**Организация непрерывных LOD ландшафтов с использованием Адаптивных КвадроДерьев**

Александр Винюков

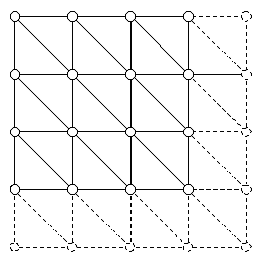
**Введение: в чем проблема?**

Рендеринг ландшафтов является сложной проблемой при програмировании реалистичной графики. В настоящий момент мы находимся в интересной точке развития технологий рендеринга (т.е. "оцифровки" данных, по которым строится ландшафт), поскольку увеличившаяся пропускная способность современных видеокарт в совокупности с опубликованными исследованиями на тему алгоритмов реального времени LOD meshing дают возможность современным графическим движкам отрисовывать сильно детализированные ландшафты. Тем не менее большинство используемых технологий предлагают компромис между размером ландшафта и его детализированностью.

**Проблемы**

Что мы хотим получить от ландшафта? Один сплошной меш, начинающийся с заднего плана и продолжающийся до горизонта без разрывов и Т-узлов. Мы хотим видеть большую область с различной детализацией, начиная от выпуклостей под ногами и заканчивая горами на горизонте. Для определенности скажем, что мы хотим оперировать размерами от 1 метра до 100000 метров с пятью уровнями детализации.

Как этого добиться? Метод решения в лоб не подойдет для средних компьютеров наших дней. Если мы заведем сетку 100000х100000 16-битных значений, и просто попытаемся ее отрисовать, то получим 2 большие проблемы. Во-первых, это проблемы треугольников - мы будет посылать до 20 млн. треугольников на каждый кадр. Во-вторых, проблема памяти - для хранения карты высот нам понадобится приблизительно 20 Гб памяти. Еще не скоро мы сможем использовать такой метод и получать хорошие результаты.



Лобовое решение для карты высот.

Несколько ранее опубликованных методов успешно решают проблему треугольников. Наиболее широко использующиеся - семейство рекурсивных алгоритмов [1], [2], [3] (смотрите в [конце](http://www.xaoc.ru/a198/#links#links) список литературы). Используя их мы можем спокойно обработать наш меш и рендерить похожий ландшафт, используя интелектуально выбираемые на лету несколько сотен треугольников из 10 милионов.

Тем не менее, все еще остается проблема памяти, так как на хранение карты высот требуется около 20 гб плюс некоторое количество на алгоритм интелектуального выбора вершин.

Одним из явных решений является уменьшение детализации путем увеличения размера ячейки. 1Kx1K является хорошим практически используемым размером карты высот. Недавно выпущенная игра TreadMarks использует набор данных 1k x 1k с великолепным эффектом [4]. К сожалению, 1k x 1k далеко от 100k x 100k. Итак, мы пришли к компромису между детализацией переднего плана и размером ландшафта.

Решение, предлагаемое в этой статье - это использовать адаптивное квадродерево (quadtree) вместо регулярной сетки для представления информации о высоте. Используя это квадродерево мы можем закодировать данные с различным разрешение в различных областях. К примеру, в автосимуляторе нам требуется высокая детализация дороги и ее окресностей, в идеале до каждой выпуклости, но окружающие пустоши, по которым мы не можем ехать, можно отображать и с меньшей детализацией. Нам нужна лишь информация для генерации холмов и долин.

Квадродерево также можно использовать и для другого подхода к решению проблемы памяти: процедурная детализация. Идея состоит в том, чтобы определять форму ландшафта на грубом уровне и автоматически генерировать высокодетализированный ландшафт на лету вокруг точки зрения. По адаптивной природе квадродерева все эти сгенерированные детали могут быть отброшены после смены точки зрения, таким образом высвобождая память для создания процедурной детализации в другом регионе.

**Алгоритм: Создание меша**

Алгоритм базирован на [1], но также использовались мысли из [2] и [3]. В отличии от [1] существует несколько ключевых модификаций, но базис тот же самый, поэтому мы будем придерживатся терминологии [1].

Рендер делится на 2 части: первую называем Update(), а вторую Render(). Во время Update(), мы решаем, какие вершины включить в выводимый меш. Затем, во время Render() уже генерируем по этим вершинам треугольники. Начнем с простой сетки 3х3 для объяснения действия Update() и Render(). Во время Update() мы рассматриваем каждую из необязательных вершин и решаем, включать ли ее в меш. Следуя терминологии [1] мы говорим, что вершина будет использована в меше тогда и только тогда, когда она "включена".

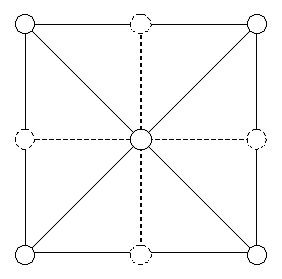


Рис. 2. Сетка 3х3. Пунктирные линии и вершины необязательны для LOD ммеша.

Пусть центральная и угловые точки включены (из иерархии выше). Тогда задача состоит в том, чтобы вычислить для каждой из точек, лежащих на ребрах, ее состояние, руководствуясь некоторыми LOD вычислениями, основанными на точке зрения и параметрах вершины.

Как только мы узнаем, какие из вершин включены, мы можем воспользоваться функцией Render(). Это просто - мы составляем веер треугольников (triangle fan) вокруг центральной вершины, включая каждую включенную вершину по часовой стрелке. См. рис. 3.

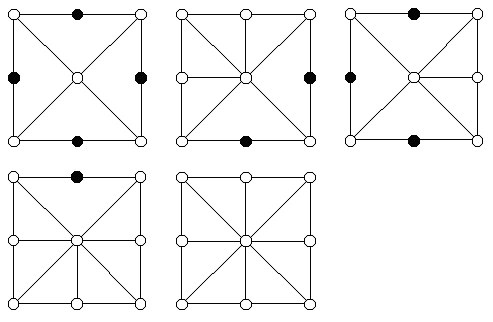


Рис. 3. Сетка 3х3. Выключенные вершины помечены черным.

Для Update() и Render() адаптивного квадродерева мы повторяем изложенный выше процесс при помощи рекурсивного деления, начиная с сетки 3х3. В процессе деления мы можем получить новые вершины и обрабатываем их также как и вершины исходного квадрата. Но, для того чтобы избежать разрывов, мы руководствуемся несколькими следующими правилами.

Во первых, мы можем поделить любую комбинацию из четырех квадратов. Когда мы делим квадрант, мы обрабатываем его как подквадрат и включаем его центральную вершину. Также мы должны включить 2 лежащие на ребрах вершины родительского квадрата, которые являются углами нашего подквадрата (рис. 4). То есть включение подквадрата означает включение его четырех угловых и центальной вершины.

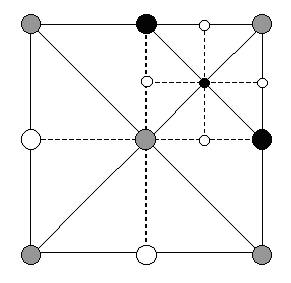


Рис. 4. Деление Северо-Восточного (СВ) квадранта квадрата. Про сервые вершины мы уже знаем что они включены, но черные включаются в результате деления квадрата.

Но, кроме того, вершина, лежащая на ребре подквадрата, является общей для соседнего подквадрата. Таким образом, когда мы включаем реберную вершину, надо убедится что и в соседнем подквадрате она тоже включена. Включение ее в соседнем квадрате может, в свою очередь, включить ее у нас, т.е. флаг включения должны передаватся через квадродерево. Синхронизация этих флагов у соседних квадратов необходима для обеспечения целосности меша. См. [1] для хорошего описания правил зависимости включения.

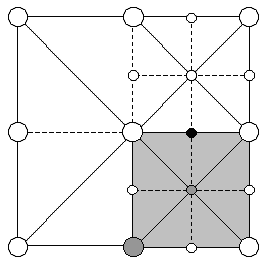


Рис. 5. Во время Update() для NE квадранта мы принимаем решение включить черную вершину. Так как эта вершина общая с SE квадрантов (помеченным серым), то мы должны включить этот квадрант тоже. Включение SE вкадранта в свою очередь заставит нас включить помеченные серым вершины.

После того как мы закончим с Update() мы можем вызвать Render(). Рендер очень прост - все вычисления по сохранению целостности меша уже были выполнены в Update(). Идея, на которой основан Render() - это рекурсивный вызов Render для включенных подквадратов и затем отрисовка всех треугольников квадрата, которые не были покрыты включенными подквадратами.

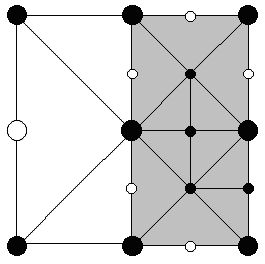


Рис 6. Пример меша. Включенные вершины помечены черным. Серые треугольники отрисованы во время рекурсивного вызова Render() для сопоставленных им подквадратов. Белые треугольники отрисоввываются во время изначального вызова Render() для этого квадрата.

**Алгоритм: Обработка вершин и квадратов**

Перейдем к самому вычислению - должна ли вершина быть включена. Для этого существует несколько методов. Все, которые я принял во внимание, называются "vertex interpolation error" (ошибка интерполяции вершины), или, короче, vertex error. Это разница между высотой вершины и высотой ребра в треугольнике, который апроксимирует вершину, когда та выключена (Рис. 7). Вершины с большей ошибкой должны быть предпочтительнее вершин с маленькой ошибкой. Другой вариант основан на растоянии вершины от точки зрения. Интуитивно, имея 2 вершины с одинаковой ошибкой мы должны выбрать более ближнюю.

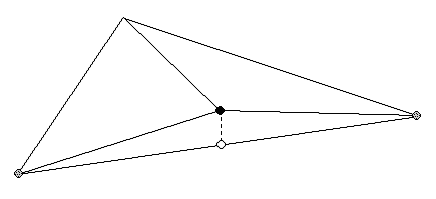


Рис. 7. Vertex interpolation error. Когда вершина включена или выключена, меш меняяет форму. Максимальное изменение, получающееся при включении вершины показано пунктирной линией. Величина изменения есть разница между реальной высотой вершины (черная точка) и высотой ребра под ним (белая точка). Белая точка - это просто середина между 2 серыми точками. (Wolverene: Середина - так как мы рассматриваем измельчающиеся квадраты в quadtree и точка деления лежит ровно посередине).

Также есть другие влияющие факторы. К примеру, в [1] принимается во внимание направление между вершиной и точкой зрения. Идея заключается в screen space error (экранной ошибке) - интуитивно vertex error менее видима чем более вертикально данное направление. В [1] описана вся эта математика в деталях.

Но я не думаю что screen space error является хорошей метрикой по двум причинам: во-первых, он игнорирует текстурную перспективу и depth buffering errors - даже если вершина не движется на экране, т.к. она сдвигается строго пенпердикулярно к/от точки зрения, Z значение влияет на коррекцию перспективы так же как и depth-buffering. Во-вторых, точка зрения прямо вниз является как легким случаем для LOD ландшафтов, так и не типичным случаем.

На мой взгляд, нет смысла оптимизировать для нетипичного легкого случая. Случай, когда ось зрения более горизонтальна и большая часть ландшафта видима определяет наименьший framerate, и, следовательно, эффективность алгоритма.

Вместо screen-space geometric error я предлагаю делать подобный тест в трехмерном пространстве с ошибкой пропорциональной дистанции. Он включает только 3 величины: L1 норму вектора между вершиной и точкой зрения, vertex error и константный "порог детализации"(Threshold).

Vertex test:

L1 = max(abs(vertx - viewx), abs(verty - viewy), abs(vertz - viewz));

enabled = error \* Threshold < L1;

На практике использование L1 нормы означает большую глубину делений вдоль диагоналей горизонтальной плоскости ландшафта. Я не смог заметить данный эффект глазом, так что я не думаю что об этом стоит беспокоиться. [4] и другие используют view-space-z вместо L1 нормы, что теоретически является даже более правильным чем истинная дистанция между вершиной и точкой зрения. Несмотря на это, L1 по-моему работает наилучшим образом и в [3] она тоже используется.

Можно использовать Threshold как регулятор подстройки скорость/качество, но он не имеет четкой геометричесой интерпретации. Грубо говоря, он означает: для данного растояния z, худшей ошибкой с которой я буду мирится будет z / Threshold. Конечно, вы можете произвести некоторые вычисления угла зрения и связать Threshold с максимальной пиксельной ошибкой, но я никогда не рассматривал его более чем значение для подстройки отношения качество/скорость.

Вот так работает включение вершин. Но что является более важным, как работает включение подквадратов во время Update()? Ответом является box test. Он задает следующий вопрос: рассматриваем выровненный по осям 3D Box, содержащий участок ландшафта (т.е. элемент quadtree), и знаем соответвующую ему максимальную vertex error всех вершин внутри. Может ли vertex enable test вернуть истину для какой-либо вершины из этого квадрата? Если да, то делим.

Прелесть этого метода в том, что делая BoxTest мы можем одним махом отсечь тысячи вершин. Это делает Update() полностью масштабируемым: стоимость его не зависит от размера всего набора данных, а лишь от размера данных, включенных в текущий LOD меш. И, как побочное явление, предпросчитанный вертикальный размер Bounding Box может использоватся для frustim culling. (примечание wolverene - координаты x и z мы уже знаем из выбора квадрата как элемента quadtree, т.е. для построения BoundBox нам надо знать лишь вертикальный его размер).

Box Test консервативен в том смысле что вершина, из-за которой Box Test вернул "Да" (т.е. необходимость деления) может находится на другой стороне bound box от ближней к точке зрения стороны bound box, и, таким образом, сама вершина может этот тест провалить. Но на деле это не влечет больших накладных расходов - несколько box test и vertex test.

Box test имеет следующий вид, очень похожий на VertexTest:

Box test:

bc[x,y,z] == coordinates of box center

ex[x,y,z] == extent of box from the center (i.e. 1/2 the box dimensions)

L1 = max(abs(bcx - viewx) - exx, abs(bcy - viewy) - exy, abs(bcz - viewz) - exz)

enabled = maxerror \* Threshold < L1

**Особенности: Детали**

В предыдущем разделе была описана суть алгоритма, но оставлена в стороне маса деталей, некоторые из которых являются ключевыми. Во-первых, где, собственно хранится информация о высоте? В предыдущих алгоритмах существовала регулярная сетка высот, на которой неявно [1] & [3] или явно [3] определялся меш. Главной инновацией в моем методе является то, что данные хранятся собственно в адаптивном quadtree. В результате получаем 2 главные выгоды. Во-первых, данные располагаются адаптивно в соответствии с той их частью, что нужна приложению; таким образом меньшее количество данных относится к сглаженной части ландшафта, в которой не предполагается нахождение камеры. Во-вторых, дерево может динамически расти или уменьшатся в зависимости от нахождения точки зрения наблюдателя; процедурная детализация может быть добавлена на лету в регионе, в котором находится камера и впоследствии удалена, как только камера покинет данный регион.

Для того, чтобы хранить информацию о высотах каждый элемент quadtree должен хранить информацию о высоте своей центральной вершины и по крайней мере двух вершин, лежащих на ребрах. Все другие высоты хранятся в соседних узлах дерева. К примеру, высота угловых вершин приходит из родительского квадрата и его соседей. Оставшиеся высоты реберных верших хранятся в соседних квадратах. В текущей реализации я храню высоту центральной вершины и все 4 высоты реберных вершин. Это упрощает работу за счет того, что необходимые для обработки квадрата данные доступны как данные квадрата или параметры функции. Недостатком является то, что каждая реберная точка хранится к дереве дважды.

Так же в текущей реализации одно и то же quadtree используется для хранения высоты и для построения меша. Вообще-то хорошо бы его разделить на 2 - одно для хранения карты высот, второе для построения меша. Потенциальные прелести такого подхода изложены ниже.

Большинство трюков реализации основано на общих для двух соседних квадратов реберных вершин. К примеру, такой вопрос - какой квадрат должен выполнять vertex-enable test для реберной вершины? Мой ответ - каждый квадрат проверяет только свои южную и восточную реберные вершины и полагается на проверки северного и западного соседа для проверки двух других вершин.

Другой интерестный вопрос - надо ли нам сбрасывать все флаги перед Update() или же продолжать с состоянием, оставшимся от предыдущего цикла отрисовки? Мой ответ - продолжаем работу от предыдущего состояния, как в [2], но не как в [1] и [4]. Это ведет к большей детализированности - мы проверяли условия на включение, но когда мы можем выключить вершину или квадрат? Как мы помним из алгоритма Update() включение вершины заставляет включится зависящие от нее вершины, и так далее по дереву. Мы не можем просто так выключить вершину в середине одной из таких цепей зависимостей, если вершина зависит от какой-либо другой включенной вершины. Иначе у нас получатся разрывы в меше, либо важные включенные вершины не будут отрисованы.

На рис. 8 видно что реберная вершина имеет 4 соседних квадрата, которые используют ее как угловую вершину. Если любой из этих подквадратов включен, то должна быть включена данная вершина. Поскольку квадрат включен когда центральная его вершина включена, то одним из подходов будет проверить все соседние подквадраты перед отключением. Тем не менее, в моей реализации это будет слишком тяжело, так как для нахождения соседних квадратов придется обегать ве дерево. Вместо этого я считаю число ссылок для каждой реберной вершины. Оно равно числу подквадратов, от 0 до 4, которые включены. Это означает что каждый раз когда мы включаем/выключаем квадрат, мы должны обновить число ссылок в двух вершинах. К счастью, число ссылок меняется от 0 до 4, т.е. все это можно запаковать в 3 бита.

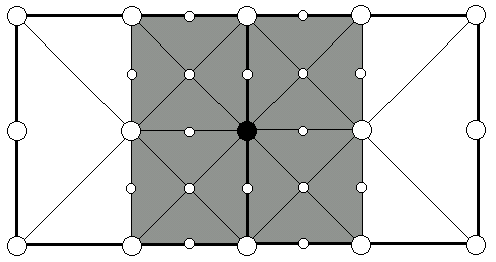


Рис. 8. Каждая реберная вершина имеет 4 соседних подквадрата, которые используют ее как угловую. Если любой из этих квадратов включен, то и вершина должна быть включена. К примеру, черная вершина должна быть включена если включен один из серых квадратов.

Таким образом выключающий тест очень прост: если вершина включена, число ссылок равно 0 и vertex test для текущей точки камеры возвращает false, выключаем вершину. Иначе не трогаем ее. Условия выключения квадрата тоже довольно прямолинейны: если квадрат включен и он не корень дерева, и нет включеных реберных вершин и нет включеных подквадратов, квадрат проваливает BoxTest, выключаем его.

**Особенности: Память**

Очень важной чертой этого или любого другого LOD метода является потребление памяти. В полном quadtree один квадрат эквивалентен трем вершинам обычной сетки высот, так что требуется сделать структуру квадрата как можно компактнее. К счастью, Render() и Update() методы не требуют от каждого квадрата информации по всем 9 вершинам. Вот список требуемых данных:

· 5 высот (углы и центр)

· 6 значений ошибок (вершины на восточном и южном ребрах и 4 подквадрата)

· 2 счетчика включенных подквадратов (для вершин на восточном и южном ребрах)

· 8 1-битовых флагов включения (по 1 для каждой вершины и каждого подквадрата)

· 4 указателя на подквадраты

· 2 значения высоты для минимального/максимального вертикального размера

· 1 1-битный флаг, показывающий что этот квадрат не может быть удален.

В зависимости от нужд приложения значения высот могут быть комфортно упакованы в 8 или 16 бит. Значения ошибок могут использовать тот же самый формат, но, используя нелинейное сжатие вы можете запаковать их еще больше. Все счетчики ссылок и статистический флаг поместятся в 1 байт. Флаги включения тоже пакуются в 1 байт. Размер указателей на подквадраты зависит от максимального числа узлов, которые могут быть использованы. Обычно это сотни или тысячи, так что я использую 20 бит на каждый указатель как минимум. Минимальное и максимальное значения высоты тоже могут быть сжаты различными способами, но 8 бит на каждый выглядит разумным минимумом. Все вместе это занимает 191 бит (24 байта) на квадрат при 8-битных значениях высоты. 16-битные значения высот требуют 29 байтов. 32-байтный размер размер квадрата выглядит хорошей целью для бережливой реализации. 36 байтов я вынужден использовать, так как я не пытался упаковывать указатели на подквадраты. Другой трюк - использовать фиксированный массив с заменой алокаторов для quadsquare::new и quadsquare::delete. Это сжимает 4 байта накладных расходов стандартного для С++ аллокатора (как я предполагаю) до 1 бита.

Существует много трюов и схем компресии для того чтобы сжать данные еще сильнее, но они увеличивают сложность и уменьшают производительность. В любом случае, 36 байтов на 3 вершины не совсем плохо. Это 12 байтов на вершину. В [1] было достигнуто 6 байтов на вершину.

С одной стороны это очень много, но с другой стороны адаптивная структура quadtree позволяет хранить разреженные данные в ровных областях или областях, для которых не требуется высокая детализация. В то же время в высоко важных областях можно достигнуть высокой детализации; к примеру, в той же игре-автосимуляторе можно хранить даже неровности и рытвины на дороге.

**Особенности: Геоморфинг**

[2] и [3] также используют морфинг вершин или, по другому, геоморфинг. Идея в том, что при включении вершин получаются резкие скачки между предыдущим мешом, в котором данная вершина была отключена и отрисованным в данном кадре, в котором вершина была включена. Для того, чтобы избавится от этого эффекта применяется плавная анимация из интерполированного положения вершины в ее настоящее значение. Это отлично выглядит и устраняет неприятные эффекты скачков, смотри McNally's TreadMarks для хорошей иллюстрации данного метода.

К несчастью, выполнение геоморфинга требует хранения еще одного значения высоты для анимируемой вершины, что представляет собой реальную проблему для алгоритма адаптивных quadtre в той его реализации, которая была описана. Потребуется несколько дополнительных байтов на каждую вершину, что не так уж легко. В [3] такие данные хранятся на каждую вершину, но в [2] этого стараются избежать, так как на деле дополнительное значения высоты должно хранится лишь для вершины, которая включена в данный меш, но не для всего набора данных.

Есть три суждения по поводу геоморфинга. Первый подход - потратить дополнительную память на хранение дополнительного значения высоты для каждого меша. Второй альтернативой является улучшить алгоритм так, чтобы достигнуть действительно относительно маленьких ошибок, т.е. геоморфность просто не потребуется. К тому же согласно закону Мура вероятно это вскоре будет реализовываться на уровне hardware. Третьей альтернативой является разделить quadtree на 2 дерева: одно для хранения данных (дерево высот), второе для хранения отображаемого меша (дерево меша). В дереве высот будут хранится все высоты и предпросчитанные ошибки, но ничего из временных данных, таких как флаги включения, число ссылок, веса морфинга и так далее. При построении дерева меша можно не задумыватся об ограничениях памяти, поскольку его размер пропорционален числу деталей, отрисовывающихся в данный момент. В то же время дерево высот может сохранить немного памяти, так как оно является статическим и, таким образом, из него можно удалить множество ссылок на детей.

Кроме того такая концепция в дополнение к уменьшения требуемой памяти может увелить локальность данных и улучшить использование кэша алгоритмом.

**Приложения**

Работающая реализация

Графический движок Soul Rider является закрытым и вряд ли будет открыт в обозримом будущем, но я переписал основы алгоритма в качестве демонстрации для этой статьи. Эти исходники могут быть свободно использованы для изучения, экспериментов, модификации и включения в ваш собственный коммерческий/некоммерческий проект. Я лишь прошу упомянуть меня.

Я не использовал никаких запаковок данных в демо-коде. Это хорошая область для экспериментов. Также я не использовал отсечение по пирамиде видимости, но все необходимые данные доступны.

Упражнения для читателя

В дополнение к запаковке данных упомяну о некоторых других вещах, включенных в движок Soul Ride, но не включенных в демо. Одной из больших является однозначная-полноландшафтная система текстурирования (wolverene: наверное, имеется ввиду что на весь ландшафт накладывается 1 текстура), описание которой выходит за рамки данной статьи.

Другой вещью, с которой я не эксперементировал, но которую легко понять по демо-коду - это процедурная детализация по запросу. По-моему, это одно из перспективных направлений развития компьютерной графики. Просто не видно другого способа хранить и моделировать виртуальные миры в деталях достаточных для достижения богатства отображения реального мира. Я думаю что алгоритм quadtree в силу его масштабируемости может быть полезен для программистов, работающих над процедурной детализацией.

Другим полезным решением является подкачка по запросу подсекций дерева. На деле это не так сложно: просто следует завести специальный флаг для определенных подквадратов; в них будет содержаться ссылка на все гиганское поддерево, хранящееся на диске с просчитанной и хранящейся в обычном дереве максимальной ошибкой. Когда Update() старается включить "специальный" квадрат, последует пауза, подкачка данного квадрата с диска и подключение его в дерево, а затем возобновление работы Update(). Реализация этого в фоновом режиме без задержек была бы еще интерестнее, но я думаю, выполнима. В результате получим бесконечную подкачку. Процедурная детализация по запросу основана на той же самой идее: вместо того, чтобы забивать диск предподготовленными данными, вы просто запускаете алгоритм увеличения детализации на лету.

И еще одной интересной работой было бы обнаружение узких мест в алгоритме.

**Список литературы**

[1] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust and Gregory A. Turner. "Real-Time, Continuous Level of Detail Rendering of Height Fields". In SIGGRAPH 96 Conference Proceedings, pp. 109-118, Aug 1996.

[2] Mark Duchaineau, Murray Wolinski, David E. Sigeti, Mark C. Miller, Charles Aldrich and Mark B. Mineev-Weinstein. "ROAMing Terrain: Real-time, Optimally Adapting Meshes." Proceedings of the Conference on Visualization '97, pp. 81-88, Oct 1997.

[3] Stefan Rцttger, Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. Technical Report 13/1997, Universitдt Erlangen-Nьrnberg.

[4] Seumas McNally. <http://www.longbowdigitalarts.com/seumas/progbintri.html> This is a good practical introduction to Binary Triangle Trees from [2]. Also see <http://www.treadmarks.com/>, a game which uses methods from [2].

[5] Ben Discoe, <http://www.vterrain.org/> . This web site is an excellent survey of algorithms, implementations, tools and techniques related to terrain rendering.