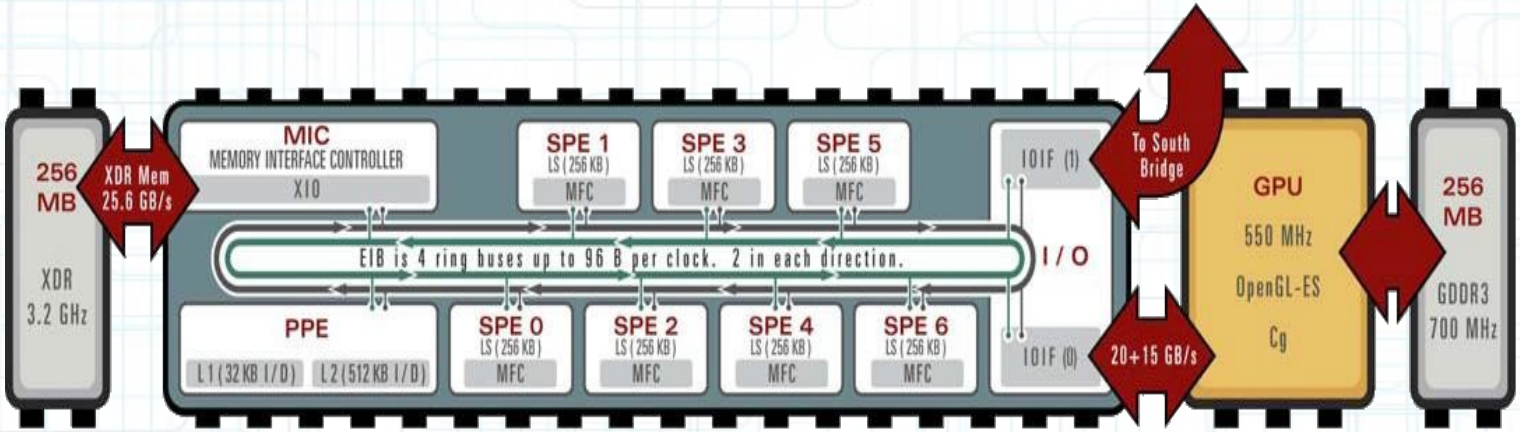
SPU Render



Арсений “Zeux” Капулкин
CREAT Studios
a.kapoulkine@creatstudios.com
http://www.creatstudios.ru/

Введение

- Проект Smash Cars 2
  - Статическая сцена
  - Много динамических объектов
  - Несколько проходов рендера
  - Суммарно до 3000 батчей
- PPU render до 12 ms
  - Таргет – 60 fps :(

Введение

Методы оптимизации

- Ускорение PPU кода
  - Несколько раз уже производилось
  - Идеал (PPU time ~ 0) недостижим
- Статические command buffers
  - Ограниченно применимы
  - Сложности с квалификацией
- Вынос кода на SPU

План

- Структура рендера
- Кратко о SPU
- Портирование
- Развитие
- Вопросы

План

- Структура рендера
- Кратко о SPU
- Портирование
- Развитие
- Вопросы

Рендер – верхний уровень

- На рендер идет набор RenderItem
  - Уже отсортированный и frustum culled
- RenderItem это:
  - SceneNode
  - Material
  - Shader
  - RenderEntity

Рендер – SceneNode

- Узел дерева трансформаций
  - Матрица локальной трансформации
  - Матрица глобальной трансформации
- Матрицы изменяются внешним кодом
  - Анимации
  - Физика
  - Игровая логика

Рендер – Shader

- Алгоритм настройки render pipeline
  - virtual void apply
    - Настраивает auto-параметры
      - Вычисляются автоматически
      - WorldViewProjection, ShadowMap, etc.
  - virtual void setup
    - Настраивает материал
      - Параметры материала (в т.ч. текстуры)
      - Шейдеры
- 99% объектов – тип HWShader

Рендер – Material

- Контейнер данных для Shader
  - Таблица, описывающая layout
    - Имя/тип параметра
    - Смещение в массиве данных
  - Данные
  - Аксессоры по имени/индексу (get/set)
  - Рендер стейты
    - Blend, alpha test, depth, cull

Рендер – RenderEntity

- Алгоритм, производящий отрисовку
  - virtual void render
- Основные реализации
  - RenderStaticGeometry
  - RenderSkinnedGeometry
  - RenderMorphedGeometry
  - DynamicObject

Рендер – нижний уровень

- Кросс-платформенные обертки
  - Установка стейтов с кешированием
  - Установка vertex/pixel констант
  - Установка шейдеров
- Реализация через GCM
  - PS3-specific API для заполнения CB
    - С некоторыми оговорками есть на SPU
      - Это сильно облегчает портирование

План

- Структура рендера
- Кратко о SPU
- Портирование
- Развитие
- Вопросы

SPU – общая информация

- 6 одинаковых ядер
  - 3.2 GHz, in-order, dual-issue
  - 128 векторных регистров
  - Local Storage (LS)
    - 256 Kb - код + данные
    - Латентность 6 тактов
    - Доступ к внешней памяти через DMA
      - Асинхронный memcpy (LS ↔ память)
      - Ограничения на выравнивание и размер

SPU – задачи портирования

- Собрать код под SPU
- Запустить код на SPU
  - Менеджер задач
  - Размер кода/данных
  - Виртуальные функции
- Оптимизация
  - Эффективное использование DMA
  - Оптимизация кода

План

- Структура рендера
- Кратко о SPU
- Портирование
- Развитие
- Вопросы

Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода

Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода

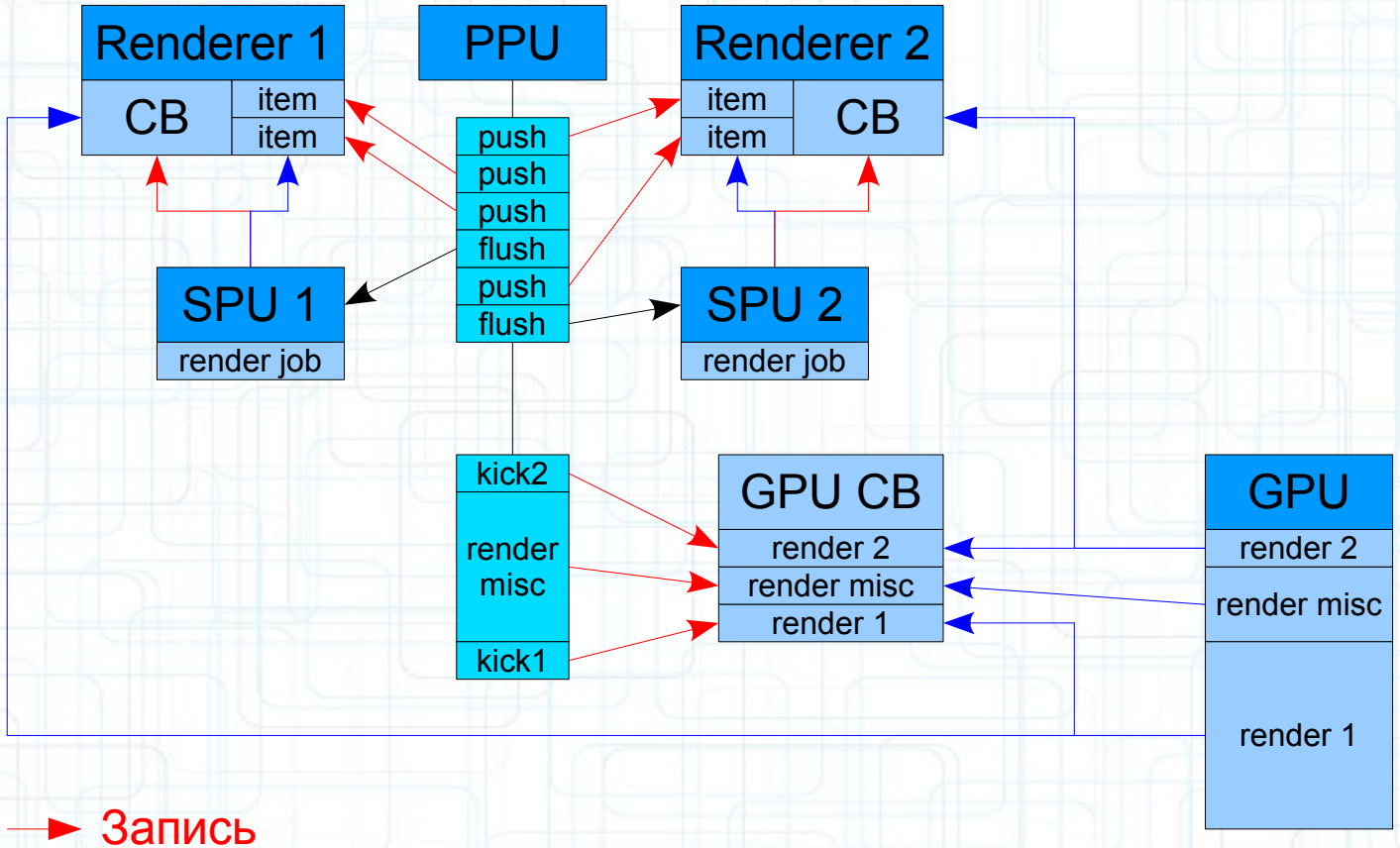
Портирование – этапы

- Этап 1 – работающий прототип
  - Скорость не важна
    - Неоптимальный код, синхронные DMA
  - Обладает полной функциональностью
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода

Этап 1 – PPU интерфейс

- async::Renderer
  - Простой интерфейс
    - push(RenderItem)
    - flush()
    - kick()
  - При создании задаются лимиты
    - Максимальное число item-ов
    - Максимальный размер CB
  - Double-buffering для CB

Этап 1 – PPU интерфейс



→ Запись

→ Чтение

Этап 1 – DMA helpers

- Набор удобных функций для DMA
  - Allocator
    - Обычный стековый аллокатор, ptr += size
  - fetchData(ea, size)
    - Аллокация памяти и синхронный DMA
    - Умеет обрабатывать некорректный align
  - fetchObject / fetchObjectArray
    - Типизированная надстройка над fetchData
  - В дальнейшем - асинхронные варианты

Этап 1 – DMA helpers

- void* fetchData(alloc, ea, size)

```
uint32_t sizeAligned = (size + (ea & 15) + 15) & ~15;
void* ls = alloc.allocate(sizeAligned);
DmaGet(ls, ea & ~15, sizeAligned);
DmaWait();
return (char*)ls + (ea & 15);
```

- T* fetchObject(alloc, ea)

```
return (T*)fetchData(alloc, ea, sizeof(T));
```

Этап 1 – виртуальность

- vfptr с PPU на SPU не имеет смысла
- Решение разное для разных классов
  - Shader
    - Один тип шейдеров – HWShader
  - RenderEntity
    - Enum из всех поддерживаемых типов
    - Значение типа хранится в указателе
      - ptr = actual_ptr | type // actual_ptr % 4 == 0

Этап 1 – инкапсуляция

- Затрудняет портирование
  - Методы с некорректным для SPU кодом
    - CRT_ASSERT(next->prev == this)
  - Дополнительные параметры методов
    - render() → render(Context)
- Затрудняет рефакторинг SPU кода
- Решение (нравится не всем...)
  - #define private public [SPU-only!]

Этап 1 – shader patch

- На RSX нет константных регистров PS
  - Константы зашиты в микрокод
  - Микрокод приходится патчить
    - RSX blitting
      - Большой RSX cost (до 50% frame time)
    - PPU render
      - Инстансы хранятся в ring buffer
      - Сложная синхронизация
    - SPU render
      - Инстансы хранятся в том же буфере, в котором хранятся команды для RSX

Этап 1 – синхронизация

- PPU/SPU
  - Конфликты по read/write данным
    - Матрицы трансформации
    - Параметры материалов
  - Объекты могут быть удалены
  - Решение
    - SPU код должен быть быстрым
    - PPU ждет SPU перед изменением данных

Этап 1 – синхронизация

- SPU/RSX
  - PPU
    - flush() ставит в начало своего буфера JTS
      - Jump-To-Self: loop: goto loop
    - kick() ставит в главный буфер CALL
  - SPU
    - Заполняет свой буфер командами
    - В конце дописывает RET
    - Меняет JTS на NOP*

Этап 1 – результаты

- Время на портирование – 3 дня
- Время рендера – 25 ms
  - На PPU та же сцена рисуется за 12.5 ms
  - Варианты ускорения
    - Brute-force – разбить очередь на 5 кусков
      - 5 ms на 5 SPU
    - Оптимизировать код
- Отдельная ветка кода
  - Общие структуры данных

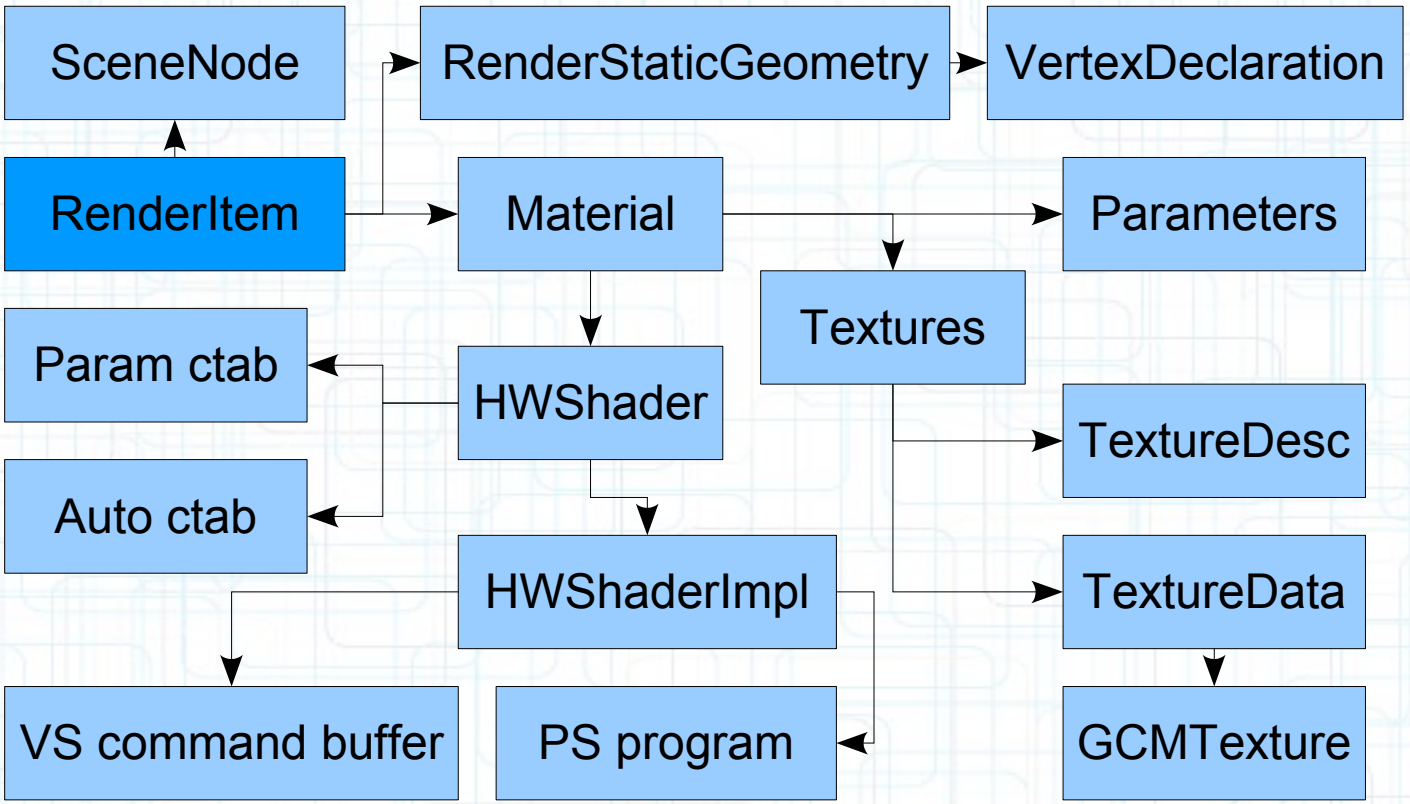
Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода

Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
  - Переукладка данных
    - Уменьшение числа индирекций
  - Асинхронные DMA
    - Double-buffering input/output данных
- Этап 3 – оптимизация кода

Этап 2 – memory layout



Этап 2 – укладка данных

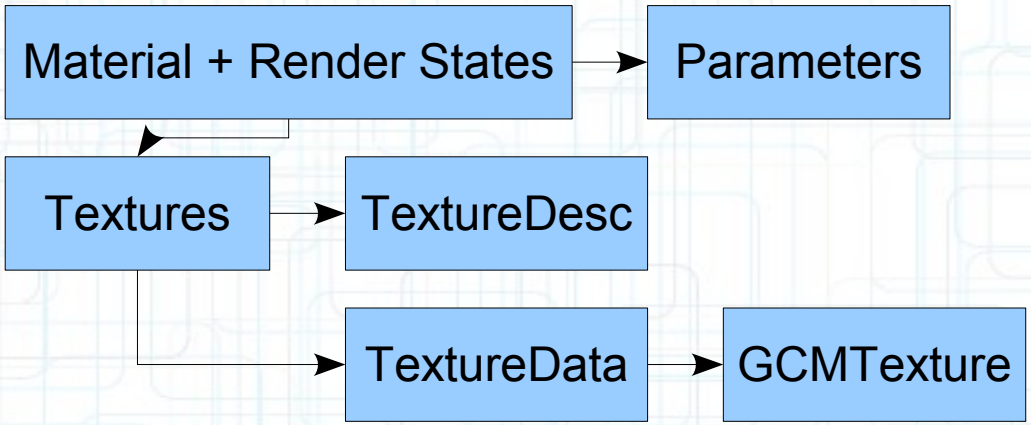
- Цель – уменьшить число индирекций
  - Точнее, уменьшить длину путей в графе
- Откуда они возникают?
  - Shared данные
  - Массивы “переменной” длины
    - Размер известен в load time
  - “Правильная” архитектура
    - Law of Demeter
  - Pimpl

Этап 2 – материалы

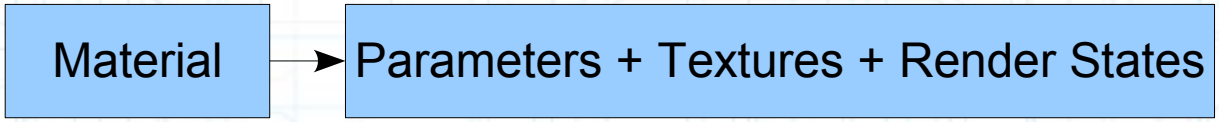
- Текстуры
  - struct TextureInfo
    - Хранится в массиве параметров
    - Обновляется в момент setValue
    - Содержимого достаточно для setup-а
      - 4 байта – sampler state, 12 байт - header
- Render States
  - Перемещаются в массив параметров
    - 16 байт на все стейты

Этап 2 – материалы

- Было:



- Стало:

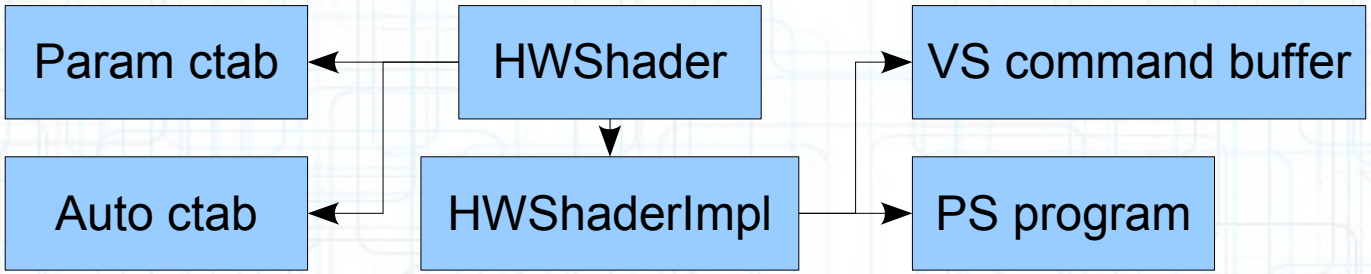


Этап 2 – HWShaderImpl

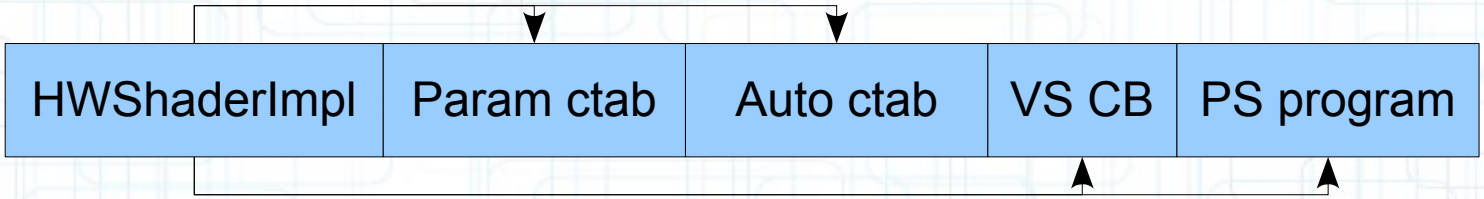
- Много массивов переменной длины
  - Таблицы констант
  - Данные шейдеров
- Решение
  - Последовательная укладка всех данных в памяти
  - Заголовок содержит смещения
  - DMA блока данных и fixup указателей
    - vsCB = (char*)impl + impl->vsCBOffset

Этап 2 – HWShaderImpl

- Было:



- Стало:



Этап 2 – VertexDeclaration

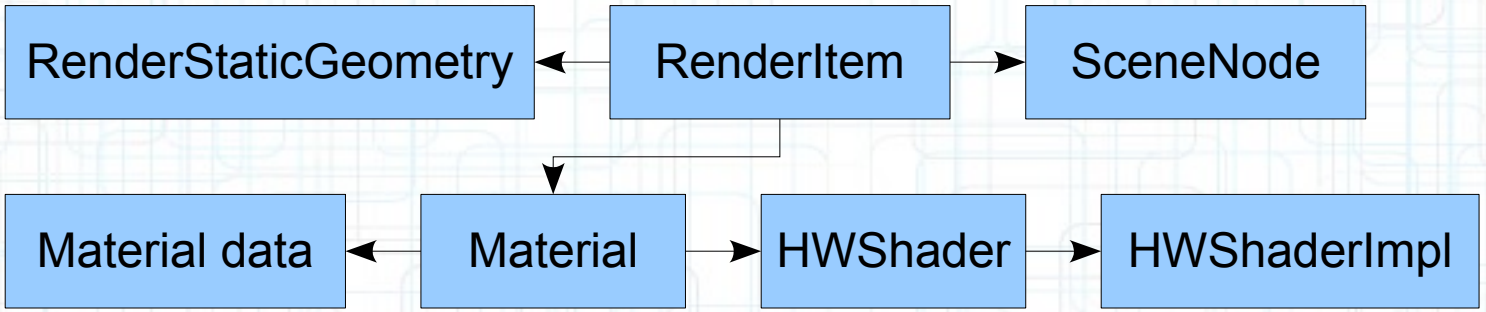
- class RenderStaticGeometry
  - VertexDeclaration* vdecl
    - Можно хранить vdecl по значению
      - Пенальти по памяти
- Уникальных указателей мало
  - Декларации кешируются при создании
  - Можно сделать software cache!
    - Кеш на 4 элемента, DMA stall при промахе
    - 30 cache misses на 3500 batches

Этап 2 – FlatRenderItem

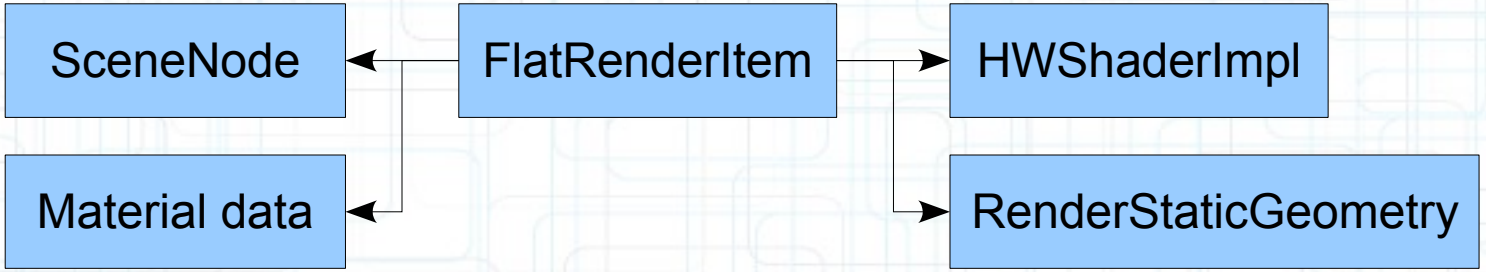
- Путь к HWShaderImpl
  - item->material->shader->impl
- FlatRenderItem
  - Кеширует указатели/размеры
    - Material data EA/size
    - Shader impl EA/size
    - Scene node/render entity EA
  - Создается при загрузке уровня

Этап 2 – FlatRenderItem

- Было:



- Стало:



Этап 2 – оптимизация DMA

- До сих пор все DMA – синхронные
- Латентность DMA можно скрывать!
  - Отправлять несколько запросов
    - Ждать один раз
  - Двойная буферизация
    - Пока обрабатывается текущий batch
      - Читаются данные для следующего
      - Пишутся данные для предыдущего
    - Требует дополнительной памяти LS
      - В нашем случае – не проблема

Этап 2 – output DMA

- Command buffer
  - Два буфера по 8 Kb
  - Переключение при переполнении
- Shader buffer
  - Можно делать double buffer
  - Проще подождать окончания трансфера
    - Но не после заполнения, а перед!
    - DmaPut успевает завершиться вовремя

Этап 2 – input DMA

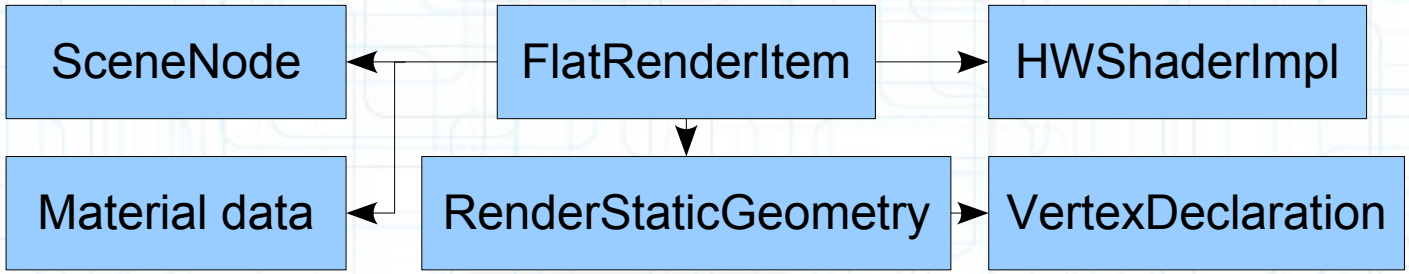
- Для каждого батча
  - Ожидание трансферов
  - Получение следующего батча
    - 4 DmaGet подряд
  - Обработка текущего батча
- Требует пролога перед циклом
  - Получение данных первого батча

Этап 2 – input DMA

- Сложный код
  - FlatRenderItem фетчатся по одному
    - Проще было бы фетчить группами
- Баги
  - Код префетчит item после последнего
    - PPU дублирует последний item
  - Последние DmaGet нужно подождать!
    - Иначе – возможный stack corruption

Этап 2 – результаты

- Время на оптимизацию – 3 дня
- Время рендера – 8 ms
  - Без double buffering – 12 ms
- PPU time не изменился



Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода

Портирование – этапы

- Этап 1 – работающий прототип
- Этап 2 – оптимизация данных
- Этап 3 – оптимизация кода
  - Профилирование
    - SN Tuner
    - SPUsim
  - Оптимизация

Этап 3 – SN Tuner

- CPU/GPU профайлер для PS3
  - SPU performance counters
    - Ожидание DMA
    - Спаривание инструкций
      - Показывает общее качество кода
  - SPU PC sampling
    - В отличие от PPU, это бесплатно
    - Показывает общую картину
      - Приходилось убирать inline с функций

Этап 3 – SPUsim

- Эмулятор SPU на PC
  - Полезно для прототипирования
    - Очень быстрые итерации
    - Статистика по stalls
    - Instruction trace
      - Показывает stalls, отсутствие спаривания
  - Для небольших self-contained функций
    - Настроить DMA можно, но неудобно

Этап 3 – ветвления

- Ветвления дорогие
- Уменьшение числа ветвлений
  - Branch flattening
  - Loop unrolling
  - Switch → function pointer table
- Zero-size DMA
- Branch hinting

Этап 3 – работа с LS

- Запись/чтение из LS только по 16 байт
  - Компилятор делает shuffle / masking
- Чтение по 16 байт
  - Padding входных структур
  - Loop unrolling
- Запись по 16 байт
  - Запись нескольких команд RSX за раз
  - Padding с помощью NOP

Этап 3 – результаты

- Время на оптимизацию – 5 дней
- Время рендера – 2 ms
- Дальнейшие оптимизации
  - Оптимизация кода возможна
    - Но уже не стоит затраченных усилий
  - Параллельный рендеринг на N SPU
    - Разные части сцены
    - Разные проходы

Портирование – результаты

- PPU time – 12.5 ms
- SPU time (прототип) – 25 ms (3 дня)
- SPU time (укладка) – 12 ms (2.5 дня)
- SPU time (async DMA) – 8 ms (0.5 дня)
- SPU time (код) – 2 ms (5 дней)
- 75 Kb SPU кода, 20 Kb PPU кода
  - Сейчас – 105 / 26 Kb

План

- Структура рендера
- Кратко о SPU
- Портирование
- Развитие
- Вопросы

Развитие

- Реализованное
  - Сортировка батчей
  - Culling (frustum, размер на экране)
  - Установка игровых параметров
- Future work
  - Occlusion culling
  - Single buffered context
  - Uber shaders

Вопросы

?

Арсений “Zeux” Капулкин
CREAT Studios
a.kapoulkine@creatstudios.com
http://www.creatstudios.ru/