creat studios

Эффективная система анимаций для Nintendo DS

Ростислав Хлебников
Евгений Заякин

20 апреля 2008

КРИ

creat studios

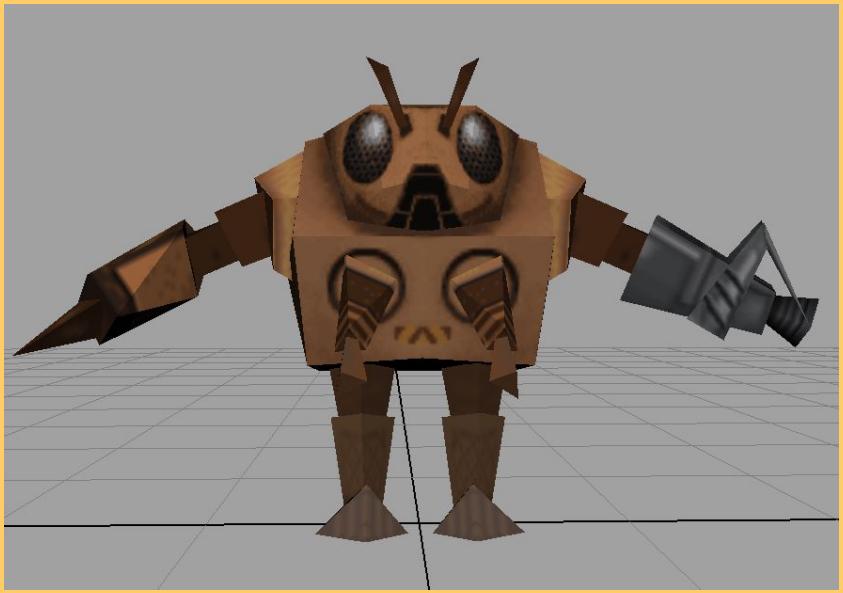
Содержание

- Постановка задачи
- Эволюция анимационной системы
  - Проблемы и решения
- Выводы (NDS animation best practices)

КРИ

Постановка задачи **(1)**

300 вершин
40 костей



200 вершин
25 костей

- Суммарно ~4000 кадров (около 2 минут) анимаций

Постановка задачи **(2)**

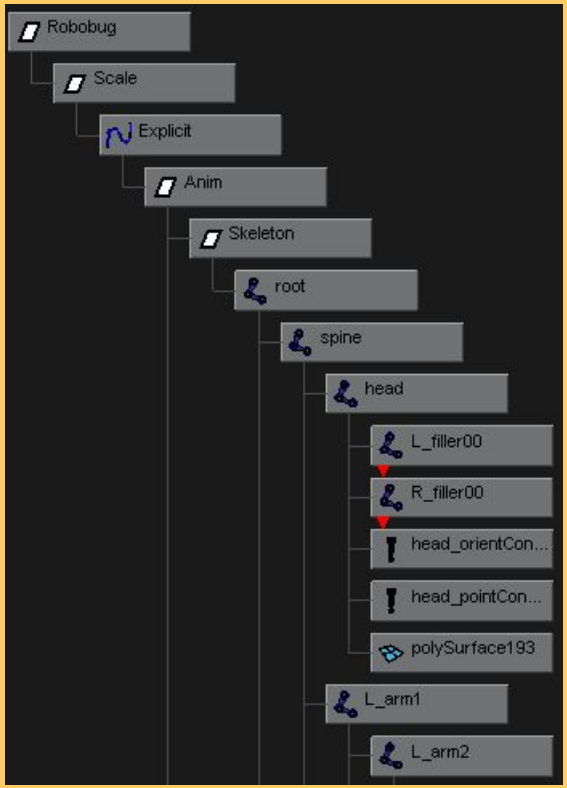
- Анимации на DS
  - Анимация на уровне матриц
  - Анимация на уровне геометрии
- Чтобы было быстро и красиво
  - На DS – учесть слабые и сильные стороны платформы

Существующее решение **(1)**

- Процесс получения результата
  - Сплайны
    - Sampling сплайнов и создание матрицы по TRS
  - Матрицы local-to-parent
    - Перемножение матриц в иерархии
  - Матрицы local-to-world
    - Скиннинг
  - Результирующий меш

Существующее решение **(2)**

Исходные данные



+

КРИ

Существующее решение: итоги

- Плюсы
  - Алгоритмы написаны и много раз проверены
- Минусы
  - Низкая производительность
    - Кроме того, нет возможности распараллелить работу

Картридж **NDS**

- Плюсы
  - Высокая скорость чтения данных (5 Мб/с)
  - Zero seek time
- Минусы
  - Сравнительно небольшой объем (максимум 128 Мб, а желательно уложиться в 64 Мб)

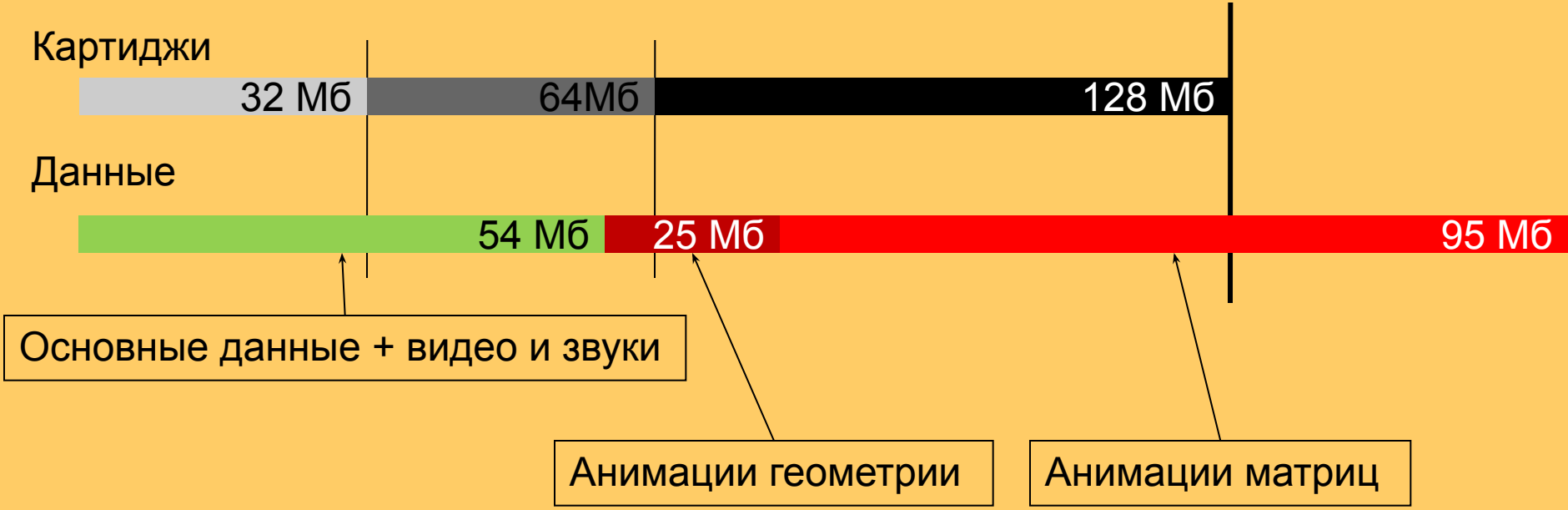
Используем сильные стороны: идея

- Preprocessing (PC)
  - Предрассчитываем данные для каждого кадра:
    - Матрицы local-to-parent
    - Скинованная геометрия
- Runtime (NDS)
  - Считываем необходимые данные
  - Считаем матрицы local-to-world

Первый вариант реализации

- Preprocessing
  - Частота дискретизации - 30 fps
  - Формат записи:
    - Матрицы - 16 значений fixed-32 (по 4 байта)
    - Геометрия (дисплей-листы для непосредственной отсылки на рендеринг)
- Runtime
  - Дополнительный буфер для матриц
  - Для дисплей листов дополнительной памяти не требуется

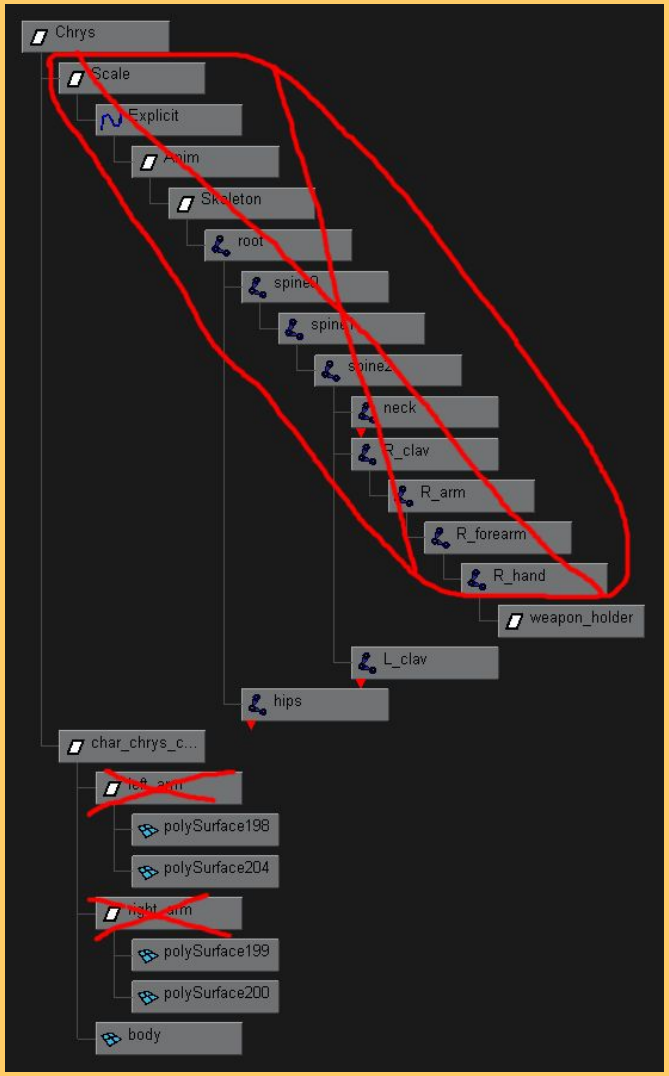
Размеры анимаций на картридже огромны



Удаляем ненужные узлы в иерархии

- Узел является нужным, если:
  1. С ним связана геометрия
  2. Его позиция требуется игре (например, позиция локатора для оружия)
- Все остальные узлы являются лишними!

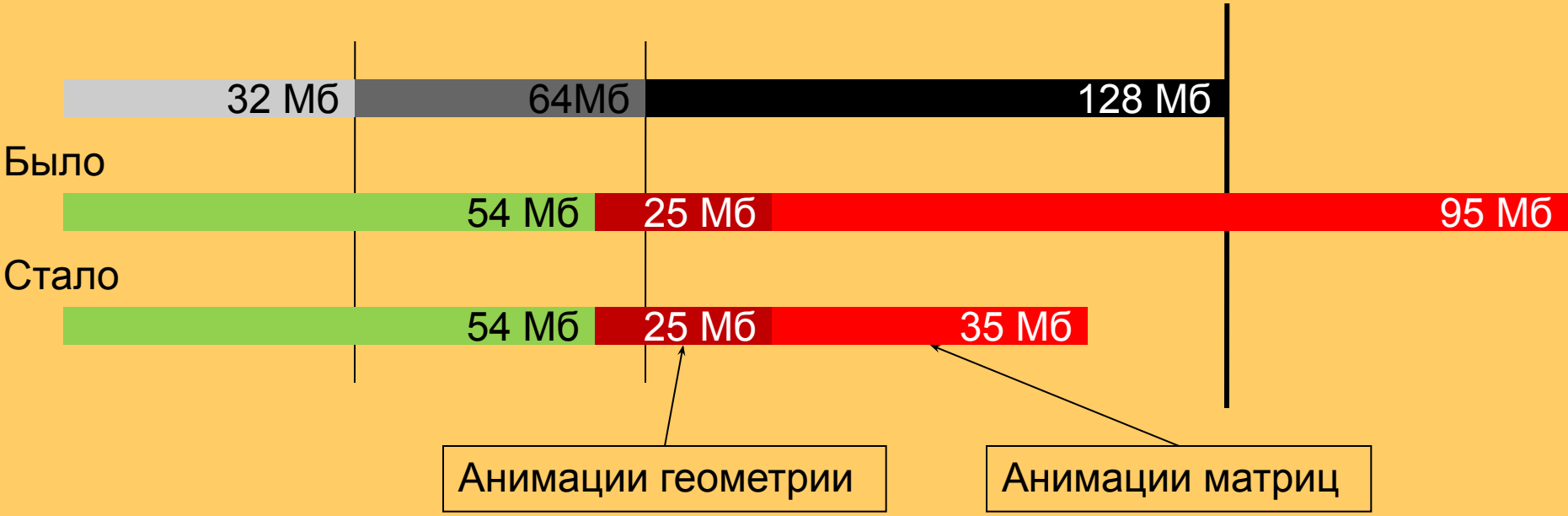
Ненужные узлы



- Ненужными оказываются 30 – 90 % узлов!

Удаляем ненужные узлы в иерархии: результат

- Объем уменьшили на 60%
- Работает ощутимо быстрее

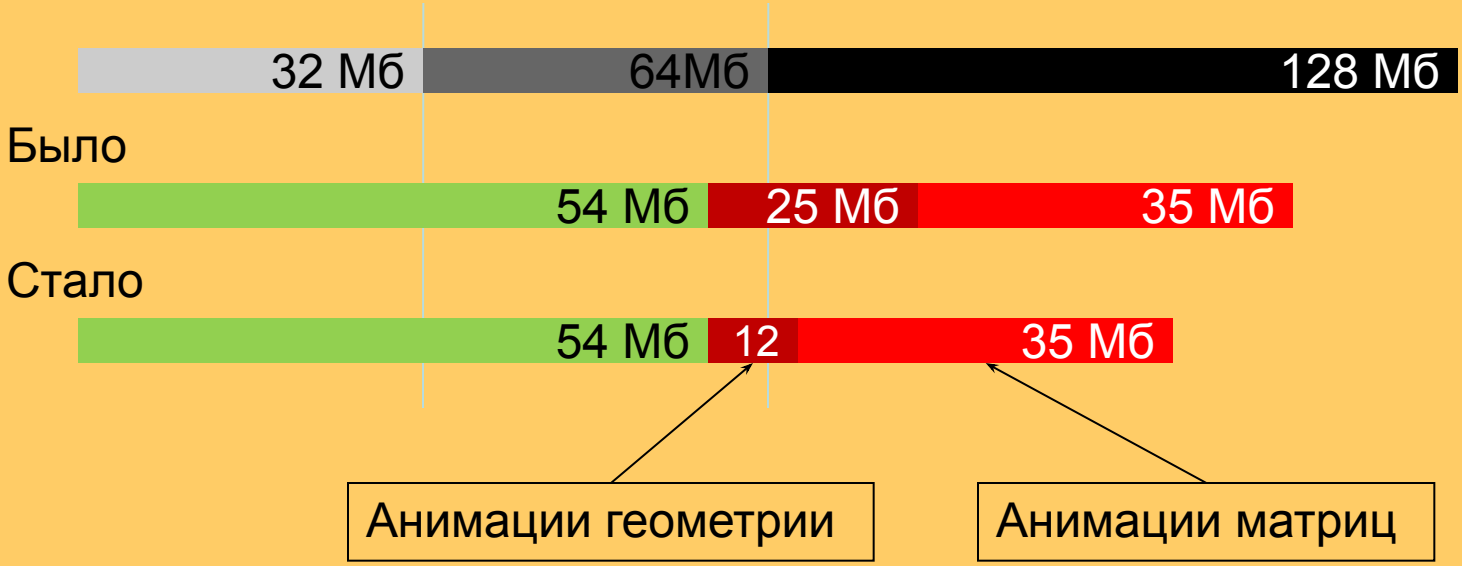


Избавляемся от дубликации в дисплей-листах

- Материалы, флаги, UV сохраняем единожды
- Вершины и нормали - для каждого кадра

Удаление дубликации: результат

- Объем уменьшили на 50%
- Требуется дополнительная оперативная память
  - Информация о размещении динамических данных
  - Буфер для считывания анимированной геометрии

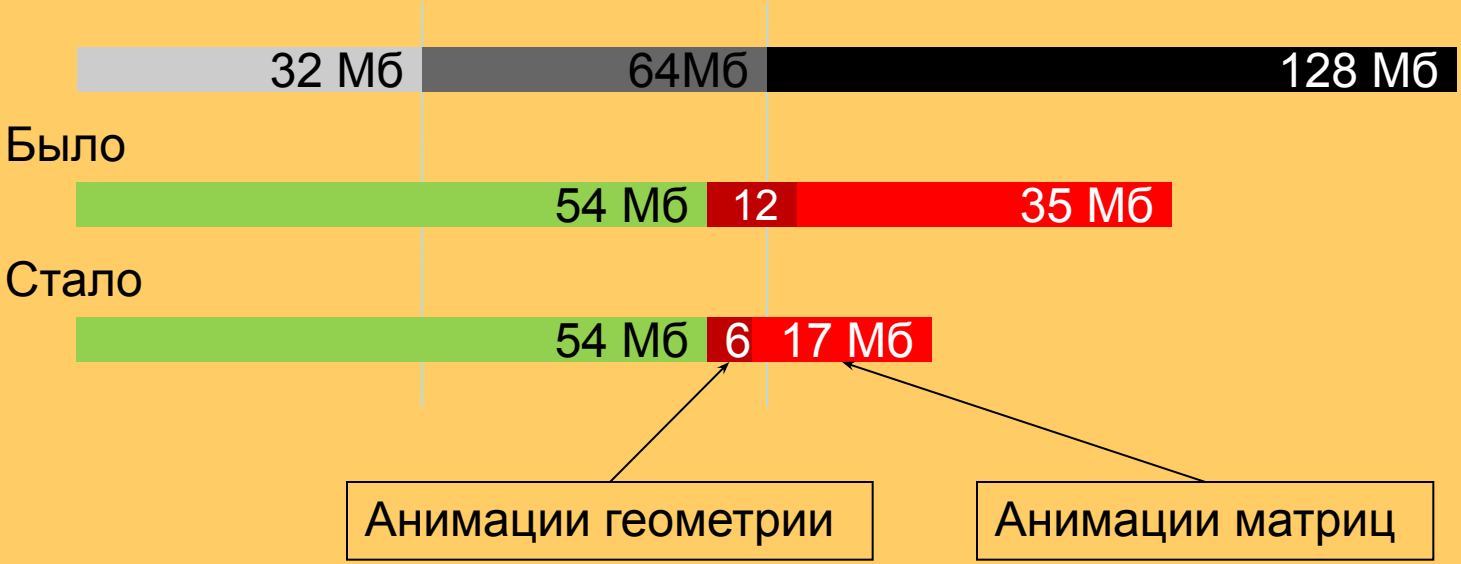


Что еще можно уменьшить?

- Можем изменить только частоту квантования
  - Размер данных = размер кадра * частота квантования * длина анимации
- Уменьшаем ее вдвое

Уменьшение частоты квантования: результат

- Точно влезем в 128 Мб!
- Анимации стали «дерганными»



Делаем анимации плавнее

- Идея: интерполяция между ключевыми кадрами
- Классическое решение (основанное на интерполяции сплайнов) не подходит!
- Наше решение:
  - Интерполяция матриц
  - Морфинг геометрии

Варианты интерполяции матриц

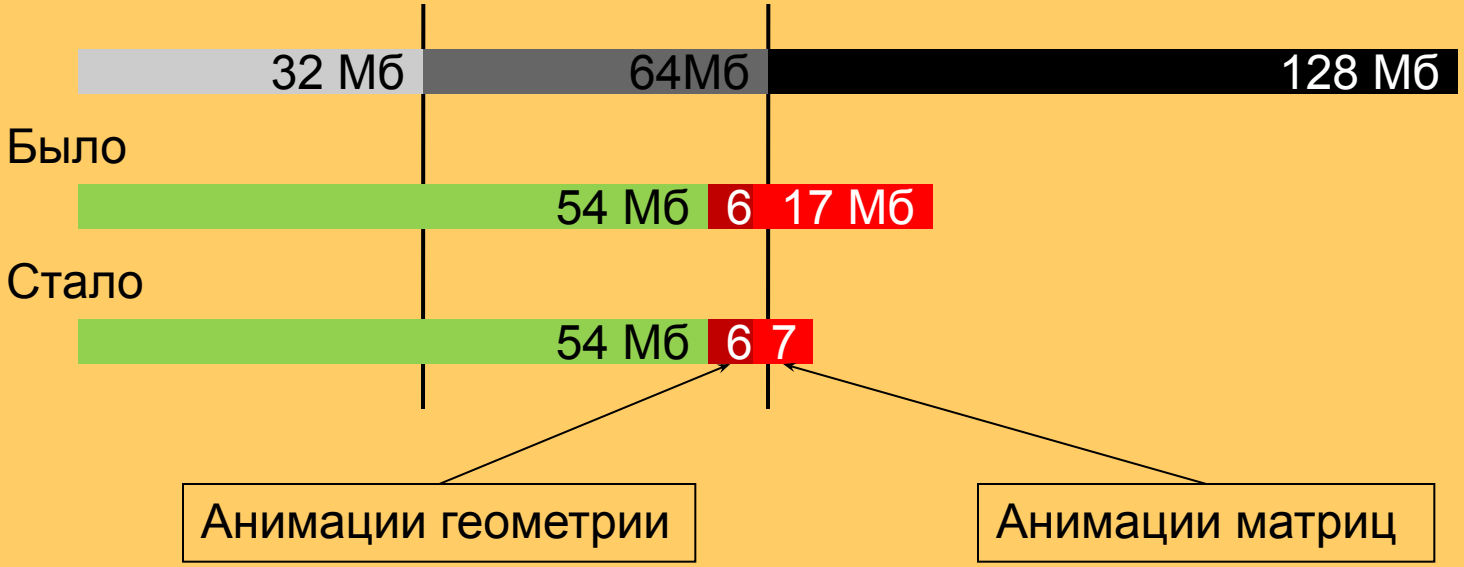
- Интерполяция координатных осей
  - \+ Простое решение
  - – Требуется ортогонализация
  - – Проблемы с масштабированием
- Интерполяция с помощью кватернионов
  - \+ Качественная интерполяция
  - – Двойное преобразование представлений
  - – Специальный код для масштабирования

Интерполируем матрицы

- Используем кватернионы
- Избавляемся от двойного преобразования
  - На экспорте сохраняем по-отдельности:
    1. Кватернионы
    2. Компоненты Translate и Scale
- Используем линейную, а не сферическую интерполяцию кватернионов

Интерполяция матриц: результат

- Матричные анимации стали плавными
- Объем уменьшили на 60%
  - 26 байт вместо 64 на один узел на кадр

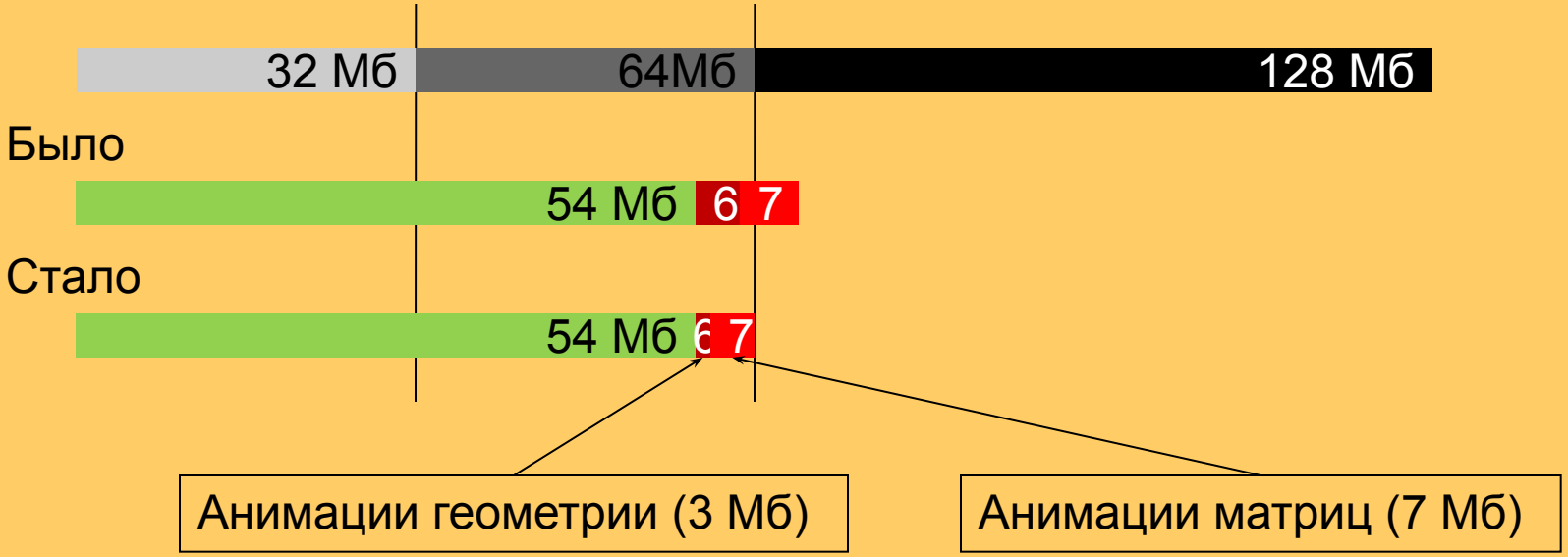


- Необходимые действия
  - Распаковка из формата графического ядра
  - Линейная интерполяция векторов
  - Обратная упаковка

- Используем возможность упаковывать три компоненты в 32 бита (vtx10)

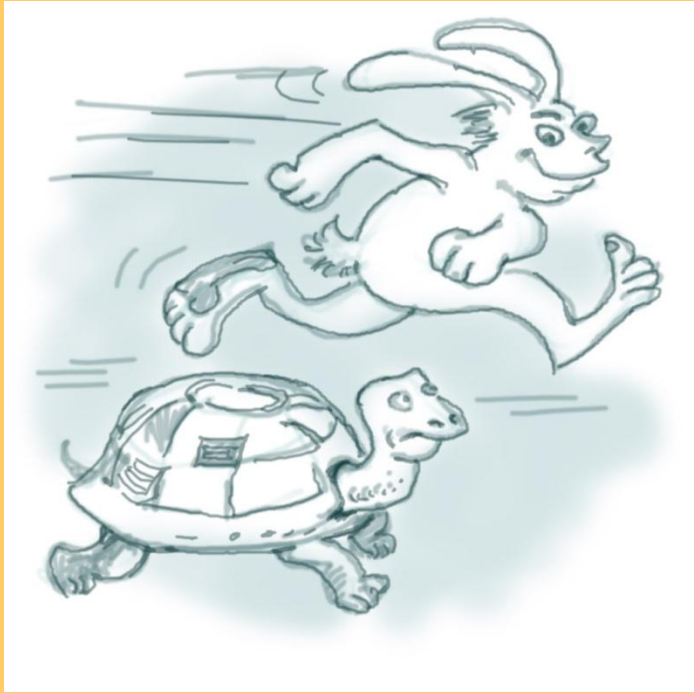
Интерполяция геометрии: результат

- Качество скиновой анимации улучшилось
- Сэкономили еще 50% памяти



Проблемы не закончились!

- Зверские тормоза
- Визуальные артефакты



Причины тормозов

- Компилятор генерирует **чрезвычайно плохой** ассемблерный код
  - Основная проблема - код внутренних циклов интерполяции
- Погружаемся в ARM ARM и ассемблируем вручную

Ручное ассемблирование: что и как?

- Ассемблируем функции интерполяции
  - *s16Lerp / s32Lerp / getDotSign*
  - *lerpVtx10* (хит сезона)
- Используем специфические команды ARM, например *SMLABB*
- Решаем проблемы компилятора, который совершенно неспособен на анализ asm-секций

Ручное ассемблирование: результат

- Скорость стала такой, что стали думать переписывать на ассемблере всю игру

Визуальные артефакты, борьба

- Причина дрожания – потеря точности из-за формата vtx10
- Решение
  - Для персонажей, на которых эффект заметен возвращаем точность (таких оказалось очень немного)

Результаты

Дальнейшие улучшения

- Уменьшить размер данных и увеличить скорость за счет неравномерного квантования анимаций
  - Автоматический выбор ключевых кадров

    и / или

  - Превью-плагин в Maya – WYSIWYG для художников

Выводы

- Используйте streaming и интерполяцию ключевых кадров вместо просчета в run-time
- Сокращайте иерархии до минимума
- Активное используйте ручное ассемблирования time-critical кода
- Будьте аккуратны с точностью

Вопросы?