Алгоритм непрерывной детализации учитывающий особенности архитектуры современных видеоускорителей.

С.А. Кузиковский, И. В. Белаго, Д.А. Гладкий, Н. А. Елыков.
{stas, glad , bel, nicolas }@sl.iae.nsk.su
Институт Автоматики и Электрометрии СО РАН
Новосибирск, Россия

АННОТАЦИЯ

В работе представлена модификация алгоритма упрощения геометрических моделей виртуальной среды основаного на методе прогрессивных сеток, позволившая за счет более оптимального, чем в классическом подходе использования кэша трансформированных вершин, значительно увеличить скорость визуализации.

Ключевые слова: *virtual reality, hardware accelerated real-time rendering, progressive meshes.*

1. ВВЕДЕНИЕ.

Широкое разнообразие центральных процессоров (*CPU)*, графических ускорителей (*GPU*), графических интерфейсов (*API*), скоростей памяти и системной шины обуславливает огромную разнородность графических платформ, на которых пользователь может запускать 3D приложения реального времени. Необходимо создание масштабируемых приложений, то есть таких, которые генерируют изображение с максимально возможным визуальным качеством на различных платформах.

Общее количество визуализируемых треугольников, задающих модели объектов сцены, зачастую превышает возможности современных графических ускорителей, однако, учитывая то, что детали в сцене распределяются перспективно, часть визуализируемых граней являются избыточными. Алгоритм, который позволяет непрерывно исключать подобные избыточные треугольники из модели, т.е. упрощать геометрическую модель объектов в сцене в зависимости от положения и ориентации наблюдателя, называется алгоритмом непрерывной детализации. Кроме того, подобный алгоритм может придать системе свойство масштабируемости.

Существует множество таких алгоритмов (*Geometrical MipMapping [1], ROAM[2], VDPM[3], VIPM[4]*), которые неоднократно рассматривались в различных работах, однако эти алгоритмы не учитывают особенности архитектуры современных ускорителей, что не позволяет им добиваться пиковой производительности. Поэтому необходим алгоритм, наиболее адекватный современной аппаратной базе.

Алгоритмы оптимизации, порождающие данные для детализации, могут быть платформо-независимыми, а могут параметризоваться характеристиками графического акселератора, то есть оптимизировать представление модели под конкретное оборудование. Первые обычно выполняются на стороне разработчика модели, и модель поставляется заказчику уже в оптимизированном виде, вторые производятся во время или непосредственно после установки приложения.

Предлагаемый алгоритм исходит именно из наличия такой предварительной оптимизации, и поэтому состоит из двух частей - собственно алгоритма непрерывной детализации, который исполняется в реальном времени во время отображения модели, а также предшествующего ему оптимизационного алгоритма, который готовит для первого необходимые данные.

Оптимизационная часть алгоритма не ограничена требованиями реального времени, не отбирает для своей работы ресурсы у конечного приложения, исполняется для экземпляра модели один раз, что позволяет производить качественную, хотя и ресурсоемкую, оптимизацию.

Часть алгоритма детализации, исполняющаяся в реальном времени, состоит в выполнении заранее построенных оптимизатором команд, и обладает линейной сложностью, зависящей от разницы между уровнями детальности модели в моменты, соответствующие двум последовательным кадрам отображения. В частности, если детальность модели не меняется (например, как модель, так и виртуальный наблюдатель, почти неподвижны), ресурсы CPU затрачиваются лишь на вычисление индекса уровня детальности.

2. ОСОБЕННОСТИ АРХИТЕКТУРЫ СОВРЕМЕННЫХ ВИДЕО УСКОРИТЕЛЕЙ.

Основное влияние на архитектуру современных аппаратных графических решений оказывают три факта:

- Высокая параллельность вычислений на *CPU* и *GPU*.
- Потоковый характер построения изображения (графического конвейера).
- Ориентация графических ускорителей на вычисления, а не на принятие решений.

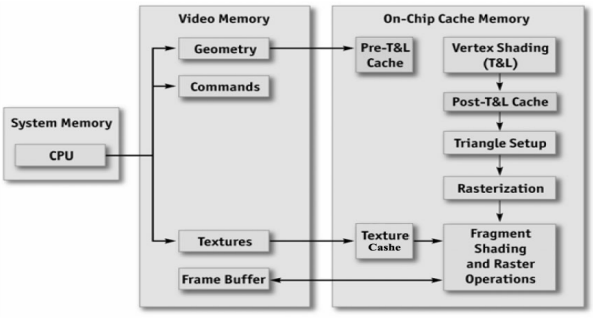

Рисунок 1: Типичный цикл построения кадра [5].

Таким образом, оптимальная работа всей системы в целом может быть достигнута только в том случае, когда нет простоев в потоке данных (ожидания данных), когда все компоненты системы работают параллельно и когда они работают с полной нагрузкой. Внесения задержек или разрушение параллелизма приводит к замедлению работы отдельных частей и всей системы в целом [5].

На рисунке 1 представлен типичный цикл построения кадра [5]. Нас интересует только та часть цикла, которая отвечает за обработку вершин модели:

1. Данные вершин выбираются по шине из системной памяти и попадают в предварительный кэш вершин («*Pre T&L Cache*»).
2. Далее, каждая из вершин попадает в вершинный процессор, в котором с помощью вершинного шейдера или фиксированного *T&L* конвейера происходит трансформация вершин.
3. После вершинного процессора вершины попадают в небольшой промежуточный буфер. Он называется «*Post T&L Cache*». Этот буфер играет двоякую роль, во-первых, служит для накопления результатов готовых отправиться на последующие стадии конвейера и таким образом уменьшает вероятность потенциальных простоев блоков ускорителя в ожидании данных, а во-вторых, позволяет избежать повторного трансформирования и обработки шейдером вершины, если она будет использована в скором времени повторно

Существует важное отличие графических ускорителей от универсальных процессоров: потоковый и предсказуемый характер данных позволяет обходиться небольшими кэшами высокой эффективности (*Post T&L* кэш вершин: ~16..32 вершины).

Для эффективного использования возможностей современных графических ускорителей необходимо обеспечить такой порядок граней, чтобы как можно большое количество обрабатываемых вершин бралось из *Post T&L* кэша. Выполнение этого условия может поднять скорость обработки геометрии в несколько раз

Однако, известные алгоритмы непрерывной детализации диктуют свой порядок следования граней в буфере объекта, как правило, этот порядок связан с особенностями работы алгоритмов и не соответсвует порядку, необходимому для эффективной работы кэша вершин графического акселератора. Таким образом, для достижения пиковой производительности, необходимо разработать новый или модифицировать уже существующий алгоритм непрерывной детализации, так чтобы утилизация *Post T&L* кэша была максимальна. В результате сравнительного анализа существующих алгоритмов в качестве основы для подсистемы непрерывной детализации объектов виртуального мира был выбран алгоритм ориентационно-независимых прогрессивных сеток треугольников [4] (*View Independent Progressive Meshes*).

3. ПРЕДСТАВЛЕНИЕ ПРОГРЕССИВНЫХ СЕТОК.

Сетка треугольников – M задается набором вершин $V \in R^3$ и набором $F \in V^3$ упорядоченных троек (v_i,v_j,v_k), которые задают вершины треугольника. Соседними вершине v треугольники называются все треугольники из M которые, содержат вершину v. Соседним по ребру $e = \{v_i,v_j\}$, называют объединение треугольников соседних вершинам v_i и v_j. Ребро, принадлежащее только одному треугольнику, называют граничным ребром. Вершины, из которых состоят граничные ребра, называют граничными вершинами, остальные вершины из M называют внутренними вершинами. Валентностью вершины назовем количество ребер, к которым принадлежит данная вершина.

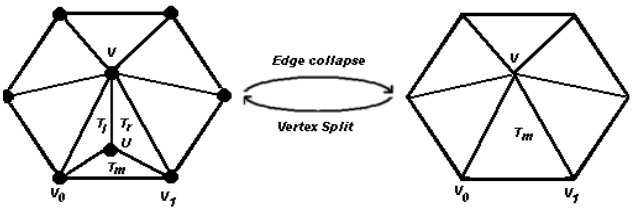

Рисунок 2. Пример выполнения операции *vsplit* и *ecol*.

Пусть M_0 – грубая триангуляция, полученная из M, тогда прогрессивная сетка (*PM*) это последовательность n операций разделения вершин - ***vsplit***, превращающих M_0 в M. Операция ***vsplit*** – добавляет в сетку одну новую вершину и два новых треугольника. Т.е. *PM* представляется, как $M_0 \xrightarrow{vsplit_0} M_1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} M_n (M_n = M)$, операция $\mathbf{vsplit}_{i-1}$ генерирует i–ую сетку из *PM*. Таким образом, *PM* задается набором (M_0,{$\boldsymbol{vsplit}_0$,$\boldsymbol{vsplit}_1$,…,$\boldsymbol{vsplit}_n$}). Можно ввести операцию обратную разделению вершин - схлопывание ребра - ***ecol***, которая в противоположность ***vspit***, убирает из M два треугольника и одну вершину. 0

На рисунке 2 представлен пример работы операций разделение вершин и схлопывания ребер, добавляется в случае ***vsplit*** или исключается в случае ***ecol*** вершина v и треугольники T_l и T_r.

4. ОРИЕНТАЦИОННО-НЕЗАВИСИМЫЕ ПРОГРЕССИВНЫЕ СЕТКИ (*VIEW INDEPENDENT PROGRESSIVE MESH)*.

Пусть мы имеем сетку *PM* заданную массивом вершин V, содержащий все вершины из полной сетки, массив триплетов F – {v_i,v_j,v_k}, которые на основе массива V описывают треугольники составляющие полную геометрическую модель и последовательность операций {$\boldsymbol{vsplit}_0$,$\boldsymbol{vsplit}_1$,…,$\boldsymbol{vsplit}_n$}.

И пусть мы имеем *PM* в некотором текущем состоянии - i, это состояние задается двумя массивами V и F, количеством активных вершин - *CurNumVerts* в V, и количеством активных треугольников – *CurNumTris* в F. Чтобы увеличить количество активных вершин, мы применим (i–1)-ю операцию ***vsplit***, которая добавит одну активную вершину, соединенную новым ребром с некоторой вершиной $splitVert \in V$, так же вдоль нового ребра появится два новых треугольника (исключение составляют граничные вершины, в этом случае появляется один новый треугольник). Так же с появлением новых треугольников произойдет изменение в самой геометрической модели, См. например рисунок 2, треугольник T_m до применения операции ***vsplit*** задается триплетом {v_0,v,v_1}, а после применения задается с помощью триплета {v_0,u,v_1}. Т.к. последовательность выполнения операций ***vsplit*** постоянна, то новая вершина и триплеты, соответствующие новым треугольникам уже находятся в массивах V и F соответственно, и для добавления новой вершины, достаточно увеличить на единицу *CurNumVerts*, а для добавления новых треугольников, увеличить *CurNumTrees* на количество новых треугольников. Для выполнения операции разделения вершины остается только модифицировать триплеты, где до выполнения операции находилась вершина *splitVert*, а теперь должна

оказаться новая вершина. Такие триплеты можно для каждой из операций ***vsplit*** выявить на этапе подготовки.

Параметризуем операцию разделения вершины, как ***vsplit***(*splitVert*,*numN*,*numF*,f_0,..,f_{numF}), где *splitVert* - вершина к которой применяется операция ***vsplit***, *numN* – количество треугольников добавляемых в модель, f_0,..f_{numF} – старые треугольники, которые необходимо модифицировать для выполнения операции ***vsplit***, *numF* – количество этих треугольников. Таким образом, операция разделения вершины будет выглядеть следующим образом [6]:

vsplit(*splitVert*, *numN*, *numF*, f_0, .., f_{numF}) {
CurNewTris = *CurNewTris* + *numN*;
Для каждого $f \in \{f_0, .., f_{numF}\}$ {
Заменить *splitVert* в *f* на *CurNewTris*;
}
CurNewVert = *CurNewVert* + 1;
}

Для того, что бы уменьшить количество вершин в текущей *PM*, определим операцию ***ecol***, которую параметризуем так же как и ***vsplit*** [6]:

ecol(*splitVert*,*numN*,*numF*,f_0,..,f_{numF}) {
CurNewTris = *CurNewTris* – *numN*;
Для каждого $f \in \{f_0, .., f_{numF}\}$ {
Заменить *CurNewVert* – *1* в *f* на *splitVert*
}
CurNewVert = *CurNewVert* – 1
}

В каждой из операций ***vsplit*** можно хранить также текущее отклонение *PM* от полной модели, таким образом, на этапе подготовке (процесс построения *PM* рассмотрен в [3, 4]) мы подготавливаем последовательность операций ***vsplit***$_0$… ***vsplit***$_n$, соответствующее этим операциям отклонение *error*$_0$… *error*$_n$. А на этапе отображения, будем изменять *PM* до достижения ею некоторого выбранного нами отклонения - *currentError*, просто выполняя над *PM* операцию ***vsplit***, пока текущее отклонение больше выбранного, или ***ecol*** если меньше. Для придания системе свойства масштабируемости будем изменять значение желаемого отклонения – *currentError* в зависимости от производительности платформы на которой производится запуск.

5. МОДИФИКАЦИЯ АЛГОРИТМА ОРИЕНТАЦИОННО-НЕЗАВИСИМЫХ ПРОГРЕССИВНЫЕ СЕТОК

Традиционный алгоритм упрощения ориентационно-независимых прогрессивных сеток диктует порядок следования граней в буфере объекта. Грани, исключение которых из геометрической модели объекта приведет к наименьшему искажению формы, лежат в конце буфера, и грани расположенные в буфере последовательно, как правило, не имеют смежных вершин. Однако, для того чтобы *Post T&L* кэш использовался оптимально и была достигнута максимальная скорость преобразования вершин акселератором, необходимо, чтобы рядом в буфере оказывались грани со смежными вершинами.

Расширим операции по управлению детализацией объекта операцией перестановки граней с целью поддержания субоптимального порядка следования вершин. В качестве способа представления перестановок выбраны последовательности циклических перестановок. Оптимизационный алгоритм имеет дело с представлением перестановок в виде таблиц соответствия, которые конвертируются в циклы на выходе алгоритма. Представление перестановок в виде циклов не требует дополнительной памяти при исполнении, за счет чего уменьшается используемая память и поток по шине, кроме того, поскольку циклическая перестановка тривиально обратима, не требуется хранить оба, прямое и обратное, ее представления.

Наличие этой операции позволяет снять требование на фиксированный порядок граней, так как операции перестановки могут уплотнять буфер после устранения граней на очередном шаге упрощения, и оптимизировать этот порядок граней, на каждом уровне детальности, исходя из эффективности использования *Post T&L* кэша.

Операцию циклической перестановки обозначим ***tswap*** (*numF*, f_0,.., f_{numF}), где f_0, .. f_{numF} – треугольники, порядок следования которых в буфере объекта необходимо модифицировать для максимальной утилизации *Post T&L* кэша, *numF* – количество этих треугольников.

tswap (*numF*, f_0,.., f_{numF}) {
Для каждого $f \in \{f_0, .., f_{numF-1}\}$ {
Заменить f_i на f_{i+1}
}
Заменить f_{numF} на f_0
}

Так же можно ввести обратную ей операцию ***tswap*** $^{-1}$ (*numF*, f_0,..,f_{numF}) , работающую с тем же набором данных что и ***tswap***.

tswap $^{-1}$ (*numF*, f_0,..,f_{numF}) {
Для каждого $f \in \{f_{numF-1}, .., f_0\}$ {
Заменить f_i на f_{i+1}
}
Заменить f_0 на f_{numF}
}

При этом последовательное применение операций ***tswap*** и ***tswap*** $^{-1}$, парметризованных одними и теми же данными, оставит буфер объекта неизменным.

На этапе подготовки массив граней *F*, для каждого уровня детальности, переупорядочивается таким образом, чтобы утилизация кэша была субоптимальна. Для группы операций ***vsplit***, соответствующей атомарному упрощению модели порождается одна или несколько опрераций ***tswap***, которые необходимы как для успешного применения операций ***vsplit*** и ***ecol***, так и для поддержания субоптимального с точки зрения загрузки кэшей порядка следования граней. Изменяя количество операции ***tswap*** можно добиться баланса между нагрузкой на центральный процессор и процессор графического акселератора.

В основе алгоритма, оптимизирующего порядок, лежит “жадный” алгоритм поиска субоптимальной последовательности. Алгоритм модифицирован таким образом, чтобы критерий оптимальности включал в себя:

- минимальность количества операций перестановок граней, необходимых для выполнения требования максимальной загрузки кэшей видеоускорителя;
- минимальность количества кэш-линий центрального процессора, затронутых этими операциями. Это необходимо для снижения нагрузки на *CPU* (Кэш-

линии, в которых располагаются грани, участвующие в текущих операциях ***vsplit***, являются с точки зрения алгоритма безусловно затронутыми).

Алгоритм начинает с буфера, состоящего из одной произвольной грани, и добавляет к ней грани по одной, так, чтобы каждая следующая грань по возможности использовала данные из кэша. При этом вышеуказанные критерии оптимизации обеспечиваются тем, что при выборе следующей грани, граням, которые уже находятся в данном месте буфера и тем, которые находятся в затронутых предыдущими операциями кэш-линиях – отдается предпочтение, путем использования весовых множителей.

Использование варианта алгоритма с оценкой ситуации на несколько шагов вперед значительно замедляет его работу, при очень незначительной прибавке эффективности, поэтому от него было решено отказаться.

Следует подчеркнуть, что предлагаемое расширение алгоритма детализации не накладывает ограничений на используемый алгоритм получения операций ***vsplit***, которыми осуществляется сама детализация, поэтому в качестве этого алгоритма может быть использован любой из известных [6][7], а предназначено для поддержания оптимального порядка граней с точки зрения утилизации *Post T&L* кэша, на произвольном уровне детальности.

6. РЕЗУЛЬТАТЫ

Оценка эффективности алгоритма производилась путем сравнения утилизации *Post T&L* кэша при использовании предложенного, расширенного алгоритма, и алгоритма детализации [6] без расширения.

В качестве критерия эффективности выбрано количество преобразований, которым подвергается вершина модели, усредненное по всем вершинам. При абсолютной эффективности использоания кэша критерий равен 1, а при абсолютной не-эффективности он равен количеству треугольников модели, умноженному на 3 и поделенному на количество вершин. Для регулярной прямоугольной сетки граней, при их случайном порядке, это число приближается к 6.

Для оценки качества работы алгоритма этот критерий вычисляется для каждого уровня детальности, после чего усредняется, в предположении, что уровни детальности моделей в сцене равновероятны.

В зависимости от модели, при использовании расширенного алгоритма, этот критерий составил 1.6-2.0, при этом стандартный алгоритм давал значения, близкие к наихудшим (4-6).

Для финальной оценки результатов использовались замеры производительности отрисовки тестовой сцены. Следует заметить, что в силу конвейерности процесса построения кадра, наибольшее улучшение производительности происходит на тех моделях, для которых преобразование вершин является “узким горлом”, и ограничено сверху тем, насколько загрузка блока преобразования вершин превышает загрузку остальных фаз конвейера. В случае, если оптимизация использования *Post T&L* кэша значительна (2-3.5 раза), и снижает загрузку этой части конвейера настолько, что делает остальные его части более значимыми, улучшение является мерой разбалансированности конвейера немодифицированным алгоритмом. Улучшение совокупной производительности отрисовки при использовании модифицированного алгоритма по сравнению с не модифицированным [6], в зависимости от модели, составляет до 2 раз, при этом загрузка *CPU* за счет усложнения алгоритма возрастает на 20%.

7. ЗАКЛЮЧЕНИЕ

Известные алгоритмы непрерывной детализации объектов диктуют порядок следования граней в буфере объекта. Алгоритмы визуализации, используемые в современных графических акселераторах, для оптимальной работы требуют другого порядка граней, что связано с наличием кэш-памяти преобразованных вершин, причем эти требования противоречат друг другу.

Алгоритм динамической детализации основанный на ориентационно-независимых прогрессивных сетках был модифицирован таким образом, чтобы порядок граней, в основном, определялся требованиями кэш-памяти, а преобразования детализации дополнены преобразованиями порядка граней с целью поддержания субоптимального порядка. Предложенный алгоритм достигает баланса между нагрузкой на центральный процессор и процессор графического акселератора. Его применение позволило достичь двукратного увеличения производительности процесса отображения, по сравнению с классическим алгоритмом.

8. ЛИТЕРАТУРА

[1]. Willem H. de Boer, *Fast Terrain Rendering Using Geometrical MipMapping*,E-mersion Project, October 2000, http://www.connectii.net/emersion

[2]. M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, M. B. Mineev Veinstein, *ROAMing Terrain: Real time optimally adapting meshes*, IEEE Visualisation’97, Nov 1998 pp.81-88.

[3]. H. Hoppe, *Smooth view-dependent level-of-detail control and its application to terrain rendering*, IEEE Visualization ’98, Oct. 1998.

[4]. S. Melax, A Simple, *Fast and Effective Polygon Reduction Algorithm*, Game Developer Magazine, Nov 1998, pp. 44-49

[5]. *NVIDIA GPU Programming Guide Version 2.4.0* 2005 http://developer.nvidia.com/object/gpu_programming_guide.html

[6]. T. Forssyth *“Comparison of VIPM Methods”,* Game Programming Gems 2, Charles River Media 2001

[7]. J. Svarovsky “*View-independent Progressive Meshing*”, Games Programming Gems 1, Charles River Media 2000

Hardware Friendly Continuous Level of Details.

Abstract

We propose an improved dynamic level of detail algorithm, based on progressive meshes, that solves the problem of *T&L* cache utilization present in the classical approach, thus significantly improving the performance.

Keywords: *virtual reality, hardware accelerated real-time rendering, progressive meshes.*

Kuzikowski Stanislaw, Belago Igor, Gladky Denis, Elykov Nikolay – Institute of Automation and Electometry SB RAS.
stas@sl.iae.nsk.su