ПРОГРАММИРОВАНИЕ, 2022, № 3, с. 40–49

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

УДК 004.921

СЖАТИЕ 3D-МОДЕЛЕЙ С ПОДДЕРЖКОЙ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ НА GPU

© 2022 г. А. В. Николаев[a,*] (ORCID: 0000-0002-7194-2471), В. А. Фролов[a,b,] (ORCID: 0000-0001-8829-9884), И. Г. Рыжова[b,***] (ORCID: 0000-0003-1613-3038)**

[a] *Московский государственный университет им. М.В. Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д. 1, Россия*

[b] *Институт прикладной математики им. М.В. Келдыша РАН, 125047 Москва, Миусская пл., д. 4, Россия*

**E-mail: anton.nikolaev@graphics.cs.msu.ru*

***E-mail: vfrolov@graphics.cs.msu.ru*

****E-mail: ryzhova@gin.keldysh.ru*

Поступила в редакцию 14.12.2021 г.
После доработки 11.01.2022 г.
Принята к публикации 16.01.2022 г.

В данной работе предлагается метод сжатия 3D-моделей с быстрой декомпрессией на GPU. Данный метод позволяет поместить от 3 до 8 раз больше геометрии в том же объеме памяти GPU и применяет декомпрессию на лету непосредственно в процессе растеризации или трассировки лучей. Для растеризации мы реализовали декомпрессию в двух вариантах – на геометрических и меш-шейдерах. Для трассировки лучей мы задействовали аппаратное ускорение при помощи расширения Vulkan VK_KHR_ray_tracing_pipeline и предложили технику кэширования листьев BVH дерева, ускоряющую визуализацию до 2 раз. Выводы, которые мы сделали в рамках исследования производительности могут быть полезными для проектирования аппаратной поддержки сжатия 3D-моделей в будущих GPU, так как мы натолкнулись на аппаратные ограничения существующих GPU.

DOI: 10.31857/S0132347422030086

1. ВВЕДЕНИЕ

Рост детализации 3D-моделей, используемых в рендеринге, неизбежно приводит к росту необходимых объемов памяти. При т.н. оффлайн рендеринге встречаются случаи, когда сцена не может быть полностью помещена в память GPU. В случае рендеринга в реальном времени память так же является ограниченным ресурсом, так как используется не только для хранения геометрии, но и для хранения текстур сцены, буферов и текстур, содержащих промежуточные результаты рендеринга и т.д. Существующие методы сжатия геометрии в основном позволяют производить декомпрессию модели только перед загрузкой в видеопамять, что позволяет экономить пространство на внешних носителях, но не уменьшает потребление памяти GPU.

2. СУЩЕСТВУЮЩИЕ МЕТОДЫ СЖАТИЯ 3D-МОДЕЛЕЙ

Сжатие данных вершин. Широко распространенным приемом является использование квантизации для сжатия данных (атрибутов) вершин. При этом производится сжатие с потерями, основанное на факте, что стандартное представление в формате IEEE-754 является избыточным для применения в большинстве задач компьютерной графики. В [1] производится разбиение интервала значений от минимума до максимума на 2^n уровней и дальнейшее округление каждого числа с плавающей точкой до одного из двух ближайших уровней. Такой подход требует хранения лишь минимального и максимального значений в виде чисел в формате IEEE-754 и n бит данных для каждого числа из сжимаемых данных.

Большинство методов позволяют независимо выбирать количества бит для координат, нормалей и текстурных координат под требования задачи. Возможна квантизация изменений, позволяющая с использованием меньшего количества бит получить ту же точность. Выбор значений, от которых отсчитываются изменения при этом может быть различным и зависит от алгоритма сжатия. Так же возможно дальнейшее сжатие результатов с использованием энтропийного кодирования [2].

Сжатие данных связей. Одним из первых подходов, позволяющих сжимать данные связей, является применение т.н. (Generalized Triangle Strip), объединяющего известные способы обхода Triangle Strip и Triangle Fan, а также предоставляющего возможность указать добавление нового треугольника, не имеющего общих индексов с предыдущими. В данном методе помимо одного индекса новой вершины для каждого треугольника указывается один из трех случаев Restart, Replace Oldest и Replace Middle, определяющих стратегию выбора вершины треугольника из предыдущих [2].

Существует большая группа методов, предполагающая постепенное сжатие области путем последовательного добавления к ней треугольников с сохранением истории обхода. Примером такого метода является Cut-Border [3]. Данный алгоритм разделяет 3D-модель на две части: внутреннюю и внешнюю, содержащие уже пройденные и еще не пройденные треугольники соответственно. На каждом шаге рассматривается один из треугольников, находящихся на границе областей, и для него выбирается один из 6 случаев. При этом сохраняется код, уникальный для каждого случая, а для некоторых случаев так же хранится дополнительный параметр. Этой же группе принадлежит алгоритм Edgebreaker [4]. Его идея заключается в последовательном обходе треугольников с записью для каждого из них одного из пяти символов: C, L, R, E, S. На каждом шаге для выбранного ребра границы рассматривается еще не пройденный треугольник, содержащий это ребро. При этом символ выбирается в зависимости от расположения противолежащей рассматриваемому ребру вершины треугольника на границе сжимаемой области. Angle Analyzer [5] объединяет некоторые идеи Edgebreaker и Cut-Border, используя пять символов, как и Edgebreaker, но описывает с их помощью другие случаи. Например, символ J, возникающий при объединении двух границ, хранит смещение относительно текущей точки. Кроме того, данный метод может применяться к 3D-моделям, содержащим одновременно треугольные и четырехугольные грани, а также позволяет добиться несколько большей эффективности сжатия.

Другим распространенным подходом является сжатие на основе степеней вершин. В таких алгоритмах обход так же начинается с одного из треугольников и далее происходит расширение границы пройденной области, но сохраняются степени добавляемых вершин. Один из первых таких подходов был предложен в [6]. В данном методе все вершины начального треугольника помещаются в список. Далее из списка выбирается вершина, обходятся исходящие ребра и добавляются в список новые вершины, а для текущей вершины записывается степень. Степени далее могут быть подвергнуты арифметическому кодированию, что позволяет достичь высокой эффективности сжатия. В некоторых случаях для корректной работы алгоритма требуется так же запись дополнительных специальных символов или добавление вершин. В [7] было предложено дальнейшее улучшение этого подхода, позволяющее повысить эффективность сжатия за счет более оптимального обхода вершин в списке, позволившего менее часто использовать добавление вершин и специальные символы.

Так же существуют методы, использующие постепенное повышение детализации 3D-модели многократным применением некоторой операции над базовой моделью с детализацией куда ниже, чем у исходной сжимаемой [8]. Возможны подходы, предлагающие пошаговое слияние вершин [9], находящихся на одном из ребер (соответственно с разбиением ребер при декомпрессии), подходы на последовательном удалении вершин [10] и т.д. В [11] предлагается подход, позволяющий объединить сжатие за счет постепенного изменения детализации с процедурной генерацией моделей. Работа [12] предлагает способ пространственно-временной сегментации для сжатия анимированных 3D-моделей. В [13] предлагается другой способ сжатия анимированных последовательностей моделей при условии неизменности данных о связях. Предлагается построение упрощенных моделей, из которых далее получается требуемый элемент последовательности. В работах [14, 15] производится обзор существующих методов сжатия и в целом выявляются направления, аналогичные описанным выше.

В целом все описанные методы имеют один общий недостаток, несмотря на разные применяемые идеи: значительное распараллеливание при реализации декодирования невозможно, так как существует зависимость по данным между каждым следующим и предыдущим шагом декомпрессии. Из этого следует, что данные методы мало подходят для декомпрессии на GPU непосредственно во время рендеринга, поскольку GPU в первую очередь позволяет ускорить выполнение алгоритма именно за счет значительного распараллеливания вычислений.

Сжатие BVH дерева и геометрии для трассировки лучей. В трассировке лучей существует отдельное направление: сжатие BVH дерева, используемого для ускорения поиска пересечений, поскольку для массивных сцен само по себе BVH дерево может занимать существенный объем, а его компактное представление дополнительно ускоряет трассировку за счет лучшей работы кэша [16, 17]. В работе [18] представлено большое количество существующих методов сжатия BVH дерева и сопутствующей геометрии для трассировки лучей. Сюда можно отнести иерархическую кван-

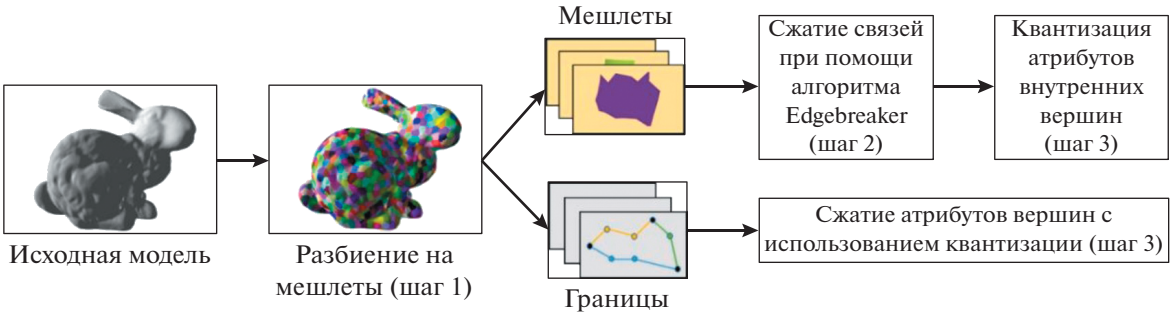

Рис. 1. Предложенный алгоритм сжатия. Сжатие связей (индексов) производится независимо от сжатия атрибутов вершин при помощи квантизации относительно ограничивающего параллелепипеда для мешлета.

тизацию (когда квантизация атрибутов вершин осуществляется относительно ограничивающего объема листа BVH дерева) [19], сжатые BVH со случайным доступом [20], в котором использовалось дельта-кодирование для индексов, преобразование индексированной геометрии в полоски (triangle strip), подходы с нулевыми затратами памяти на дерево [21, 22] и другие [18].

Ограничение этих работ заключается в том, что они рассматривают вопросы сжатия только в применении к трассировке лучей и не затрагивают сжатие геометрии в целях эффективной растеризации. В нашей работе это потребовало существенного изменения базовых алгоритмов. Кроме того, на данный момент еще не известно применение сжатия мешей в комбинации с аппаратной поддержкой трассировки лучей на GPU.

3. ПРЕДЛОЖЕННЫЙ МЕТОД

Основной идеей предложенного алгоритма является деление исходной 3D-модели на части (мешлеты) и сжатие отдельно каждой из этих частей. Мешлеты представляют из себя группы треугольников, такие что от любого из треугольников внутри группы можно дойти до любого другого, переходя только между треугольниками данной группы, имеющими общие ребра. Для повышения эффективности сжатия, наборы вершин, принадлежащие двум мешлетам одновременно, (далее называемые границами) сохраняются отдельно, что позволяет избежать дублирования данных. Схема предложенного метода сжатия показана на рис. 1.

При сжатии границ и внутренних вершин мешлетов используется квантизация изменений. Сжатие данных о связях треугольников внутри каждого мешлета производится при помощи алгоритма Edgebreaker [4]. Похожий подход был предложен в статье [23], однако авторы использовали мешлеты со значительно большим количеством треугольников и основной целью такого подхода являлась возможность декомпрессии лишь части 3D-модели, видимой виртуальной камерой, что позволяло использовать меньшее количество ресурсов для декомпрессии. Так же авторами [23] использовался Angle Analyzer для сжатия связей. В данной работе предлагается использование коротких (менее 64) последовательностей треугольников в каждом мешлете и делается упор на рассмотрение возможностей параллельной декомпрессии сжатой таким образом 3D-модели.

Применение описанного деления 3D-модели позволяет сделать обработку каждого мешлета независимой и, следовательно, декомпрессия всех мешлетов может производиться параллельно. В предлагаемом подходе результаты декомпрессии не сохраняются в памяти, а сразу передаются на графический конвейер (или конвейер трассироки лучей), благодаря чему на GPU хранится лишь та же сжатая модель, что и во внешней памяти. Это существенно для оптимизации т.н. стриминга, фоновой загрузки моделей в память GPU напрямую с диска [24, 25].

Генерация мешлетов. Был использован алгоритм разбиения на мешлеты во многом аналогичный предложенному в [23]. Суть алгоритма заключается в повторном выполнении двух итераций: выбора центров мешлетов и присоединения всех остальных треугольников модели к мешлетам до тех пор, пока мешлеты с предыдущей итерации не перестанут отличаться от полученных на следующей. Перед первой итерацией центры мешлетов распределяются равномерно по модели. Для присоединения треугольников к мешлетам используется функция весов (уравнение (1)), позволяющая получать мешлеты, регулярной формы, что положительно сказывается на отношении количества вершин на границе к количеству вершин внутри мешлета. Так же при помощи этой функции создаются мешлеты с малым отличием нормалей треугольников и примерно одинаковым количеством вершин, не большим заданного.

$$W_{t,m} = \left(\frac{F_m}{F_a} + C(F_m)\right) P(c_t, c_m)(\lambda - \langle n_t, n_m \rangle) \qquad (1)$$

Таблица 1. Сравнение разбиения с функцией из [23] и разбиения с предложенной функцией весов на модели Stanford Dragon

Функция	Количество мешлетов с количеством треугольников из указанного диапазона			
	[0, 32)	[32, 64)	[64, 72)	>72
Предложенная	1276	18474	216	23
Версия из [23]	1285	18435	232	38

$$C(N) = \begin{cases} 0, & N < N_{max} \\ C_m, & N \geq N_{max} \end{cases} \quad (2)$$

F_m – количество треугольников в текущем мешлете, F_a – среднее количество треугольников в мешлете, c_t – центр треугольника, c_m – центр мешлета, n_t – нормаль треугольника, n_m – нормаль мешлета, λ – параметр, контролирующий значимость планарности и компактности, N_{max} – максимальное количество треугольников в мешлете, C_m – константа, контролирующая вклад ограничения на количество треугольников в мешлете.

Основным отличием от алгоритма, описанного в [23] является наличие явного ограничения на количество треугольников в мешлете. Причиной его введения является особенность работы геометрических шейдеров GPU, требующих явного указания максимального количества генерируемых треугольников, а так же имеющих ограничение на суммарный размер атрибутов всех вершин, полученных в ходе запуска шейдера. Сравнение предложенной функции весов и исходной приведено в табл. 1. Так как это не позволяет полностью исключить образование мешлетов с количеством треугольников большим, чем указанное ограничение, такие мешлеты делились пополам до тех пор, пока ограничение не будет выполнено.

Сжатие данных о связях треугольников мешлетов. Для сжатия мешлетов, как было указано ранее, использован алгоритм Edgebreaker. Граница мешлета считалась уже описанной, в качестве начального ребра выбиралось первое ребро первой в списке границы мешлета. Так же был проведен анализ частоты появления каждого из символов C, L, R, E, S. Поскольку для любой модели форма мешлетов определяется в первую очередь функцией весов, используемой при разбиении мешлетов, можно считать, что при неизменной функции весов распределение символов будет одинаковым. Это позволяет выбрать фиксированные коды символов. На тестовых моделях были получены следующие распределения: 34–42% для R, 24–29% для C, 12–17% для E, 9–14% для S и 5–12% для L. Путем применения кодирования Хаффмана были получены коды из 2 бит для символов C, R, E и из 3 для символов L и S. В редких случаях такой выбор может оказаться неоптимальным, но данные истории занимают менее 5% от размера сжатой модели (рис. 3), и даже в худшем случае потери будут незначительны (менее 2.5% от общего размера). По этой же причине не было применено арифметическое кодирование.

Сжатие атрибутов вершин. Для сжатия координат вершин используется квантизация. При этом для координат первой и последней вершин каждой границы используются абсолютные значения, в то время как для всех остальных сохраняются только значения изменений. Для уменьшения количества не несущих информации ведущих нулей (в двоичном представлении), возникающих из-за сжатия изменений с тем же количеством бит, что и для абсолютных значений, для каждой из компонент используется дополнительный бит. С их помощью определяется одна из двух форм записи компоненты вектора позиции: квантизация с полным количеством бит n, или запись младших m бит, где m – целое число, выбираемое из интервала $[1, n-1]$, путем перебора всех возможных значений с целью минимизации итогового размера. Значение m является общим для всех компонент всех координат вершин. Если такое число найти не удалось, то оно считалось равным n, а дополнительный бит не записывался. На рис. 2 показан пример работы алгоритма в случае $n = 8$ и $m = 4$.

Для сжатия нормалей производился предварительный перевод векторов нормалей в сферические координаты, что позволяет хранить два значения вместо трех, так как радиус считается единичным. Сжатие изменений не использовалось.

При сжатии текстурных координат применялась квантизация абсолютных значений. Тут так же возможно сжатие изменений с представлением, описанным выше для координат. Однако, текстурные координаты занимают меньше данных чем координаты вершин (при используемых параметрах сжатия это максимум два 12-битных значения в сравнении с тремя для вершин при том же или большем количестве бит на каждую координату), а данный подход позволит достичь для них дополнительного сжатия не более чем в 1.5 раза (на тестовых моделях), при этом усложняя декомпрессию, поэтому использован не был.

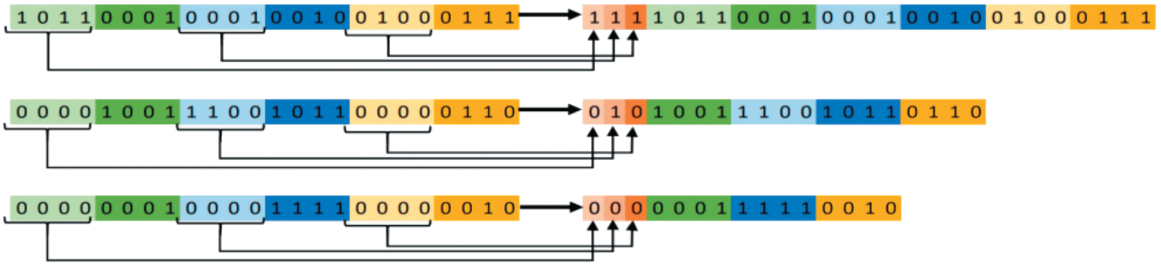

Рис. 2. Запись координат вершин на трех примерах при использовании 8 бит квантизации и выборе 4 бит для записи части числа. Слева более темными цветами показаны младшие биты каждой компоненты вектора после квантизации, светлыми цветами – старшие. Справа: первые три бита служебные, указывают форму записи каждой из компонент вектора. Далее идут блоки по 4 младших бита и 4 старших (пишутся только если хоть один бит в блоке не нулевой).

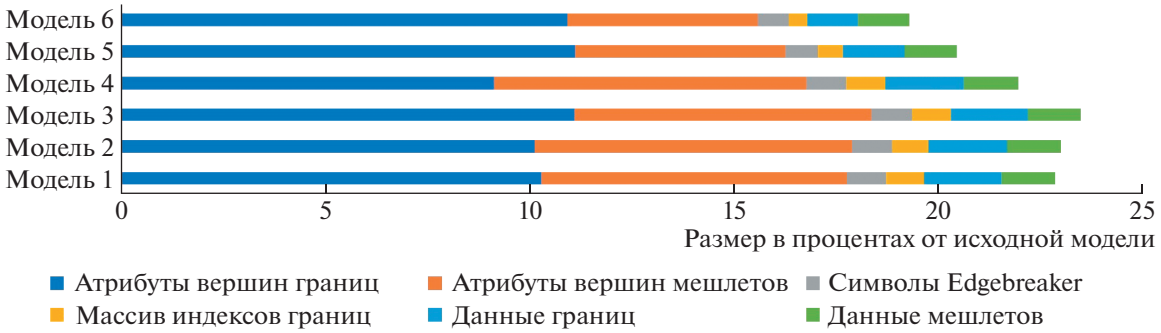

Рис. 3. Данные моделей после сжатия (в процентах от исходного размера модели) при использовании 64 треугольников на мешлет. Можно заметить, что сжатые атрибуты мешлетов (оранжевые прямоугольники) в сумме с остальными сжатыми данными занимают почти столько же, сколько несжатые атрибуты вершин границ (относительно которых и производится квантизация). Следовательно, для 64 треугольников в мешлете предложенный подход сбалансирован. Для дальнейшего повышения эффективности сжатия нужно увеличивать размер мешлета, что снижает параллельность при декомпрессии.

4. ДЕКОМПРЕССИЯ НА GPU

Растеризация с использованием геометрических шейдеров. Для декомпрессии при растеризации были использованы геометрические шейдеры как доступное средство на современных GPU. Каждый поток геометрического шейдера последовательно разжимал треугольники из одного мешлета. Недостатком такого подхода является относительно низкая скорость работы геометрических шейдеров, вызванная существующими аппаратными ограничениями. В большинстве аппаратных реализаций на GPU из геометрического шейдера нельзя просто отправлять треугольники на растеризацию по одному. Сначала нужно записать некоторое предопределенное заранее число треугольников (в нашем случае это количество треугольников в мешлете, так как они друг от друга зависят) в L1 кэш, после чего их все можно отправить на растеризацию. Объем L1 кэша ограничен, и в результате при большом числе треугольников в мешлете, геометрические шейдеры работают с низким occupancy (то есть GPU не может запустить достаточное число потоков), из-за чего геометрические шейдеры работают медленно [26].

Кроме того, проецирование вершин в экранное пространство из пространства модели обычно осуществляется при помощи умножения координат каждой вершины на одну или несколько матриц размерности 4×4. В случае декомпрессии модели по предложенному алгоритму данная операция производится после полного декодирования мешлета в геометрическом шейдере и может быть выполнена лишь последовательным обходом всех вершин, что так же снижает производительность.

Несмотря на указанные недостатки, в классическом конвейере растеризации нет другого этапа, позволяющего осуществлять декомпрессию мешлетов, сжатых предложенным образом. Вершинные и тесселяционные шейдеры не могут создавать более одной вершины в каждом потоке, в то время как для декомпрессии каждой следующей вершины требуется информация о предыдущей. Кроме того, использование тесселяции требует существенных ограничений не только на количество, но и на взаимное расположение треугольников

в мешлете, и разработать алгоритм, позволяющий для произвольной модели найти разбиение, удовлетворяющее этим требованиям, значительно сложнее.

Растеризация с использованием меш шейдеров. Частичным решением описанных проблем декомпрессии в геометрическом шейдере является применение расширения стандарта Vulkan API: меш шейдеров. Данное расширение было представлено компанией Nvidia и поддерживается GPU Nvidia Turing и последующими поколениями видеокарт Nvidia [27]. Оно описывает изменение классического графического конвейера, заменяя все этапы до растеризатора двумя программируемыми: таск и меш шейдерами.

В предложенном алгоритме таск шейдер осуществляет восстановление индексов вершин треугольников по истории обхода Edgebreaker. Для символа C, указывающего на добавление новой вершины, так же производится декомпрессия квантованных изменений и указываются индексы вершин, относительно которых отсчитывается это изменение. Далее данные индексов и описания добавленных вершин (изменения координат, остальные атрибуты и индексы предыдущих вершин) передаются в меш шейдер.

Меш шейдер запускается с 16 потоками для каждого мешлета. Производится параллельная декомпрессия вершин границ (вершины каждой из границ могут быть обработаны только последовательно из-за хранения изменений координат, однако границы независимы между собой, и каждая разжимается своим потоком). Далее в одном из потоков происходит преобразование координат внутренних вершин в глобальные на основе уже имеющихся абсолютных значений на границах и изменений с индексами вершин, полученных из таск шейдера. Эта операция является относительно простой (осуществляется небольшое количество простых операций сложения векторов), поэтому не сильно снижает эффективность параллельного исполнения меш шейдеров. Далее, когда получены абсолютные значения координат всех вершин, производится умножение на матрицы параллельно в 16 потоках. После этого индексы из таск шейдера и полученные координаты записываются в качестве результатов работы меш шейдера.

Декомпрессия при трассировке лучей. При декомпрессии с трассировкой лучей использовалось расширение стандарта Vulkan: VK_KHR_ray_tracing_pipeline, добавляющее поддержку ускоряющих структур, а также предлагающее новый конвейер, используемый для трассировки лучей.

Для поиска пересечений со сжатой 3D-моделью использовались intersection-шейдеры. В качестве ускоряющей структуры использовался набор описывающих прямоугольников, ориентированных по осям координат (AABBs, Axis Aligned Bounding Boxes). Координаты узлов AABB считались при запуске приложения с использованием вычислительного шейдера, осуществляющего разовую декомпрессию 3D-модели и поиск наибольших и наименьших координат каждого мешлета вдоль оси координат. В intersection-шейдере номер мешлета определялся на основе индекса описывающего прямоугольника, с которым проверяется пересечение в данный момент. Для проверки пересечения луча с мешлетом сначала производится декомпрессия, а далее производится проверка пересечения луча с каждым из треугольников мешлета. Если пересечений несколько выбирается ближайшее. Однако при такой реализации производительность декомпрессии оказывается крайне низкой. В отличии от растеризации, где каждый примитив и каждый мешлет обрабатывался один раз на кадр, в случае трассировки лучей их количество оказывается основным фактором: для каждого луча необходима декомпрессия мешлета, проверяемого на пересечение, при том выполняемая в одном потоке (в расширении VK_KHR_ray_tracing_pipeline нельзя запускать несколько потоков в intersction шейдере). Частично исправить эту проблему призвано кэширование декомпрессированных данных некоторых мешлетов, описанное далее.

Кэш для ускорения трассировки лучей. Идея данной оптимизации заключается в предварительной декомпрессии некоторого количества мешлетов при помощи вычислительного шейдера. Меняя количество кэшируемых мешлетов, можно добиться желаемого баланса между дополнительным потреблением памяти и производительностью трассировки.

Выбор мешлетов попадающих в кэш производится на основе подсчета количества запусков intersection-шейдеров для каждого из мешлетов. Используется операция atomicAdd, увеличивающая счетчик количества пересечений с текущим мешлетом на 1 при каждой проверке пересечения. Поскольку непосредственное получение этих данных для текущего кадра требует дополнительных вычислений, используются значения с предыдущего кадра с расчетом на малые изменения между кадрами. На основе полученных значений производится декомпрессия мешлетов, не присутствующих в кэше (записываются на место мешлетов, далее не попадающих в кэш). После этого в intersection шейдере производится проверка на наличие мешлета в кэше и либо проверяется пересечение с уже расжатым мешлетом, либо предварительно производится декомпрессия.

На графике (рис. 5) приводится зависимость производительности трассировки в зависимости от размера кэша.

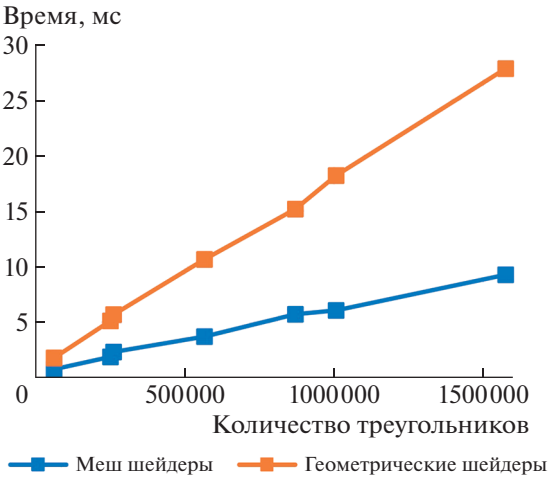

Рис. 4. Сравнение декомпрессии при помощи геометрических и меш шейдеров (проводилось на Nvidia RTX 2070 Mobile).

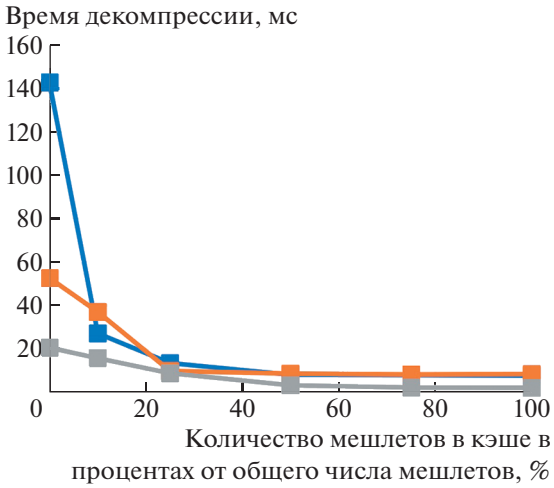

Рис. 5. Зависимость скорости декомпрессии от количества мешлетов в кэше. Количество лучей 921600. Разными цветами обозначены разные модели.

5. ОГРАНИЧЕНИЯ ПРЕДЛАГАЕМОГО АЛГОРИТМА

Большинство ограничений применимости предложенного метода обусловлено делением на мешлеты. Так, данный алгоритм требует сглаживания как минимум большей части нормалей 3D-модели, так как в противном случае разбиение на мешлеты не будет находить смежные треугольники, создавая много мешлетов малого размера и понижая эффективность сжатия. Другой проблемой, вызванной применяемым алгоритмом разбиения на мешлеты, является использование алгоритма на моделях, имеющих наборы из малого количества треугольников, не имеющих общих вершин с остальной моделью. Это приводит к снижению эффективности сжатия, так как описание мешлета не изменяется, а количество треугольников в нем уменьшается, приводя к большему количеству бит на треугольник.

6. АНАЛИЗ РЕЗУЛЬТАТОВ

Оценка эффективности сжатия. Для оценки эффективности сжатия имеет смысл оценить степень сжатия каждого из компонентов, полученных в результате.

Для данных вершин квантизация позволяет сократить объем данных, требуемых для каждой вершины с 256 бит (8 чисел с плавающей точкой IEEE-754) до 76 бит (при использовании параметров квантизации: количество бит для координат – 12, для нормалей – 8, для текстурных координат – 12), а использование уменьшенного количества бит для квантизации изменений координат дает дополнительное снижение размера этих данных еще в 1.23–1.34 раза (на тестовых 3D-моделях), позволяя получить итоговый размер в примерно 69 бит (уменьшение общего объема данных вершин в более чем 3.7 раза). На практике хранение обеих крайних точек для границы несколько снижает эффективность сжатия (например, с 3.7 до 3.1 раза на тестовых моделях).

Для данных связей применение алгоритма Edgebreaker позволяет получить использование менее чем 3 бит (2.19–2.22 на тестовых моделях) на треугольник вместо 32 бит для каждой из трех вершин, давая сжатие в более чем 32 раза. Объем дополнительных данных определяется числом мешлетов и количеством границ, которое на тестовых 3D-моделях превышает количество мешлетов в среднем в 2.9 раза (при этом каждый мешлет использует в среднем 5.5 индекса границ), что дает в результате порядка 400 бит дополнительных данных на каждый мешлет. Если относить дополнительные данные к описанию связей, то итоговый результат составляет порядка 11.63 бит на треугольник на тестовых моделях, что эквивалентно сжатию в 8.25 раз.

Таким образом, в зависимости от объема, занимаемого в 3D-модели данными вершин и связей, можно получить эффективность сжатия от 3.1 до 8.25 раза. На тестовых моделях, где примерно 40 процентов от исходного объема составляют данные связей, сжатие позволяло уменьшить объем требуемой памяти в 4.26–4.35 раза. Как можно видеть из рис. 4, большинство данных в сжатой модели составляют атрибуты вершин, дальнейшее уменьшение размера которых представляется мало возможным. Для этого требуется либо уменьшение количества бит квантизации (применимо в некоторых задачах, однако в общем случае ведет к потере точности и появлению визуальных искажений), либо применение эн-

Таблица 2. Анализ ограничений производительности декомпрессии при растеризации на моделях Blender Suzanne и Stanford Dragon (По данным Nvidia Nsight Graphics [28]). Время декомпрессии отличается от указанного в результатах, так как при замере в Nsight отключен GPU Boost

Модель	Шейдер	Процент варпов, не запущенных из-за нехватки L1 кэша	Время декомпрессии
Suzanne	Геометр.	50.8%	1.25 мс
Suzanne	Меш	14.2%	0.54 мс
Dragon	Геометр.	77.5%	19.1 мс
Dragon	Меш	28	7.13 мс

тропийного кодирования для этих данных, которое значительно усложняет декомпрессию и снижает скорость.

Анализ скорости декомпрессии. На рис. 4 и рис. 5 приводится сравнение производительности декомпрессии с использованием геометрических и меш шейдеров, а также декомпрессии при трассировке в зависимости от размера кэша. Было установлено, что основным ограничивающим производительность фактором является объем данных, записываемых геометрическими и меш шейдерами, а также массивов, используемых для временного хранения данных. При этом, наиболее заметен данный эффект оказывается при использовании геометрических шейдеров. Реально же используется менее 15% доступных вычислительных ресурсов GPU. Соответствующие результаты можно видеть в табл. 2. В случае декомпрессии при трассировке видно, что начиная с 50% мешлетов в кэше производительность уже практически не меняется. Это вызвано тем, что почти всегда часть мешлетов оказывалась закрыта другими. Ограничением же становится проверка пересечений со всеми треугольниками мешлета для каждого луча, реализуемая без аппаратного ускорения в intersection шейдере.

Сравнение с существующими методами. При сравнении с существующими методами была выбрана версия декомпрессии на меш шейдерах. Для сравнения были выбраны две библиотеки с открытым исходным кодом: Draco и Corto. Обе библиотеки поддерживают квантизацию данных, сжатие изменений и в том или ином виде используют для кодирования связей в обход треугольников.

Сравнение проводилось на шести 3D-моделях (рис. 6): Blender Suzanne, содержащей 62976 треугольников и 32057 вершин, Stanford Bunny (33528 вершин и 65630 треугольников), Material Ball (31310 вершин и 61088 треугольников), Sponza (184330 вершин и 262267 треугольников), Stanford Dragon (438929 вершин и 871306 треугольников), Amazon Lumberyard Bistro (использовалась только сцена интерьера, 1020907 вершин и 762263 треугольника).

Для квантизации во всех случаях использовались следующие значения: количество бит для координат – 12, для нормалей – 8, для текстурных координат – 12. Тестирование производилось на GPU Nvidia RTX 2070 Mobile и CPU Intel Core I7 10750H. При оценке скорости работы методов на CPU оценивалось среднее время декомпрессии по результатам 5 запусков, время загрузки 3D-модели в оперативную память не учитывалось. Результаты сравнения по 6 моделям с разными характеристиками приводятся на диаграммах (рис. 7–9).

Рис. 6. 3D-модели, использованные для сравнения.

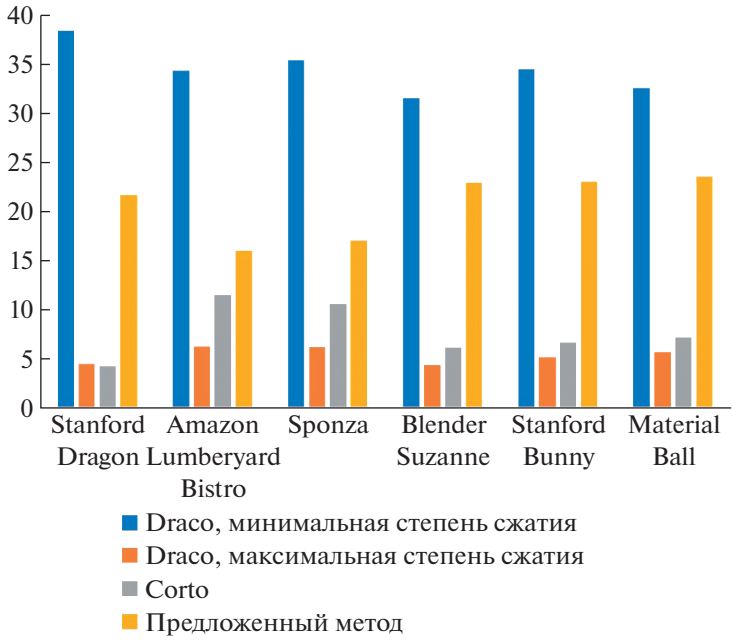

Рис. 7. Сравнение размера, занимаемого сжатой моделью в процентах от исходного размера.

Сравнение по объему памяти, требуемой для хранения на GPU не показано, так как для предложенного метода значение будет тем же, что на рис. 7, в то время как остальные методы будут хранить несжатую модель.

7. ВЫВОДЫ, ОБСУЖДЕНИЕ, БУДУЩИЕ ИССЛЕДОВАНИЯ

Предложенный нами подход позволяет от 3 до 8 раз повысить объем геометрии, который можно поместить в память GPU. Причем, почти всегда удается достичь 4 раз без видимых геометрических искажений даже при приближении камеры к поверхности. Однако и потенциальное замедление в визуализации достаточно существенное (5–10 раз). В реальных приложениях оно может быть меньше из-за того, что конвейер ограничен производительностью в других стадиях (например, фрагментный шейдер). Анализ производительности на специализированных инструментах [28] показывает, что существующие конвейеры на GPU имеют ряд ограничений, которые становятся причиной узких мест: (1) Для растеризации с использованием геометрических шейдеров это необходимость сохранять все данные декомпрессируемых вершин в L1 кэше GPU (shared memory) перед отправкой всего мешлета на растеризацию. (2) Для растеризации с использованием меш-шейдеров это необходимость записи данных

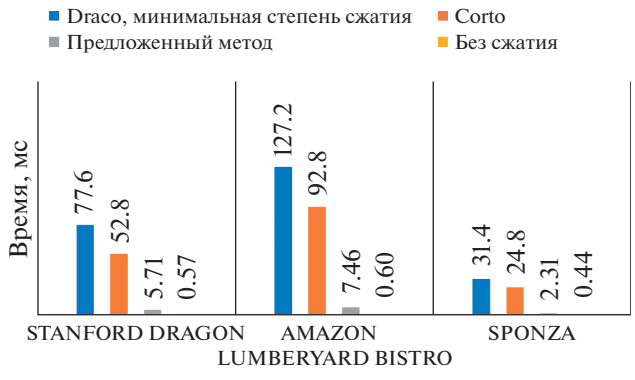

Рис. 8. Сравнение времени декомпрессии. Проводилось на GPU RTX 2070 Mobile для предложенного метода и версии без сжатия и на Intel Core I7 10750H для остальных методов, так как ими не поддерживается декомпрессия на GPU.

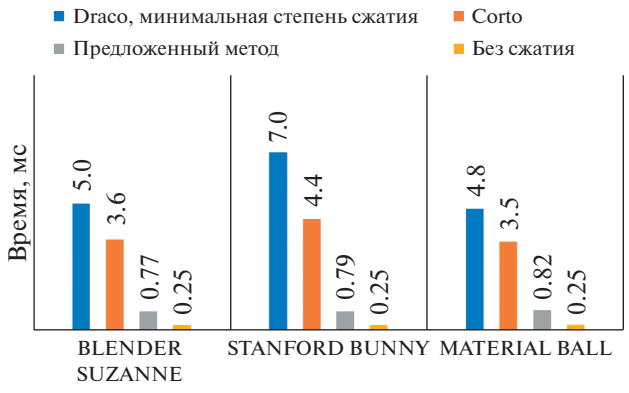

Рис. 9. Сравнение времени декомпрессии (Продолжение).

при передаче из таск шейдера в меш шейдер, а также использование L1 кэша для хранения временных данных для декомпрессии, так как не все их удается разместить на регистрах. (3) Для трассировки лучей производительность была ограничена скоростью декомпрессии данных в узлах.

СПИСОК ЛИТЕРАТУРЫ

1. *Chou P.H., Meng T.H.* Vertex data compression through vector quantization // IEEE Transactions on Visualization and Computer Graphics. 2002. V. 8. № 4. P. 373–382.
2. *Deering M.* Geometry compression // Proceedings of the 22nd annual conference on Computer graphics and interactive techniques. 1995. P. 13–20.
3. *Gumhold S.* Improved cut-border machine for triangle mesh compression // Erlangen Workshop. 1999. V. 99. P. 261–268.
4. *Rossignac J.* Edgebreaker: Connectivity compression for triangle meshes // IEEE transactions on visualization and computer graphics. 1999. V. 5. № 1. P. 47–61.
5. *Lee H., Alliez P., Desbrun M.* Angle-analyzer: A triangle-quad mesh codec // Computer Graphics Forum. Oxford, UK: Blackwell Publishing, Inc., 2002. V. 21. № 3. P. 383–392.
6. *Touma C., Gotsman C.* Triangle mesh compression // Proceedings-Graphics Interface. Canadian Information Processing Society, 1998. P. 26–34.
7. *Alliez P., Desbrun M.* Valence-driven connectivity encoding for 3D meshes // Computer graphics forum. Oxford, UK and Boston, USA: Blackwell Publishers Ltd., 2001. V. 20. № 3. P. 480–489.
8. *Abderrahim Z., Techini E., Bouhlel M.S.* State of the art: Compression of 3D meshes // Int. J. Comput. Trends Technol (IJCTT). 2012. V. 4. № 6. P. 765–770.
9. *Hoppe H.* Progressive meshes // Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996. P. 99–108.
10. *Cohen-Or D., Levin D., Remez O.* Progressive compression of arbitrary triangular meshes // IEEE Visualization. 1999. V. 99. P. 67–72.
11. *Uyttersprot S.* Mesh Compression and Procedural Content Generation. 2018.
12. *Luo G. et al.* Spatio-temporal segmentation based adaptive compression of dynamic mesh sequences // ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM). 2020. V. 16. № 1. P. 1–24.
13. *Hajizadeh M., Ebrahimnezhad H.* Eigenspace compression: dynamic 3D mesh compression by restoring fine geometry to deformed coarse models // Multimedia Tools and Applications. 2018. V. 77. № 15. P. 19347–19375.
14. *Maglo A. et al.* 3d mesh compression: Survey, comparisons, and emerging trends // ACM Computing Surveys (CSUR). 2015. V. 47. № 3. P. 1–41.
15. *Elmas A.A.* Investigation of Single-Rate Trianguar 3D Mesh Compression Algorithms.
16. *Mahovsky J.A.* Ray tracing with reduced-precision bounding volume hierarchies. 2005.
17. *Ylitie H., Karras T., Laine S.* Efficient incoherent ray traversal on GPUs through compressed wide BVHs // Proceedings of High Performance Graphics. 2017. P. 1–13.
18. *Meister D. et al.* A Survey on Bounding Volume Hierarchies for Ray Tracing // Computer Graphics Forum. 2021. V. 40. № 2. P. 683–712.
19. *Segovia B., Ernst M.* Memory efficient ray tracing with hierarchical mesh quantization // Proceedings of Graphics Interface. 2010. V. 2010. P. 153–160.
20. *Kim T.J. et al.* RACBVHs: Random-accessible compressed bounding volume hierarchies // IEEE Transactions on Visualization and Computer Graphics. 2009. V. 16. № 2. P. 273–286.
21. *Eisemann I.M., Bauszat P., Magnor M.* Implicit object space partitioning: The no-memory BVH. Technical Report 2011-12-16, Computer Graphics Lab, TU Braunschweig, 2011. V. 365.
22. *Chitalu F.M., Dubach C., Komura T.* Binary Ostensibly-Implicit Trees for Fast Collision Detection // Computer Graphics Forum. 2020. V. 39. № 2. P. 509–521.
23. *Choe S. et al.* Random accessible mesh compression using mesh chartification // IEEE Transactions on Visualization and Computer Graphics. 2008. V. 15. № 1. P. 160–173.
24. AMD Smart Access Memory. https://www.amd.com/en/technologies/smart-access-memory/. Accessed: 03.06.2021.
25. *Thompson A., Newburn C.* GPUDirect Storage: A Direct Path Between Storage and GPU Memory // NVIDIA Developer Whitepapers. 2019. V. 8.
26. *Joshua Barczak*, 2015. Why geometry shaders are slow. http://www.joshbarczak.com/blog/?pf7. Accessed: 05.05.2020.
27. *Christoph Kubisch*, 2018. Introduction to turing mesh shaders. https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/. Accessed: 05.05.2020.
28. Nvidia NVIDIA Nsight Graphics. https://developer.nvidia.com/nsight-graphics. Accessed: 03.06.2021.