МОДЕЛИРОВАНИЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМ

УДК 004.94

С.Н. Верзунов, Д.А. Токсаитов
Институт автоматики и информационных технологий НАН КР, г. Бишкек
verzunov@hotmail.com, toksaitov_d@auca.kg

СПОСОБЫ ОПТИМИЗАЦИИ РАСЧЁТА 3D ГРАФИКИ ДЛЯ ТАЙЛОВЫХ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ НА ПРИМЕРЕ ВИЗУАЛИЗАЦИИ МОДЕЛИРОВАНИЯ ГРАВИТАЦИОННОГО ВЗАИМОДЕЙСТВИЯ N ТЕЛ

Визуализация физических явлений, таких как, например, гравитационных взаимодействий небесных тел, требует прорисовки большого количества однотипных 2D и 3D объектов. К сожалению, тайловые архитектуры графических ускорителей устройств отложенной визуализации, популярные на мобильном рынке, не предназначены для обработки большого количества запросов на рисование, поэтому необходимо использовать различные методы оптимизации для уменьшения количества таких запросов. В настоящей работе представлена реализация графической системы визуализации, которая поддерживает дерево изменений состояния каждого объекта сцены, для передачи только необходимого набора данных для рисования. Кроме того, система минимизирует количество объектов для передачи, динамически объединяя и/или дублируя их геометрию. Система также разгружает шину данных, вычисляя и передавая только разницу между конечными сетками геометрии. Прирост производительности относительно неоптимизированной версии продемонстрирован путем сравнительной оценки скорости работы систем на примере визуализации результатов моделирования гравитационного взаимодействия большого количества тел.

***Ключевые слова*:** тайловые процессоры; TBDR архитектура; графический движок; дерево обновлений; комбинирование геометрии; дублирование геометрии; разница конечной сетки% задача N тел.

Введение

Визуализация физических явлений, таких как, например, гравитационных взаимодействий небесных тел, требует прорисовки большого количества однотипных 2D и 3D объектов. Такая необходимость возникает также при визуализации результатов обработки временных рядов, например при построении результатов вейвлет-преобразования временного ряда в виде двухмерной или трехмерной скалограммы [1]. Современные мобильные устройства, такие как мобильные телефоны, ноутбуки и планшетные компьютеры, содержат программные интерфейсы графического адаптера, перенесенных с персональных компьютеров. Такие библиотеки не имеют представления об особенностях мобильного графического ускорителя, который кардинально отличается архитектурой от своего настольного собрата. Микросхемы мобильной графики построены на основе архитектуры отложенного расчёта и обрабатывают изображение отдельными регионами фиксированного размера (тайлами) для минимизации потребления электроэнергии устройства [2, 3]. Архитектуре отложенного расчёта требуется, чтобы центральный процессор подготовил информацию обо всех запросах на рисование объектов сцены в буфере для проверки данных, распределения информации на

тайлы и передачи позже, целиком в память графического ускорителя. Такой подход взаимодействия между процессором и графическим ускорителем позволяет, с одной стороны, упростить схему мобильного графического процессорного устройства (ГПУ), минимизировать его потребление памяти в системе и уменьшить потребление энергии устройства [3]. С другой стороны, этот подход требует выполнять дополнительную работу по сбору данных и поддержке буферов для каждого последующего кадра. Это накладывает серьезные ограничения на максимальное количество объектов, которые могут быть представлены на экране одновременно из-за ограниченного размера вышеупомянутых буферов [3]. Отложенные системы требуют от разработчиков тратить драгоценное время процессора на поддержку графической подсистемы, а не на выполнение полезной работы их программ.

В настоящей работе представлены результаты исследования разработки системы расчёта графики в реальном времени, оптимизированной для работы на тайловых мобильных графических процессорах с архитектурой отложенного расчёта. Система минимизирует затраты на передачу данных между центральным процессором и графическим ускорителем путём создания и поддержки дерева изменений состояния для каждого объекта сцены, объединением геометрии на центральном процессоре, дублированием геометрии на графическом процессоре и минимизацией передачи данных путем нахождения разницы конечной сетки геометрии между двумя последующими кадрами. Вышеупомянутая система была разработана и оптимизирована для создания системы визуализации на мобильных устройствах и браузерах результатов моделирования гравитационного взаимодействия между большим количеством небесных тел.

Обзор аппаратных систем

Рынок мобильных платформ ныне представлен различными вариациями тайловых процессоров. Идея тайлового ГПУ (рис. 1), в отличие от популярной архитектуры немедленного рисования настольных систем, показанной на рис. 1, заключается в том, что графический буфер изображения разделяется на регионы одинакового размера (тайлы) [2–5].

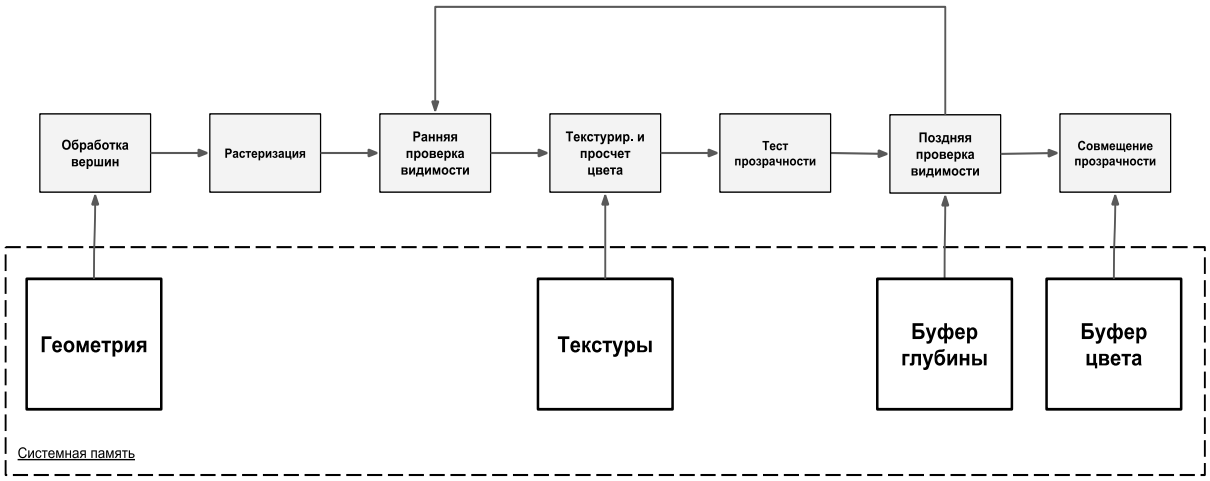

Рисунок 1. Конвейер архитектуры ГПУ немедленного режима расчёта

Большинство реализаций процессоров используют размер 16 на 16 и 32 на 32 пикселей для каждого тайла (рис. 2). Тайловое ГПУ имеет схемотехническое решение

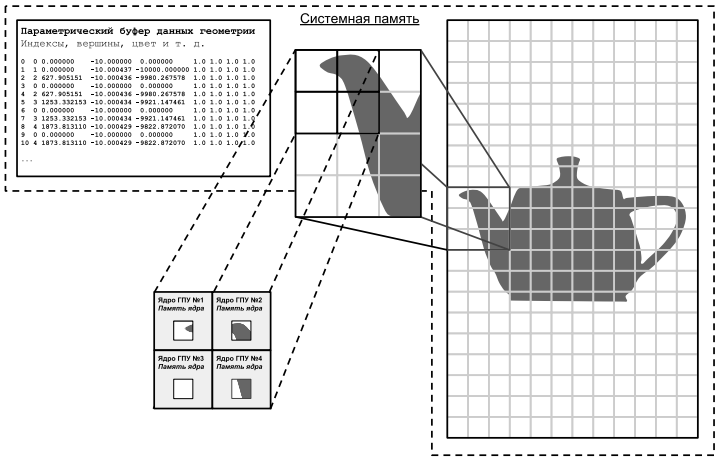

Рисунок 2. Общая схема работы ГПУ с тайловой архитектурой

(рис. 3), которое может использовать специально подготовленный буфер геометрии для просчета графики только в пределах одного тайла. Из-за небольшого размера тайла такое аппаратное решение может уместить все промежуточные данные и данные просчитанной части изображения в регистровой памяти ГПУ.

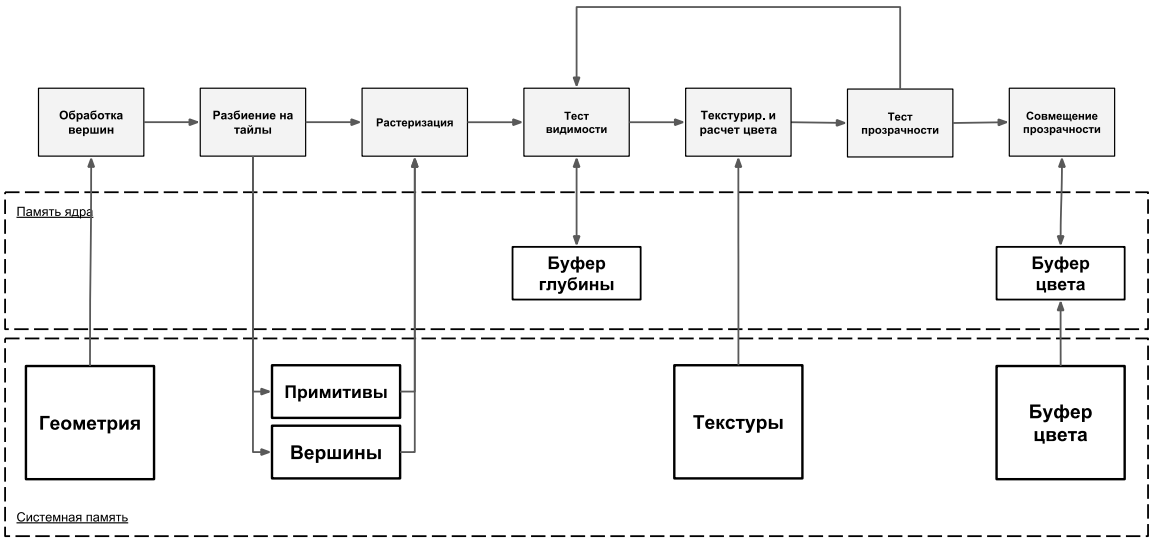

Рисунок 1. Конвейер архитектуры тайлового ГПУ

Это позволяет ускорить расчет, минимизировать энергопотребление, упростить схемотехническое решение и получить горизонтально-масштабируемую систему, проводя обработку нескольких тайлов одновременно несколькими ядрами графического процессора [3, 6].

На базе тайловой архитектуры ключевые игроки рынка стали вводить модификации для последующего увеличения производительности. Британская компания Imagination Tech PowerVR™, которая занималась производством ГПУ для мобильных продуктов компании Apple, использовала архитектуру отложенного рисования. Архитектура отложенного рисования (TBDR, Tile-Based-Deferred-Rendering, рис. 4) также позволяет проводить сбор всех данных объектов сцены в специальном параметрическом буфере [3]. Но после сбора информации или при переполнении буфера система командует ГПУ нарисовать все объекты за один проход. Информация обо всех или большей части объ-

ектов сцены позволяет тайловому ГПУ произвести более эффективную сегментацию данных между тайлами, тем самым значительно ускорить работу.

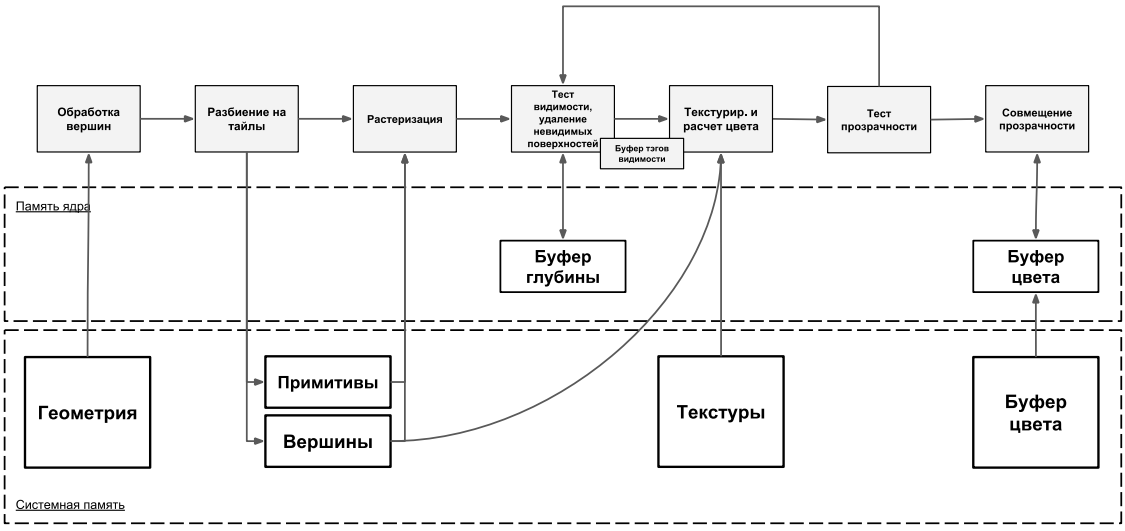

Рисунок 2. Конвейер архитектуры тайлового ГПУ с отложенной системой рисования

В частности, специализированная часть ядра может удалить все полигоны геометрии, которые были перекрыты полигонами других объектов сцены стоящими ближе к виртуальной камере. Это позволит проводить дорогую операцию просчета цвета для непрозрачных объектов всего лишь один раз для каждого фрагмента (пикселя) многоугольника.

Несмотря на многие преимущества тайловых процессоров отложенного рисования, они имеют ряд существенных недостатков. Одним из них является ограничение на количество объектов рисования, вследствие наличия фиксированного размера параметрического буфера [6]. Другим ограничением являются высокие временные затраты на подготовку и проверку данных этого буфера, которую необходимо проводить на центральном процессоре перед каждым просчетом кадра [5]. Это еще более усугубляет проблему рисования большого количества объектов на тайловых архитектурах. Независимо от сложности 3D модели, будь это всего один треугольник, попытка нарисовать сцену в 1000 объектов, совершая 1000 отдельных запросов на рисование без потери производительности практически невозможна в ГПУ такой архитектуры. Другая проблема отложенного рисования заключается в обработке прозрачных объектов. Тайловая архитектура создавалась для ускорения общего случая, когда больше половины объектов сцены непрозрачны. К сожалению, специализированный конвейер этой архитектуры не приводит к ускорению, а иногда и замедляет процесс рисования объектов, при наличии прозрачности. По рекомендации дизайнеров системы, для минимизации потери скорости обработки данных необходимо передавать запросы на рисование в определенном порядке. Сначала нужно нарисовать все непрозрачные объекты, а после рисовать прозрачные, в порядке от дальних к самым близким, исходя из позиции виртуальной камеры [3].

Архитектура системы расчёта 3D графики

В качестве известного примера, где необходим расчёт сложных 3D моделей, можно привести задачу визуализации гравитационного взаимодействия небесных тел. Необходимо рисовать схематическое представление двумерных и трехмерных примитивов вроде прямоугольников, окружностей, сфер и параллелепипедов. Для моделирования частей галактики или силовых полей, системе расчёта графики (т.е. графическому движку) необходимо нарисовать от 10 тысяч до 1 миллиона примитивных объектов.

Все объекты системы меняют свою позицию каждый кадр, что требует повторной передачи данных при каждом запросе на рисование. Первые версии графического движка имели классическую архитектуру, применяемую во многих популярных решениях, например, в такой библиотеке, как SceneKit от компании Apple.

Реализация первой версии (рис. 5) состояла из двух ключевых частей: базы данных объектов сцены и системы передачи информации базы данных на ГПУ. База данных объектов сцены представляла собой дерево объектов, где отношение родителя и дочернего узла состояло из отношения локальных матриц трансформации. Перемножение матрицы каждого дочернего узла с матрицами предков позволяло получить финальную трансформацию мира 3D сцены. Отношение матриц позволяло соединить несколько объектов сцены вместе и, трансформируя корневой объект, получать верную позицию и ориентацию, подчиненного относительно изменения положения и поворота, корневого пространства сцены. В то же время, задачи системы передачи данных заключались в том, чтобы проходить по всем узлам дерева объектов, рассчитывать финальную матрицу, передавать всю информацию объекта на ГПУ и давать команду на его рисование.

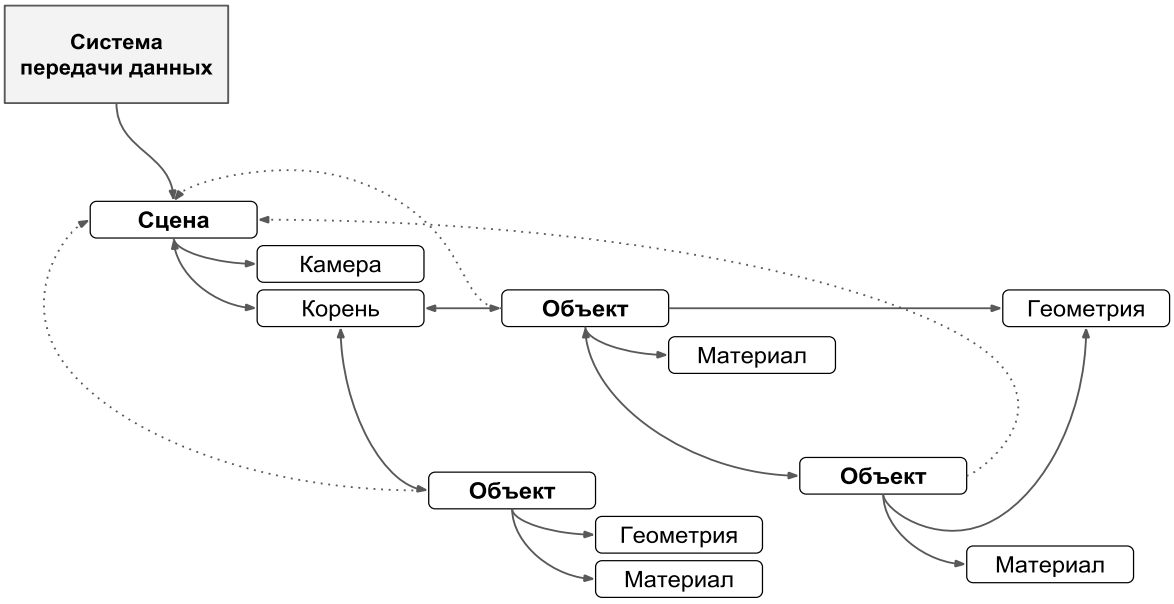

Рисунок 5. Дерево сцены (база данных состояний) и система передачи данных на ГПУ неоптимизированной версии графического движка

Замеры производительности первой версии системы на тестовых устройствах (мобильные телефоны iPhone 6, iPhone 6s, iPhone 7) показали, что такой подход может просчитывать до ~800 объектов на ГПУ, что очень мало для поставленной задачи. Анализ графической подсистемы показал, что ядро центрального процессорного устройства (ЦПУ), а не ГПУ было загружено работой по подготовке и проверке буфера параметров для передачи данных в память графического ускорителя. Само ГПУ простаивало и не выполняло полезную работу расчёта графики. Поэтому возникла задача минимизации количества запросов на рисование при сохранении большого количества видимых объектов сцены, и было принято решение изменить архитектуру системы для оптимизации процесса передачи данных.

Предлагаемая нами архитектура графического движка (рис. 6) имеет несколько оптимизаций для сохранения визуальной плотности объектов сцены, используя набор ухищрений для минимизации размера параметрического буфера, а также обходы си-

стемы проверки параметрического буфера группировкой данных, не отражающей действительное количество видимых объектов. Из набора оптимизаций можно отметить следующие ключевые моменты:

- поддержка дерева обновлений объектов;
- объединение геометрии и параметров нескольких объектов;
- дублирование геометрии однотипных объектов;
- вычисление разности геометрии и частичное обновление буферов ГПУ.

Цель проведения данных оптимизаций – создание графической системы, способной производить визуализацию результатов физических симуляций с большим количеством объектов.

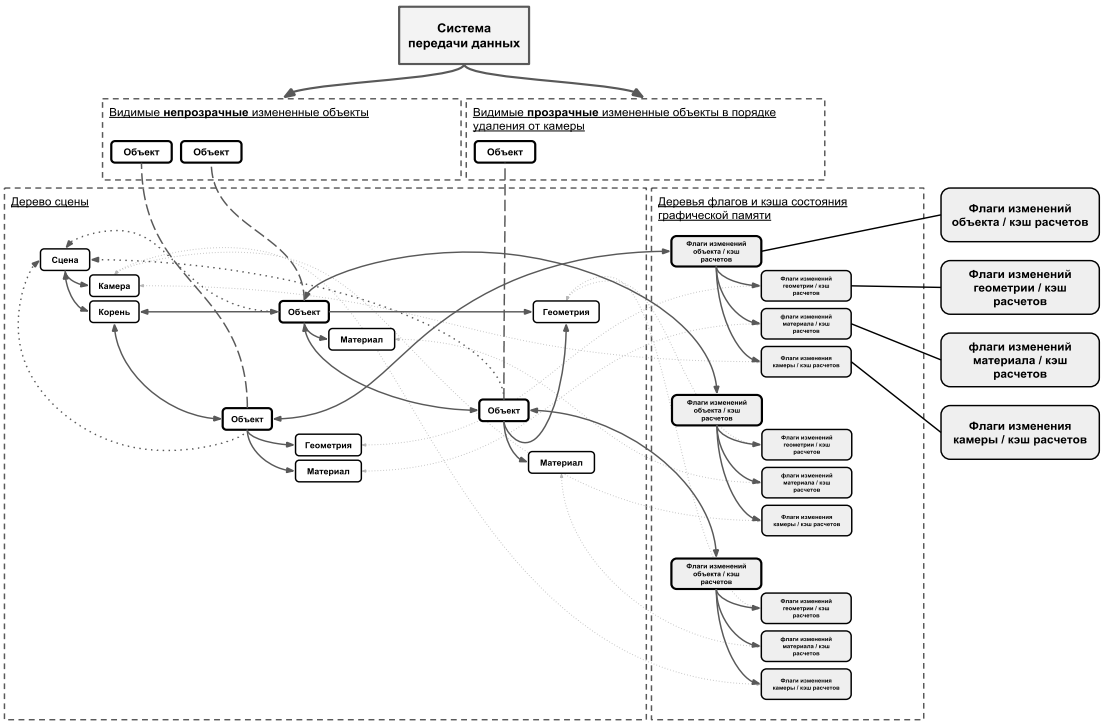

Рисунок 3. Дерево сцены, деревья изменений и кэша (Renderables) и система передачи данных на ГПУ оптимизированной версии графического движка

Дерево обновлений объектов

Первая важная оптимизация – уменьшение параметрического буфера данных. Для каждого объекта базы данных сцены хранится специальный экземпляр класса флагов и кэша изменений. В нашей системе мы называем такой объект Renderable. Каждый объект Renderable имеет набор чисел, используемых как битовые карты флагов, которые могут ответить на вопрос – изменился ли определенный параметр связанного объекта мира с момента предыдущего вызова на рисование, или нет. Каждый объект базы данных сцены теперь имеет набор сеттеров для обновления соответствующих битов флагов Renderable. В будущем планируется заменить сеттеры на шаблон наблюдения (Observable pattern) при наличии поддержки библиотеками или хост-средствами среды исполнения языка. Изменение параметров прозрачности приводит к тому, что объект самостоятельно перемещает себя в одноуровневый список прозрачных или непрозрачных объектов системы передачи данных.

Система передачи данных из первых версий движка также была изменена и теперь производит анализ двух списков на рисование (прозрачные объекты и непрозрачные объекты) и выполняет необходимые сортировки и комбинации объектов по мере необходимости. Далее система проходит по списку непрозрачных и по списку прозрачных объектов. На каждой итерации система анализирует связанный объект флагов Ren-

derable, вычисляет промежуточные данные (по необходимости) и кэширует их на случай отсутствия изменений при повторном использовании в следующем кадре. Затем каждый объект флагов делает запрос к интерфейсу драйвера ГПУ на передачу данных, но только, если флаг изменений для определенного параметра был установлен. В завершение система обнуляет все флаги для анализа изменений следующего кадра. Нужно отметить, что такой подход передает только те данные, которые были изменены. Кроме того, система следует рекомендациям тайловых архитектур отложенного расчета и рисует непрозрачные и прозрачные объекты в порядке, помогающем ГПУ максимизировать прирост производительности при удалении невидимых поверхностей [3, 6]. Система сначала рисует все непрозрачные объекты, а после начинает работать с прозрачными, если они имеются в наличии.

Объединение геометрии

Системы визуализации физических явлений часто требуют демонстрации простых примитивов схематично при дальних расстояниях удаления от объектов. Количество примитивов в таких случаях может достигать миллионов фигур, одновременно видимых на экране. Средства минимизации передачи данных никак не помогут преодолеть ограничения тайловых архитектур. Так как подобного рода примитивы не требуют большого количества вершин, мы предоставляем механизмы, комбинирующие геометрию объектов со всеми связанными данными на центральном процессоре. Комбинирование можно делать как статически перед началом цикла рисования, так и динамически между последующими кадрами. Функции комбинирования используют вставки на ассемблере для разных целевых семейств процессоров (x86, ARM) для ускорения процесса. В будущем планируется заменить ассемблерные вставки на встроенные (intrinsic) функции компилятора для улучшения переносимости кода. Для модификации положения объектов на ГПУ мы предоставляем возможность передавать вспомогательные данные трансформации методом упаковки в текстуры. Для этого в нашем графическом движке есть специальный класс, позволяющий упростить трудоемкий процесс подготовки на стороне ЦПУ и распаковки на стороне шейдерных программ. Такой подход сложен в реализации, но, тем не менее, позволяет получить портативное решение для дешевых или старых устройств без обновлений ОС и новых графических API (**a**pplication **p**rogramming **i**nterface, интерфейс прикладного программирования), таких как, например, многие телефоны и ноутбуки рынков развивающихся стран [2, 5, 7].

Дублирование геометрии

Подход комбинирования хорошо подходит для простых примитивов из-за низкого количества вершин многоугольников. К сожалению, он не применим для объектов с большим количеством вершин. В нашей системе визуализации многие тела вблизи камеры отображены в виде сфер. Для демонстрации плавных переходов на видимых границах необходимо иметь более 1000 вершин для каждого объекта. Комбинирование такого количества данных на ЦПУ приведет к значительной потере производительности. Для решения этой проблемы, используется API дублирования геометрии (geometry instancing API), доступный на 90 % целевых устройств. Идея дублирования заключается в том, что мы передаем геометрию однотипных объектов всего лишь один раз (рис. 7). Кроме того, подготавливается специальный буфер дублирования с глобальными параметрами трансформации, цвета и материалов для каждого отдельного объекта. Отметим, что второй буфер значительно меньше и часто состоит из нескольких векторов для каждого объекта. В заключение, используя специализированные вызовы драйвера ГПУ, мы командуем нарисовать определенный объект N раз, используя одинаковую геометрию, но передавая разные глобальные параметры (трансформации, цвета, материалов) из буфера дублирования в шейдерную микропрограмму графического адаптера. Таким

образом, мы получаем *N* объектов с одинаковой геометрией, но отличительными свойствами позиции, ориентации, цвета, текстурирования и материалов. Используя вершинный шейдер, можно также изменить геометрию отдельных объектов. Такой подход кардинально разгружает шину передачи данных для рисования тысяч однотипных объектов на существующих графических адаптерах [3, 6].

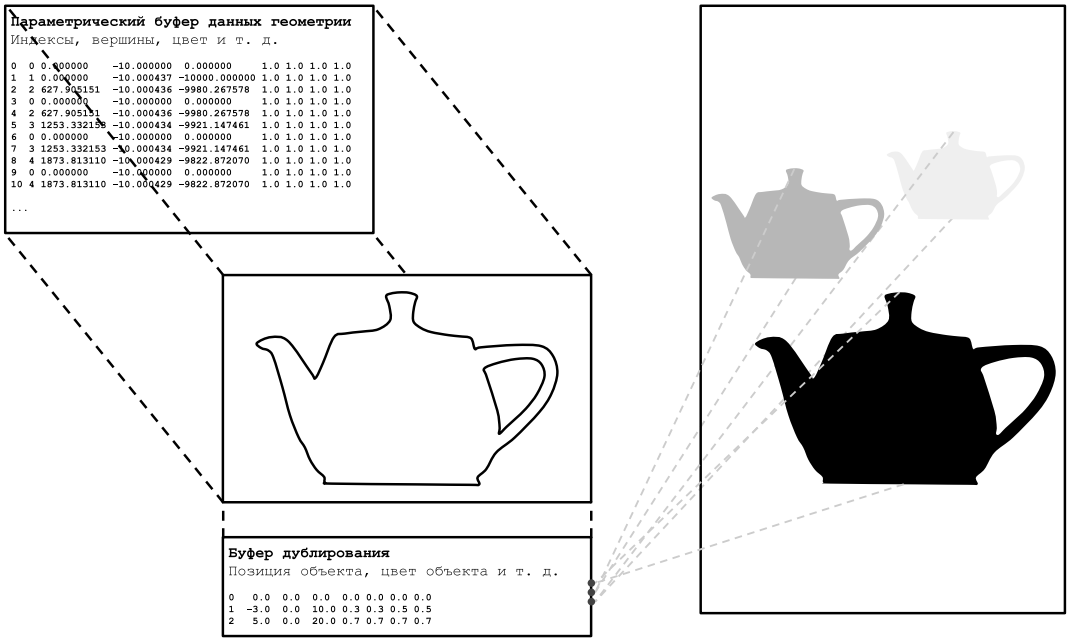

Рисунок 7. Схема дублирования геометрии на графическом адаптере

Определение разности геометрии

Для проверки движения небесных тел нам необходимо рисовать след в виде ленты за каждой планетой. Без поддержки геометрических шейдеров мы можем обновлять геометрию ленты на центральном процессоре. Обновление ленты происходит по методу игры «Змейки» – мы обновляем первый и последний элемент буфера и сдвигаем указатели начала и конца, что сводится к изменению 4 векторов для каждого кадра. К сожалению, первая версия не имела механизма частичной передачи данных и анализа разницы конечной сетки геометрии объектов. По этой причине для 4 обновленных векторов на ЦПУ система передавала до 1000 векторов всей ленты на ГПУ. Для решения проблемы класс геометрии был изменен. Каждое изменение вершин или элементов многоугольников записывается в специальный список изменений. Список сортируется, и последовательные векторы объединяются в регионы обновления. Каждый список дублируется для поддержки одновременной асинхронной передачи на ГПУ и обновления геометрии для следующего кадра на ЦПУ [3, 5]. При проходе по списку объектов на рисование класс флагов использует специализированные API асинхронного обновления регионов памяти ГПУ, следуя по сформированному списку изменений на ЦПУ. Тем самым, в случае рисования ленты движения небесного тела передается всего 4 вектора, вместо одной тысячи.

Оценка производительности

Для замера производительности новой системы была использована запись результатов моделирования гравитационного взаимодействия *N*-тел. Первая версия графического движка не могла справиться со сценами такой симуляции с более чем 800 объектов (при частоте обновления в 30 кадров в секунду), состоящей из простых геометрических примитивов, вроде прямоугольников и сфер. Однопоточная версия новой системы

с применением описанных оптимизаций позволила увеличить количество сфер до 2000, а количество прямоугольников – до 260 000 (рис. 8, 9).

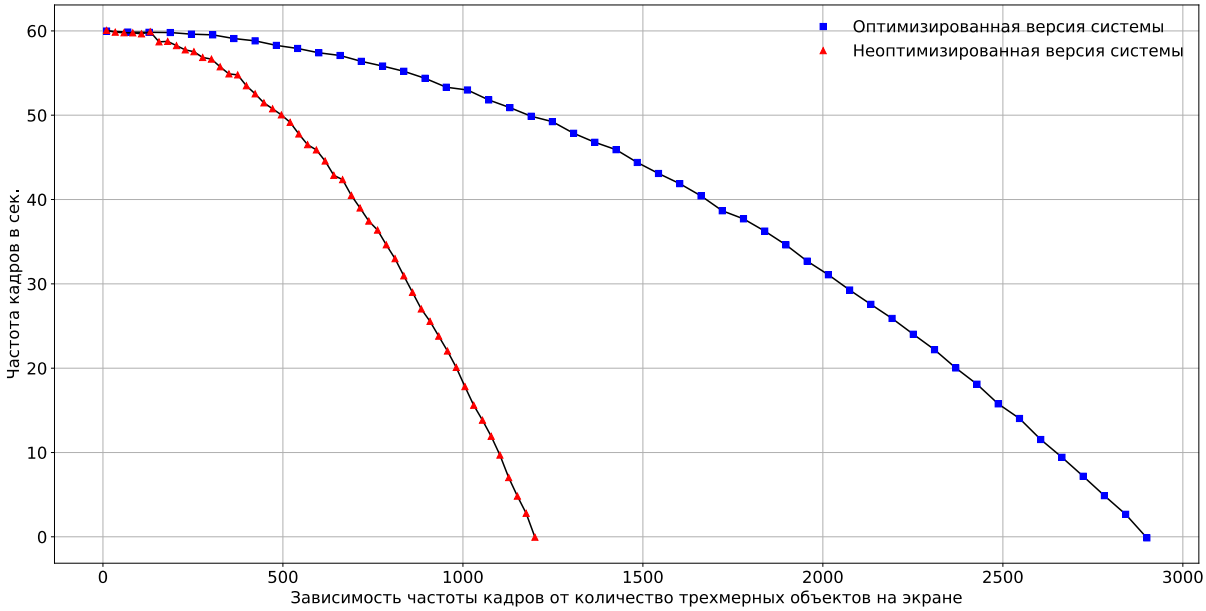

Рисунок 8. Частота обновления экрана при различных количествах трехмерных объектов

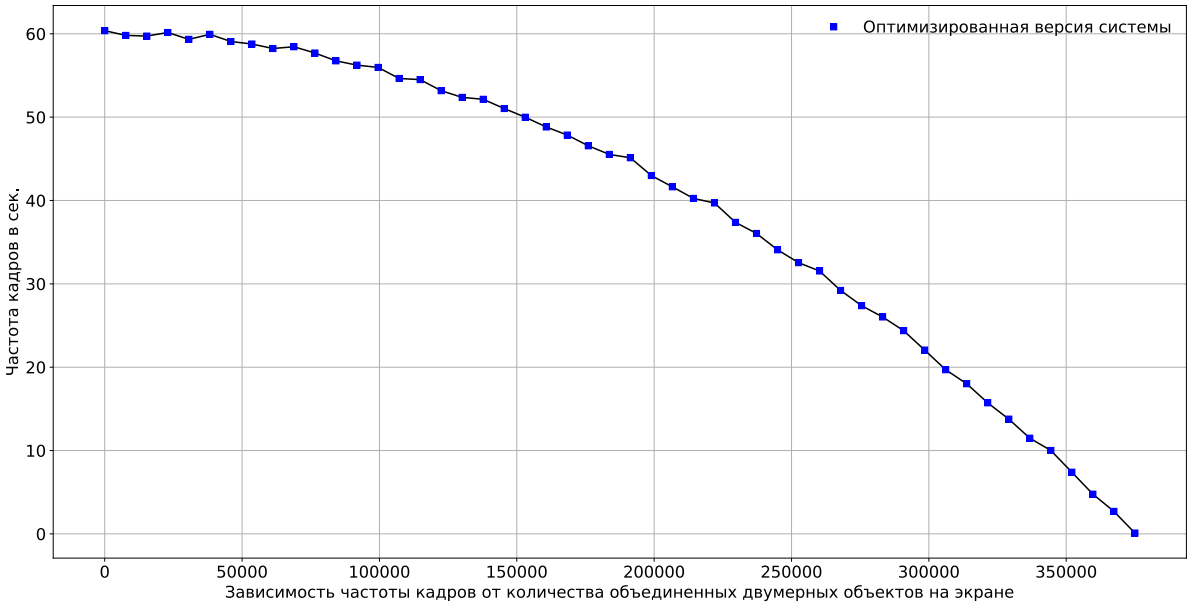

Рисунок 9. Частота обновления экрана при различных количествах объединенных двумерных объектов (неоптимизированная версия не имела подсистемы объединения)

Прирост количества трехмерных объектов в основном получен методом дублирования. Прирост количества простых двумерных примитивов получен путем комбинирования. Другие оптимизации имеют меньше влияния на ускорение, но, безусловно, их значение будет возрастать по мере усложнения видов визуализации. Примеры визуализаций результатов моделирования гравитационного взаимодействия *N*-тел с большим количеством графических примитивов можно увидеть в [8].

Заключение

Таким образом, представлены реализация и четыре способа оптимизации графического движка для визуализации большого количества двумерных и трехмерных примитивов для мобильных тайловых процессоров. Особенности архитектуры требуют минимизировать количество данных, передаваемых на графический адаптер, и количество запросов на рисование. Показано, что предложенная нами система достигает этих целей, храня дерево изменений состояния каждого объекта сцены, для подготовки и минимизации списка объектов на рисование. При этом также уменьшается количество объектов передачи путем динамического объединения их геометрии. Кроме того, используется API для дублирования геометрии при повторной прорисовке, передается только разница между конечными сетками геометрии. Произведена оценка влияния предложенных оптимизаций на примере визуализации моделирования гравитационных взаимодействий N тел. Результаты измерений на наборе целевых устройств демонстрируют прирост производительности в 2,5 раза для трехмерных объектов и более 2-х порядков для двумерных объектов. Такой прирост делает систему рентабельной на переносных устройствах с ограниченным профилем энергопотребления.

Литература

1. Верзунов С.Н. Разработка программных средств для вейвлет-анализа одномерных временных рядов // Проблемы автоматики и управления. – 2014. – № 2(27). – С. 62–71.
2. Shebanow, M. An evolution of mobile graphics. Презентация конференции High Performance Graphics. 2013.
3. Imagination Tech Limited. PowerVR Hardware Architecture Overview for Developers // PowerVR SDK Manuals. 2017.
4. Navik, Ankit P., и другие. Microbenchmark Based Performance Evaluation of GPU Rendering. // Emerging Research in Computing, Information, Communication and Applications. Springer, New Delhi, 2015.
5. McCaffrey, Jon. Exploring Mobile vs. Desktop OpenGL Performance. // OpenGL Insights. 2012.
6. Arnau, Jose-Maria, Joan-Manuel Parcerisa, и Polychronis Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. // Proceedings of the 22nd international conference on Parallel architectures and compilation techniques. IEEE Press. 2013.
7. Ma, Xiaohan, и другие. Characterizing the performance and power consumption of 3D mobile games. // Computer 46.4. 2013. (С. 76–82).
8. http://iait.kg/wp-content/uploads/2018/04/Ver_Toks.png (дата обращения: 2. 04.2018).

METHODS OF OPTIMIZATION 3D GRAPHICS RENDERING ON TILE-BASED GRAPHICSPROCESSORS BY THE EXAMPLE OF A VISUALIZATION MODELING N-BODY GRAVITATIONAL INTERACTIONS

Visualization of physical phenomena such as gravitational interactions of celestial bodies requires rendering a large number of identical 2D and 3D objects. Unfortunately, tile-based deferred architectures of graphical accelerators, popular on the mobile market, are not suitable to process a large number of rendering requests. On such devices, it is necessary to use various optimization techniques to minimize the number of draw calls. In this work, we present implementation of a graphical engine that maintains a tree of state changes for every scene object to allow resubmission of only necessary sets of data to perform rendering. Furthermore, our system minimizes the number of objects to transfer by merging meshes and performing geometry instancing. The system also offloads the data bus by computing and sending only the differences between geometry meshes. The speedup relative to an unoptimized version is demonstrated by measuring the speed of systems to visualize modeling results of gravitational interactions between a large number of celestial bodies.
Keyword: tile-based processors, TBDR architecture, optimization, graphics engine, state update tree, geometry merging, geometry instancing, mesh difference, N-body simulation.