка нормалей
- Отрисовка полупрозрачных объектов
- Оптимизации и альтернативные подходы
  - *Tile-based rendering*
  - *Inferred render*
  - *Forward+ render*

Forward render

Результирующей цвет вычисляется шейдером объекта

- Учет освещения от всех источников
- Применение теней
- …

```
struct light_t
{
    ...
};

uniform light_t lights[N];

void main()
{
    ...
    vec3 res = vec3(0);
    for (int i = 0; i < N; ++i)
    {
        res += calc_light(pos, normal, lights[i]);
    }
    ...
}
```

Плюсы *forward render*

Шейдера различных объектов могут быть абсолютно произвольными

Стандартная модель освещения

Специальная модель освещения для деревьев

Недостатки *forward render*

- Сложность определения и передачи всех источников света, влияющих на объект
- Большая сложность отрисовки перекрытия объектов: $O(n \cdot m)$, n – количество перекрывающихся фрагментов, m – количество источников света
- Неудобство поддержки шейдеров объектов из-за их большого размера

Необходимо до отрисовки объекта определить все источники света, оказывающие на него влияние

Deferred render

- Впервые в “промышленных масштабах” был использован в *S.T.A.L.K.E.R*, описан в «*GPU Gems 2, ch.19*»
- Сейчас использует в большинстве *state-of-the-art* игровых проектов

Deferred render

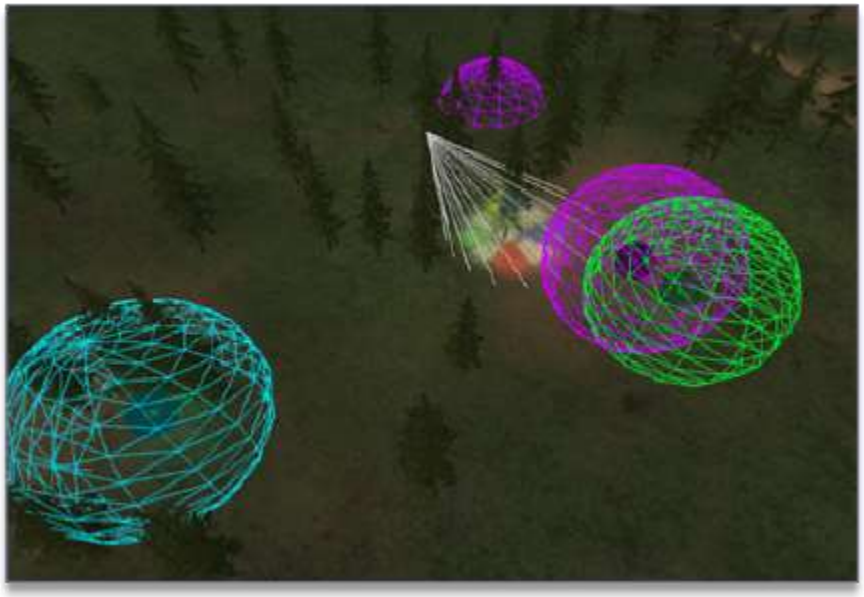
- Основная идея: отложить расчет освещения до самого последнего момента
- Шейдера объектов выдают данные, необходимые для освещения. $\left(K_a + (\bar{n} \cdot \bar{l})\, K_d + (\bar{n} \cdot \bar{r})^m K_s\right)$
  - K_a, K_d, K_s, m – свойства поверхности
  - $\bar{n}$ - нормаль
  - $\bar{p}$ - позиция объекта, влияет на $\bar{r}$
- Освещение накладывается, используя заранее подготовляемые параметры
- Меньшая сложность в случае сильно перекрывающихся объектов $O(n + m)$

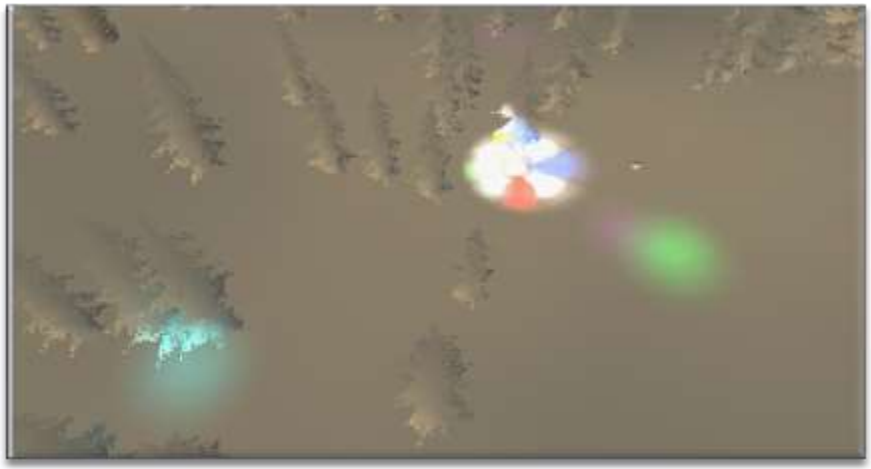
Основные стадии *deferred render*

- Построение набора буферов, называемых *geometry buffer* или *gbuffer*, и содержащих параметры освещения
  - *Albedo*
  - Зеркальность
  - *ambient*, diffuse, …
- Аккумуляции освещения в *light buffer* или *lbuffer*
- Композиция *gbuffer* и *lbuffer* с выводом на экан

Аккумуляция освещения

- Каждый источник рисует объем своего влияния
  - Сфера
  - Конус
  - Вся область (от солнца)
- В растеризуемых фрагментах вычисляется освещение и аккумулируется в буферах





Deferred render в Killzone2

Преимущества *deferred render*

- Отрисовка геометрии за один проход
- Простота изменения глобальной системы освещения
- Слабое влияние количества источников света на производительность

Преимущества *deferred render*

Легко добавить глобальный эффект

Например, дождь из S.T.A.L.K.E.R.:clear sky

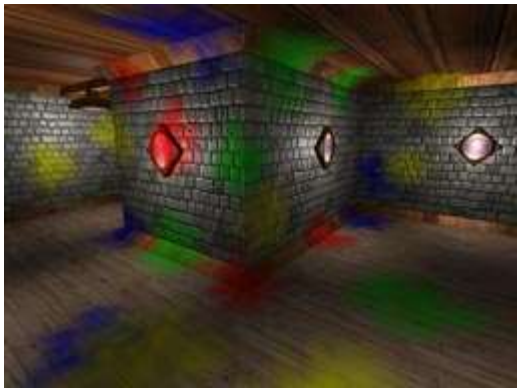
Динамический дождь выключен

Динамический дождь включен

Нормали дождя

Преимущества *deferred render*

- Наложение проективных объектов (*decals*), модифицирующих *gbuffer*
  - *Albedo* – изменение цвета
  - *Normal* – для модификации освещения
  - *Pos* – для реализации *parallax occlusion'a*



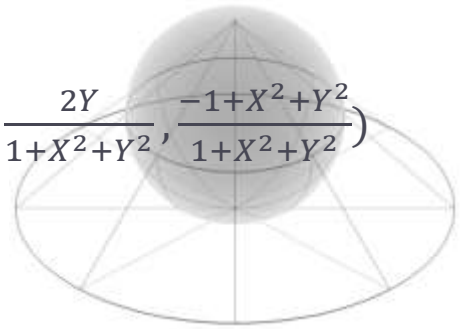
Недостатки *deferred render*

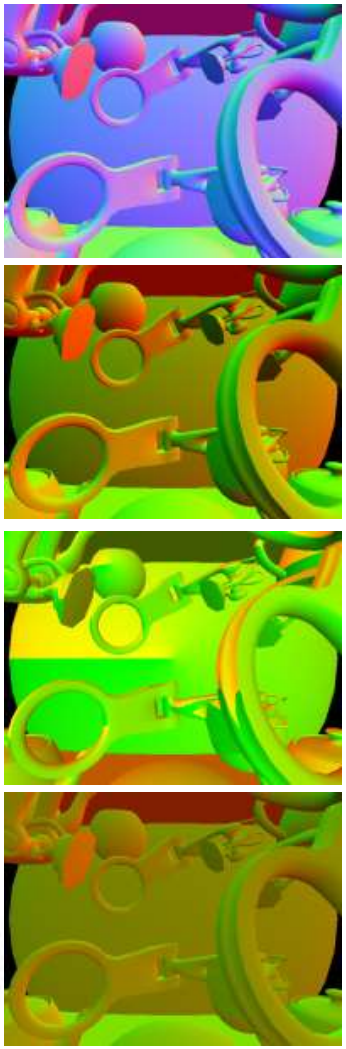
- Плохая совместимость с *MSAA*
- Трудно поддержать несколько принципиально разных типов шейдеров
  - Индекс материала
  - На практике дает 2-3 разных материала, не более
- Б**о**льшая сложность по памяти
- Отдельный проход для спец. полупрозрачных объектов (море)

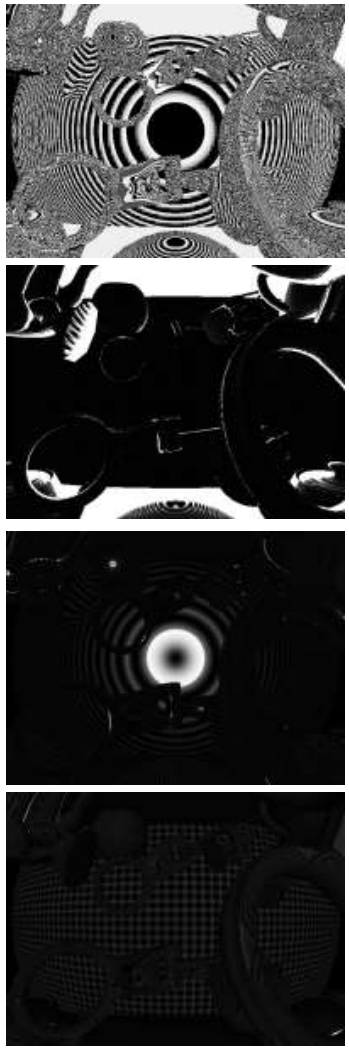
Преимущества перекрывают недостатки

Упаковка нормалей

- Без кодировки
  - *RGB8, RGB32, RGB16*
- $x, y - F16, z = \sqrt[2]{1 - x^2 - y^2}$
  - Нормаль может быть направлена от камеры
- $\varphi, \theta - F16$, углы в сферических координатах
  - Кодировка содержит тригонометрические инструкции
- Стереографическая проекция
  - $(X, Y) = (\frac{x}{1-z}, \frac{y}{1-z})$
  - $(x, y, z) = (\frac{2X}{1+X^2+Y^2}, \frac{2Y}{1+X^2+Y^2}, \frac{-1+X^2+Y^2}{1+X^2+Y^2})$

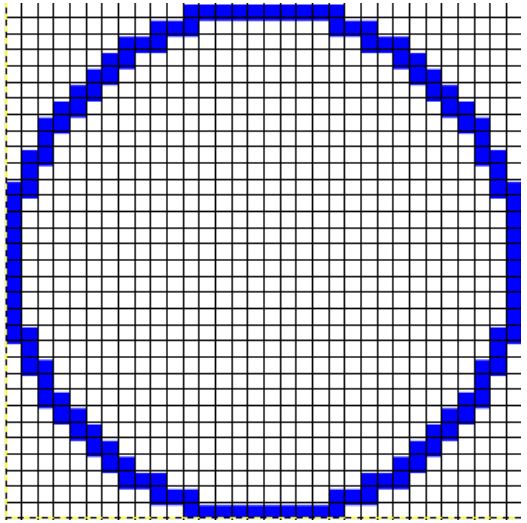


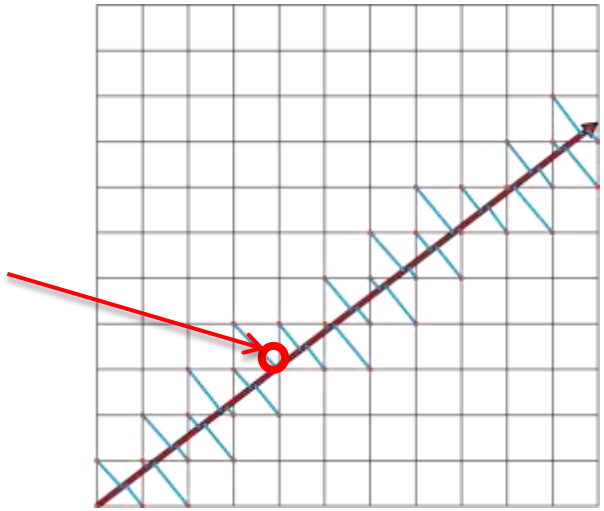




Упаковка нормалей, *Best fit*

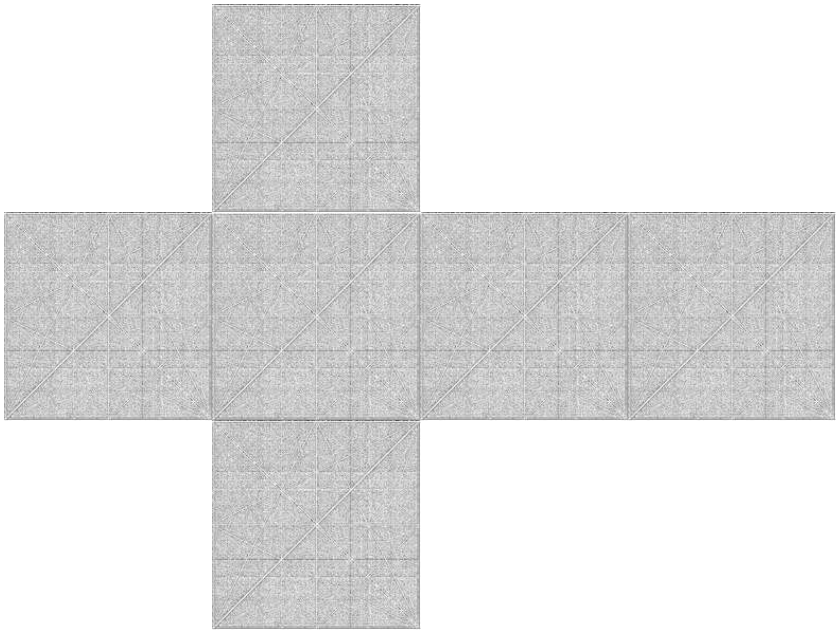
- Рассмотрим *RGB8*
  - Куб $256x256x256 = 16777216$ ячеек
  - Используются только ячейки на сфере, вписанной в куб $\sim \frac{289880}{16777216} = 1.73\%$
- *Best fit*
  - Поиск ячейки с минимальной погрешностью
  - Результат запишем в *cube map* текстуру
  - Разрешение текстуры $> 256x256$
  - Текстура считается заранее
  - Текстура содержит расстояния

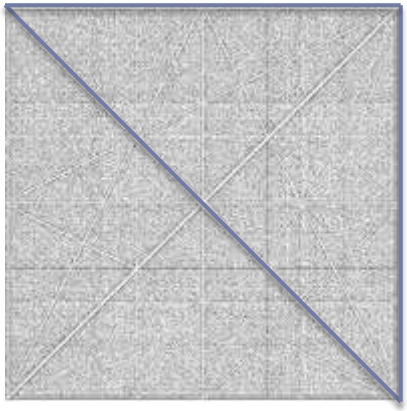




Упаковка нормалей, *Best fit*

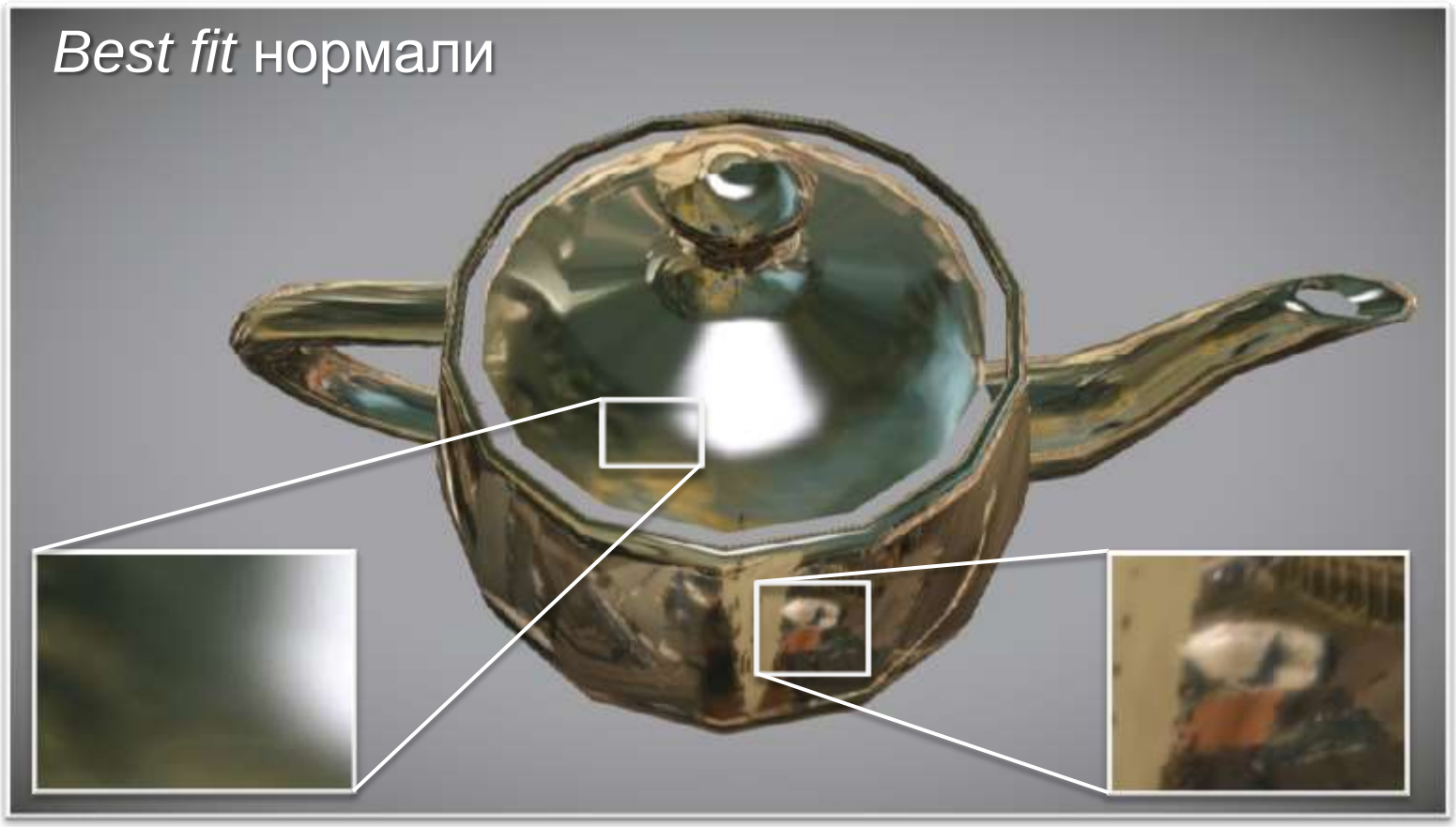
- Текстура получилась очень симметричной, извлечем из нее несимметричную часть
- Сохраним ее в *2d* текстуру
- Считаем значение из этой текстуры перед записью нормалей в *gbuffer*
- Отмасштабируем нормаль
- Запишем изменную нормаль в *gbuffer*



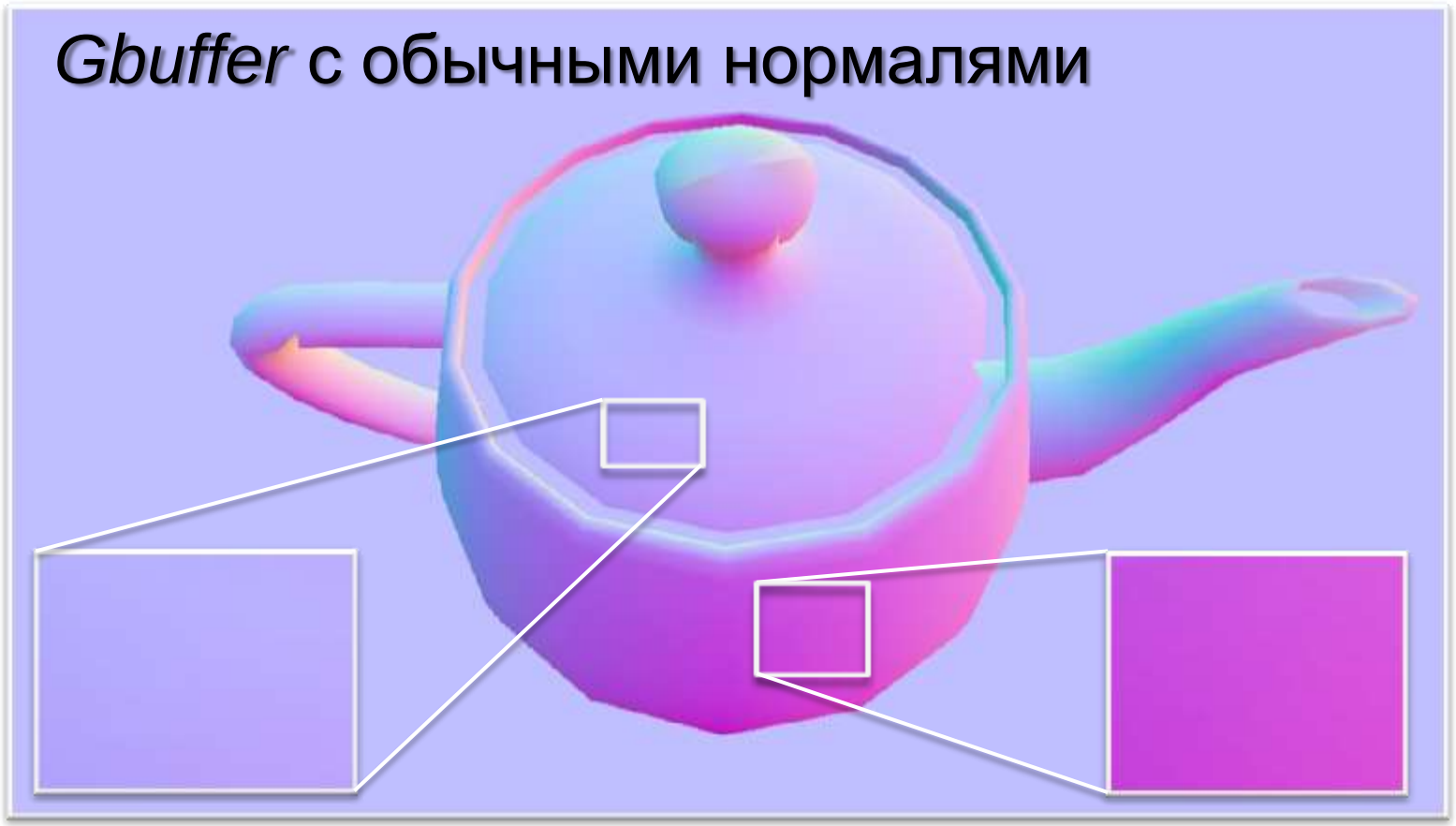


Упаковка нормалей, примеры

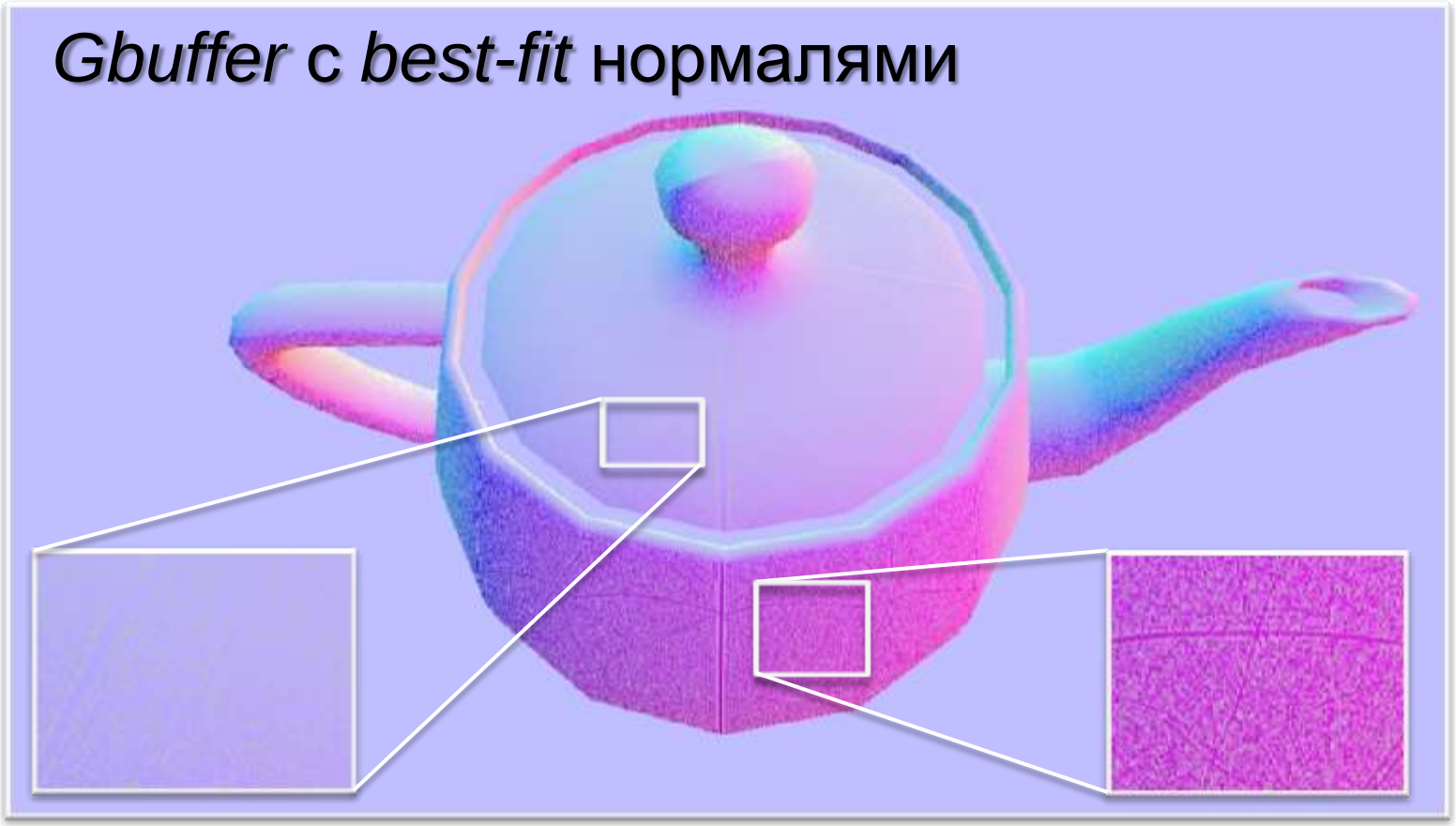
Упаковка нормалей, примеры



Упаковка нормалей, примеры



Упаковка нормалей, примеры



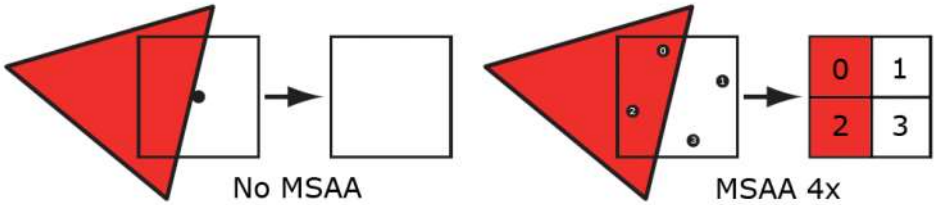
Полупрозрачные объекты: растительность

- Blending на gbuffer не работает
  $L\big(mix(g_1, g_2)\big) != mix(L(g1), L(g_2)$
- Корректно смешивается освещение
- Alpha to coverage

Увеличенный фрагмент: количество отброшенных фрагментов коррелирует с прозрачностью

Alpha test

Alpha to coverage



Работает только с аппаратным антиалиасингом (MSAA, …)

Полупрозрачные объекты: дождь

- Дождь – моделируется большим количеством частиц
- Render particles center positions as points to gbuffer
- Применение освещения к gbuffer’у
- Отрисовка частиц дождя, запрос освещения в центральной точке

Освещения для точек

Освещение применено к частицам

- Удаление точек из gbuffer’а

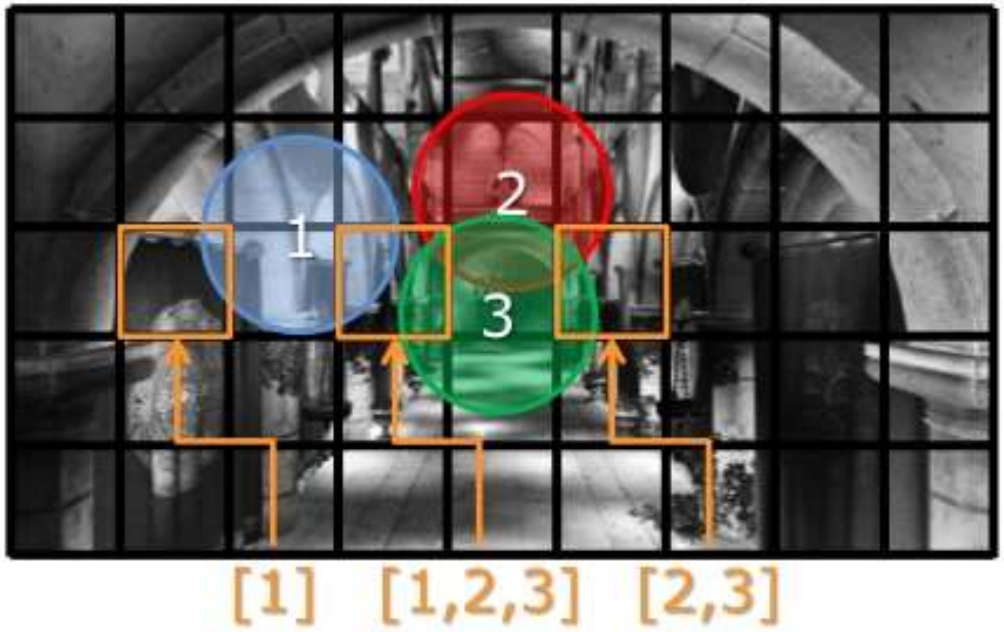
Оптимизации и альтернативные подходы

- *Tile-based rendering*

  Frostbite 2, Battlefield 3

- *Forward+ render*

  AMD Leo demo



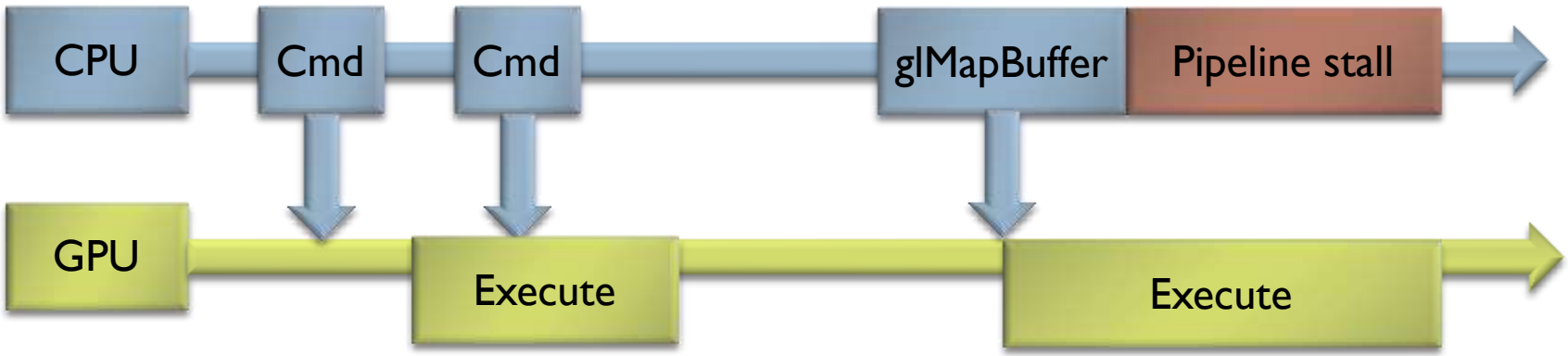
  - *Zprepass*
  - *Lights tiled culling*
  - *Forward render*

GPU/CPU проблемы синхронизации

- Необходимо на каждом кадре передать информациб об источниках света на GPU
- Грубый подход – mapbuffer на кажом кадре

```
glBindBuffer(GL_ARRAY_BUFFER, id);
void * data = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
// fill data
glUnmapBuffer(GL_ARRAY_BUFFER);
```

Это приводит к появлению точек синхронизации CPU/GPU



GPU/CPU synchronization issues

- Buffer orphaning – говорим драйверу, что данные не нужны

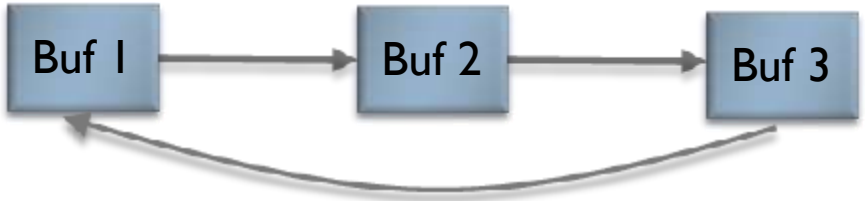
```
glBufferData(GL_ARRAY_BUFFER, size, NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, real_data_ptr);
```

Не работает на графических адаптерах nVidia

- Асинхронный mapping

Нету сихронизации, но данные еще могут использоваться

- Асинхронный mapping + round-robin




```
int id = buf_id[cur = (cur + 1) % 3];
glBindBuffer(GL_ARRAY_BUFFER, id);
void * data = glMapBufferRange(GL_ARRAY_BUFFER, 0, size,
   GL_MAP_UNSYNCHRONIZED_BIT | GL_MAP_WRITE_BIT);
// fill data
glUnmapBuffer(GL_ARRAY_BUFFER);
```

Вопросы?