 DOI 10.25699/SSSB.2021.38.4.012

05.13.17

ПРОЦЕДУРНАЯ ГЕНЕРАЦИЯ МАССИВНОЙ 3D-ГЕОМЕТРИИ С ИСПОЛЬЗОВАНИЕМ УЛУЧШЕННОГО АЛГОРИТМА MARCHING CUBES

А.А. Третьяков

Пермский национальный исследовательский политехнический университет, г. Пермь

Процедурная генерация, или создание контента во время работы программы, это сложное направление, которое требует не только понимания 3D-графики, но и навыков программирования графики, что часто сводится к изучению работы графических процессоров. Из-за такой сложности разработчики часто используют уже готовые инструменты для создания контента. Такие инструменты обобщают и упрощают работу, предоставляя большой заготовленный набор функции, который можно использовать не зная программирования вовсе. К сожалению, обобщение часто приводит к уменьшению гибкости и вводит новые ограничения. Статистика показывает, что использование процедурной генерации, для создания массивной 3D-геометрии, невозможно при использовании готовых инструментов с уже заготовленными функциями. Такие инструменты не позволяют воплотить огромные масштабы массивной геометрии в жизнь из-за различных ограничений. Кроме того, существующие алгоритмы создания 3D-геометрии часто не учитывают применение этих алгоритмов для создания массивной 3D-геометрии, например, планет. Рассматриваемый в этой работе алгоритм Marching Cubes также не учитывает применение алгоритма для создания массивной геометрии, из-за чего применение этого алгоритма в таких целях будет иметь много ограничений и много недостатков. Но данный алгоритм выбран не случайно, он обладает большой популярностью и мы поговорим почему. Данная работа фокусируется на представлении новой модификации на уже существующий алгоритм Marching Cubes в целях применения его в рамках массивной геометрии. Данный алгоритм найдет применение в компьютерных играх с космической тематикой, наш алгоритм позволяет создавать массивную 3D-геометрию планетарных масштабов даже на слабых компьютерах без особых затрат по ресурсам. Кроме того, наш алгоритм позволяет изменять сгенерированную геометрию в реальном времени, без задержек по времени, что так важно компьютерным играм.

Ключевые слова: процедурная генерация, ускорение, графический процессор, gpu, marching cubes.

ВВЕДЕНИЕ

Процедурная генерация – это процесс создания контента (геометрии, сценариев, и т. д.) во время работы компьютерной программы. Процедурная генерация находит применение во многих сферах, от медицины до кинематографа, но чаще всего процедурная генерация применяется в игровой индустрии, а именно, в компьютерных играх. Говоря о компьютерных играх, очень часто весь контент, такой как 3D-модели окружения, персонажей, ландшафта, оружий и прочего создается 3D-художниками, которые «воплощают» в трехмерном пространстве идеи концепт-художников, которые часто работают в 2D, потому что это проще. Процедурная генерация – это противоположность этого процесса. Если обычно контент создают заранее (т. е. до того как игра будет запущена), то процедурная генерация создает контент во время работы игры и в качестве художников, или создателей, контента выступают программисты. Но зачем нагружать компьютер каждый раз создавая одно и тоже во время работы программы, например, 3D-геометрию? Суть процедурной генерации заключается в случайностях. Процедурная генерация почти всегда использует различные псевдослучайные генераторы случайных чисел и алгоритмы-шумы для того, чтобы создаваемая геометрия каждый раз была уникальной. Многие люди не знакомые с 3D-графикой с трудом понимают термин процедурная генерация, поэтому мы приведем пример: нашей игре нужна земля, по которой мог бы ходить игрок. Мы можем поручить создание ландшафта 3D-художнику и получить статичную геометрию, которая не изменится. Если бы нам нужен был ландшафт для другого уровня игры, то 3D-художник начинал бы заново делать все с нуля. В случае с процедурной генерацией мы поручим это задание программисту, который напишет алгоритм, использующий генераторы случайных чисел, который бы создавал уникальную геометрию каждый раз запустив алгоритм.

Конечно же, не все так хорошо, часто используя процедурную генерацию очень тяжело достичь уровня художника в плане качества. Если созданная художником геометрия, например ландшафт, может выглядеть очень реалистично, то в случае процедурной генерации этого очень тяжело достичь в связи с тем, что у художника есть неограниченное количество времени, а у алгоритма нет, пока алгоритм работает – игрок ждет и скучает, а это не хорошо. Но со временем алгоритмы становились все лучше и лучше, и сейчас, мы уже вполне можем поравняться с

уровнями художников, например в задачах создания ландшафтов или планет.

Процедурная генерация часто требует не только знания программирования и 3D-графики одновременно, но и понимания низкоуровневой работы графического процессора. Поэтому, разработчики небольших проектов часто выбирают уже готовые инструменты, называемые игровыми движками.

Игровой движок – это программный инструмент, который предоставляет различный функционал для создания графического приложения. Не редко, игровые движки включают инструменты для процедурной генерации, которые обобщают процесс создания некоторой 3D-геометрии. К сожалению обобщения вносят ограничения и не предоставляют большой гибкости, тем более, когда мы говорим о массивной геометрии, такой как планеты, например. В добавок, существующие алгоритмы создания 3D-геометрии часто не учитывают применение этих алгоритмов для создания массивной 3D-геометрии.

В этой работе мы рассмотрим алгоритм Marching Cubes [1], который также не учитывает применение алгоритма для создания массивной геометрии, и опишем нашу модификацию этого алгоритма, которая позволяет эффективно применять его при создании массивной геометрии, такой как планеты, даже на слабых компьютерах без больших задержек по времени. Кроме того, мы также объясним почему мы выбрали именно этот алгоритм. Наш алгоритм также учитывает использование его в компьютерных играх и позволяет изменять геометрию в реальном времени (т. е. во время работы программы) без задержек по времени, что так важно для компьютерных игр.

1.1. АЛГОРИТМ MARCHING CUBES ДЛЯ СОЗДАНИЯ ПРОЦЕДУРНОЙ 3D-ГЕОМЕТРИИ

Алгоритм Marching Cubes был выбран не случайно, так как мы говорим об интерактивных графических приложениях – компьютерных играх, то стоит сфокусироваться на производительности в первую очередь, так как компьютерные игры – это очень сложные программы, как в плане создания, так и в плане нагрузки на компьютер.

Компьютерные игры уже давно начали использовать ресурсы графического процессора, чтобы значительно увеличить производительность в тех задачах, которые с трудом выполнялись на центральном процессоре. Со временем разработчики находили все больше и больше применений графических процессоров при разработке компьютерных игр, но нет никакого смысла переносить все вычисления на графический процессор. Общее правило такого: если задачу можно разделить на независимые логические элементы, то стоит задуматься о переносе вычислений на графический процессор. Любой графический процессор имеет такую архитектуру, которая позволяет выполнять огромное количество параллельных задач. Центральный процессор же, наоборот, может выполнить небольшое количество параллельных задач, но он сохраняет высокую производительность в случае ветвления программы. Например, возьмем алгоритм Нелдера-Мида [2] используемый для нахождения экстремума функции нескольких переменных. Любой, кто работал с этим алгоритмом знает, что весь алгоритм пропитан ветвлениями типа "если-то", в связи с чем, нет никакого смысла реализовывать этот алгоритм на графическом процессоре, так как архитектура этого процессора предназначена для решения другого типа задач, поэтому такой алгоритм, работающий на центральном процессоре, будет выполняться быстрее.

Как мы уже сказали ранее, компьютерные игры – это очень сложные и требовательные программы, если мы не будем использовать графический процессор, то это будет равносильно использованию только 50% данных мощностей компьютера. Поэтому, нашим ключевым критерием при выборе алгоритма была возможность разделить алгоритм на независимые логические элементы. Алгоритм Marching Cubes – это как раз то, что нам нужно. Разберем принцип работы оригинального метода.

Начало алгоритма заключается в определении трехмерной матрицы точек, которые будут находится на равном друг от друга расстоянии. В области этой трехмерной матрицы и будет находится наша 3D-модель, которую мы будем создавать во время работы программы. Каждая точка этой трехмерной матрицы может иметь два состояния: внутри или снаружи, что определяет положение точки относительно финальной 3D-модели. Для большего понимания рассмотрим 2D-вариант алгоритма Marching Cubes (который называется Marching Squares) для создания круга [3] (см. рис. 1). Сначала алгоритм требует создания 2D-матрицы точек, находящихся на одном и том же расстоянии друг от друга.

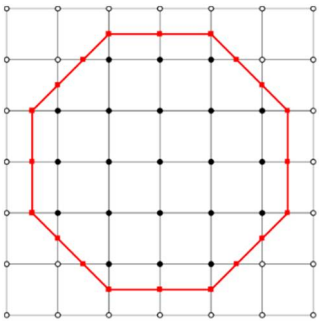

Рис. 1. Точки внутри круга имеют состояние "внутри", а точки вне круга, состояние "снаружи", на границе смены состояний алгоритм проводит линию

Алгоритм Marching Cubes в 2D (как и в 3D) использует итерационный подход: алгоритм проходит

по каждому квадрату матрицы и на основе состояний 4-х точек квадрата он создает новую точку на границе между двумя точками с разными состояниями. После этого алгоритм соединяет новые точки и переходит к следующему квадрату, отсюда и название алгоритма "марширующие квадраты/кубы".

Финальная фигура определяется функцией, в случае 2D-варианта, это обычное уравнение окружности. Алгоритм, на каждой итерации высчитывает длину вектора от центра круга до каждого из 4-х точек (углов) квадрата, высчитывает длины векторов и проверяет находится ли данная точка "внутри" радиуса круга (меньше ли длина вектора радиуса круга), если это так, то данная точка получает состояние "внутри". Такая же логика используется и в 3D-пространстве.

Из рисунка 1 видно, что результирующая фигура не очень похожа на круг. Это можно легко исправить используя линейную интерполяцию. На каждой грани квадрата, мы имеем по точке на каждом конце грани. Для каждой точки вычислим значение функции и за место того, чтобы просто использовать середину грани, мы посмотрим на каком месте грани происходит переход через границу радиуса, т.е. мы определим точное положение на грани, в котором происходит переход.

При реализации данного алгоритма для 2D-пространства все что мы будем иметь это индексы точек и индекс куба, с которым мы работаем на данной итерации. Всего есть 16 (2^4) комбинаций состояний точек (углов) куба (см. рис. 2).

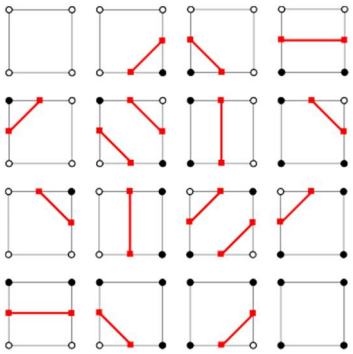

Рис. 2. 16 возможных комбинаций состояний и их решения

В 3D-пространстве ситуация усложняется так как там мы будем работать с кубами, а не квадратами. Так, у нас будет 256 (2^8) уникальных случаев. Но не стоит пугаться так как в Интернете можно легко найти таблицу триангуляции, которая будет описывать все 256 случаев.

Используя такой алгоритм в 3D для сборки трехмерной сферы получим результат изображенный на рисунке 3.

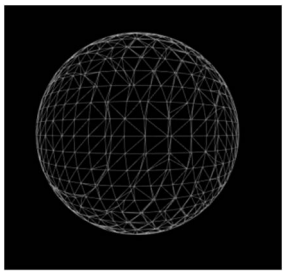

Рис. 3. Результат создания сферы используя алгоритм Marching cubes

Во время выполнения программы на каждой итерации мы будем иметь куб с 8-ю вершинами, их мы будем интерпретировать как биты 8-битного числа. Если вершина имеет состояние "внутри", то биту ставится значение 1, иначе 0. Таким образом мы получаем индекс в таблице триангуляции, она то нам и говорит какие грани нужно использовать.

Из рисунка 3 становится сразу виден первый недостаток метода, топология [4] нашей сферы не очень хорошая. Второй недостаток метода заключается в том, что он не может создавать острые углы. К слову об острых углах, в наших целях они не особо и нужны.

Так как алгоритм обрабатывает каждый куб трехмерной матрицы независимо друг от друга, мы можем распараллелить этот алгоритм используя графический процессор. Благодаря этому мы сможем достичь большого ускорения в скорости работы алгоритма. В таблице 1 показано наше сравнение скорости работы алгоритма. Мы сравнивали работу алгоритма Marching Cubes на 8 потоках центрального процессора (ЦП) с тем же алгоритмом, но реализованным под архитектуру графического процессора (ГП).

Таб. 1. Сравнение скорости работа алгоритма Marching Cubes на разных процессорах

Эксперимент №	Время выполнения алгоритма на ЦП (сек.)	Время выполнения алгоритма на ГП (сек.)
1	6.278	0.568
2	6.083	0.512
3	6.250	0.510
4	6.214	0.526

На данном этапе уже становится понятно, как применять этот метод для создания трехмерных планет. Чтобы все планеты не были одинаковыми, мы будем использовать алгоритмы-шумы, например, один из самых популярных – шум Перлина [5]. Шум Перлина очень хорошо подходит для целей нашей задачи так как он позволяет создавать довольно реалистичные случайные ландшафты. Шум Перлина,

как и любой другой алгоритм шума, можно представить в виде функции, которая принимает 2-3 параметра (например, координаты в 3D-пространстве) и выдает значение шума в этой точке. Алгоритмы-шумы используют псевдослучайные генераторы случайных чисел и часто требует т.н. значение "семя" для начала работы. Значение "семя" и будет определять конфигурацию генератора случайных чисел, который используется алгоритмом-шумом. Таким образом, при одинаковых входных параметрах, алгоритм-шум может выдавать разные значение при разных конфигурациях генератора случайных чисел. Используя алгоритмы-шумы, мы будем как-бы "выдавливать" вершины сгенерированной сферы от ее центра на значение функции-шума. Таким образом, мы будем получать довольно неплохое подобие планеты.

1.2. ПРОЧИЕ АЛГОРИТМЫ СОЗДАНИЯ ПРОЦЕДУРНОЙ 3D-ГЕОМЕТРИИ

Как мы уже не раз говорили, мы выбрали алгоритм Marching cubes потому, что он позволяет использовать многопоточность в большой степени и разделять алгоритм на более мелкие независимые логические элементы. Алгоритмы, которые мы опишем ниже, как и алгоритм Marching Cubes, относятся к классу алгоритмов визуализации изоповерхностей [6], но алгоритмы, описанные ниже не позволяют использовать многопоточность (т.е. алгоритм невозможно разбить на независимые под задачи), так как они ставят качество сгенерированной геометрии на первое место, из-за чего не позволяют в полной мере использовать ресурсы компьютера.

Алгоритм Dual Contouring был впервые представлен на конференции компьютерной графики SIGGRAPH в 2002 году. Алгоритм разработан как расширение алгоритмов Marching Cubes и Surface Nets. Алгоритм Dual Contouring разрешает проблемы Marching Cubes путем требования большей информации о функции, которая определяет геометрию, а именно алгоритм требует знания производной в каждой точке трехмерной матрицы. "Dual Contouring помещает в каждую ячейку по одной вершине, а затем «соединяет точки», создавая полный меш. Точки соединяются вдоль каждого ребра, имеющего смену знака, как и в marching cubes" [5].

Главная проблема Dual Contouring, в отношении наших целей, заключается в том, что алгоритм вводит т.н. inter-cell dependency – работать с ячейками (кубами трехмерной матрицы) по отдельности нельзя, ячейки зависят друг от друга. Такая проблема исключает использование графического процессора для ускорения вычислений, из-за чего алгоритм получается не пригоден для наших целей.

Эта проблема также затрагивает и другие более корректные (в плане топологии) методы, вот некоторые из них [7]:

Extended Marching Cubes – улучшение алгоритма Marching Cubes в пользу качества. Использует расширенную таблицу триангуляции размером не 256 (2^8), а 6561 (3^8), также исключает треугольники с острыми углами ($< 20^o$), а также исключает появление "дыр" в геометрии, что улучшает топологию.

Dual Marching Cubes – улучшение Dual Contouring, исключает появление т.н. manifold геометрии.

Отдельно стоит сказать про Cubical Marching Squares [8][9] – альтернативный подход к построению геометрии, основывается на Marching Squares и Dual Contouring, но исключает inter-cell dependency, а также позволяет получать практически идеальную геометрию с минимумом ошибок. Является самым корректным методом из всех перечисленных. К сожалению, очень сложен в реализации к тому же, авторы алгоритма еще работают над реализацией алгоритма под графические процессоры.

Большая часть этих алгоритмах используется в других задачах: задачах визуализаций изоповерхностей и задачах ремешинга (подробнее [10]), которые не требуют высокой скорости выполнения, а основываются на качестве результата.

2.1. УПРАВЛЕНИЕ МАССИВНОЙ ГЕОМЕТРИЕЙ

Так как в нашей работе мы будем реализовывать функционал для графических интерактивных приложений, в частности для компьютерных игр, то это означает что такие приложения будут требовать высокой производительности и высокого энергопотребления. Практика показывает, что для таких задач единственным правильным вариантом будет выбор языка программирования C++, так как язык C++ обладает высокой производительностью.

В качестве графического API [11] мы выбрали DirectX [12], так как мы будем разрабатывать под платформу Microsoft Windows.

Реализация алгоритма Marching Cubes на графическом процессоре, подразумевает использование compute [13] шейдеров и языка шейдеров HLSL [14].

Как мы уже говорили, алгоритм Marching Cubes можно перенести на графический процессор. Для определения того, какие потоки графического процессора будут работать над какими кубами трехмерной матрицы, в алгоритме Marching Cubes, мы будем создавать группы потоков размерностью 8 x 8 x 8 (это значение должно быть кратно двум), что в произведении дает 512. Попытка увеличить размерность группы еще больше приведет к ошибке компиляции шейдера, так как DirectX не позволяет использовать группы потоков, которые в произведении дают больше 1024 потоков. Определив размер трехмерной матрицы алгоритма Marching Cubes, мы поделим каждое измерение этой трехмерной матрицы на 8, и тем самым определим количество групп потоков, по каждой оси для запуска

вычислений. Таким образом, каждому потоку на графическом процессоре будет выделен ровно один куб в трехмерной матрице алгоритма Marching Cubes.

Перед тем, как рассматривать алгоритм на графическом процессоре, мы, первым делом, рассмотрим алгоритм Marching Cubes разработанный под архитектуру центрального процессора используя язык C++.

В случае реализации на C++ мы создаем буфер вершин и индексов в оперативной памяти компьютера, заполняем его вершинами и индексами по мере работы алгоритма Marching Cubes, а после этого копируем эти буферы в видео память, доступ к которой может получить графический процессор.

В случае реализации используя compute шейдеры и HLSL, мы должны создать вершинный буфер и буфер индексов, затем, мы должны перенести их в видео память, и запустить compute шейдер, который и будет выполнять алгоритм Marching Cubes и заполнять буферы сгенерированными данными.

Так как шейдеры не могут управлять размерами используемых ресурсов, в нашем случае вершинным буфером, это означает, что шейдер не может "расширять" буфер вершин по мере генерации новых вершин. Для решения этой проблемы, мы создаем достаточно большой буфер вершин сначала в оперативной памяти, который будет заполнен нулевыми значениями, затем мы передаем этот буфер в видео память, чтобы compute шейдер использовал это зарезервированное место. Такой подход, очевидно, не является лучшим решением.

Еще одна проблема возникает на этом этапе, так как техники синхронизации потоков в compute шейдерах довольно скудны, мы не можем обеспечить последовательный и синхронизированный доступ к буферу вершин. Для решения этой проблемы, мы предлагаем каждому потоку, который ответственен за обработку отдельного куба в алгоритме Marching Cubes, выделить уникальный индекс на элемент в вершинном буфере. Так, поток, обработав определенный куб, будет использовать этот индекс и записывать данные только в выделенное ему место в буфере вершин.

Для обеспечения требуемых размеров вершинного буфера, для каждого куба в алгоритме Marching Cubes мы выделяем 15 вершин, так как максимальное количество вершин, которые могут быть сгенерированны в результате обработки куба равно 15 (но на самом деле, это количество может принимать одно из следующих значений: 0, 3, 6, 9, 12, 15). Так как до выполнения алгоритма мы не знаем финальное количество вершин, мы предполагаем худший случай и выделяем по 15 вершин на каждый куб в вершинном буфере. После чего мы передаем каждому потоку индекс куба в вершинном буфере, куда он должен записывать данные.

Проблема такой реализации заключается в создании большого количества лишних вершин. Количество не используемых вершин чаще всего больше 50% от всего количества вершин. Например, мы выделили буфер на 7680000 вершин, а на самом деле использовались только 52788 вершин (данные взяты с реального примера), таким образом, мы использовали только ~0.68% (менее одного процента) из всех вершин. Если 7680000 вершин занимают 4 ГБ оперативной памяти, то реально нам нужно ~27 МБ памяти. Это очень большая разница в памяти и таким образом очень серьезная проблема, которую мы собираемся решить в нашей модификации.

2.2. ПРЕДЛАГАЕМЫЙ АЛГОРИТМ РЕШЕНИЯ ПРОБЛЕМ С ПАМЯТЬЮ

Для решения проблемы с памятью, предложенной реализации алгоритма Marching Cubes, мы предлагаем решение, включающее в себя две стадии работы compute шейдера. Сначала, мы запустим половину алгоритма Marching Cubes на compute шейдере, суть данного (первого) этапа в определении финального количества вершин для всей геометрии. На вход данному шейдеру мы не будем подавать зарезервированный буфер, заместо этого, мы подадим начальные данные для работы алгоритма. Выполнив примерно 40% алгоритма мы уже будем знать сколько вершин в конечном итоге будет сгенерированного для конкретного куба в трехмерной матрице алгоритма Marching Cubes. На этом моменте мы заканчиваем работу алгоритма и добавляем количество вершин в конкретном кубе в общую переменную, которая хранит общее количество вершин для всей геометрии.

Функция InterlockedAdd [15] языка HLSL позволяет выполнить атомарную операцию сложения (но только для чисел типа int и unsigned int), используя которую мы добавляем количество вершин в общую переменную и при этом гарантируем, что данная операция не будет пересекаться с такой же операцией из другого потока.

После того, как все потоки закончили работу мы обращаемся к DirectX и просим его скопировать значение переменной, которая хранит общее количество вершин, с видео памяти в оперативную память. На этом первый этап заканчивается.

На втором этапе мы создаем буфер вершин требуемого размера и передаем его в новый compute шейдер, который уже выполняет оставшуюся логику алгоритма Marching Cubes и создает геометрию.

Самой сложной частью данного алгоритма является синхронизация на втором этапе. Если в изначальной реализации каждому потоку отводилось 15 ячеек в вершинном буфере, то в данном случае мы так сделать не можем. На втором этапе каждый поток не будет знать в какую ячейку ему обращаться и сколько значений в вершинном буфере ему выделено.

Мы предлагаем следующее решение. На первом этапе мы будем использовать особую перегрузку функции InterlockedAdd, которая имеет 3 параметра, 3-ий параметр – original value, т.е. значение, которое было до сложения, это значение мы будем использовать. Выделим буфер целых чисел, по одному числу на каждый поток первого этапа и передадим этот буфер в шейдер первого этапа. Каждому потоку будет отведена ячейка с числом в этом буфере, при вызове функции InterlockedAdd поток сохранит в свою ячейку буфера значение original value. После первого этапа мы передадим этот буфер во второй этап.

Для второго этапа будет запущено такое же количество потоков, как и для первого. В начале второго этапа каждый поток, исходя из результата первого этапа будет знать количество вершин, которое он сгенерирует, если это количество равно нулю, то поток завершается преждевременно, иначе продолжает работу. После того, как поток создаст нужное количество вершин, для определения места в вершинном буфере, которое отведено конкретно этому потоку мы используем следующий подход: поток обращается к буферу значений original value по своему индексу и берет это значение. Затем, поток обращается в вершинный буфер по индексу равному original value, которое он заранее получил в начале второго этапа из своей ячейки. Начиная с этой позиции (с индекса original value) потоку уже будет выделено N позиций для N вершин, которые он сгенерирует. Рисунок 4 иллюстрирует этот подход.

Рис. 4. Визуализация буферов

Буфер со значениями original value будет заполнен на первом этапе. Каждый поток на первом этапе будет иметь уникальный индекс, по которому можно обращаться в этот буфер. Содержимое буфера original value будет перемешано (т.е. не упорядочено) и всегда случайно, и хотя первому потоку будет выделен индекс 0, нет никакой гарантии, что он первым дойдет до функции InterlockedAdd.

На втором этапе потоки только читают из буфера original value, и не записывают в него. Как вы можете видеть рисунок 4 иллюстрирует работу трех потоков (красные стрелки), которые на втором этапе обращаются в буфер origin value по заранее выделенному уникальному индексу, получают значение original value и обращаются по этому индексу в вершинный буфер.

Благодаря предложенному алгоритму, мы не только решим проблему с лишними (не используемыми) ячейками вершинного буфера, но и в добавок не сильно увеличим длительность алгоритма, по сравнению с оригинальной реализацией, так как второй этап даже не использует техники синхронизации.

При сравнении скорости изначального алгоритма (реализованного под графический процессор) и предложенного решения скорость будет даже выше, так как, во-первых нам не нужно изначально выделять огромный вершинный буфер (что является не очень быстрой операцией), и во-вторых даже если не учитывать создание огромного вершинного буфера в изначальной реализации, предложенное решение будет выполняться в среднем на 1 секунду дольше.

2.3. ВРЕМЕННЫЕ РАМКИ И ИЗМЕНЕНИЕ ГЕОМЕТРИИ В РЕАЛЬНОМ ВРЕМЕНИ

На этом предложенный алгоритм не заканчивается. В добавок к описанному алгоритму мы учли, что чем больше объем создаваемой геометрии, тем дольше будет процесс создания геометрии и тем самым дольше процесс обновления (изменения) созданной геометрии. Так как мы подразумеваем, что данный алгоритм будет использоваться в компьютерных играх, которые будут иметь функционал изменения сгенерированной геометрии в реальном времени по желанию пользователя, временные требования к такой операции становятся очень высокими. Чтобы игре сохранять минимум 60 кадров в секунду (негласный стандарт игровой индустрии) и изменять геометрию в реальном времени без ущерба количеству показываемых кадров в секунду, процесс обновления (изменения) геометрии должен выполняться быстрее, чем 16.6 миллисекунд. Например, процесс создания небольшой планеты нашим алгоритмом на практически самом дешевом графическом процессоре Intel UHD 600 может занимать 1-2 секунды, а на графическом процессоре средней ценовой категории NVIDIA Geforce GTX 1060 (который был представлен в 2016 году) тот же процесс с теми же параметрами займет около 500-600 миллисекунд.

На данном этапе, если пользователь захочет изменить геометрию, нам нужно будет заново генерировать всю геометрию, т.е. снова провести 2 этапа compute шейдеров, но как это уже было описано выше, данный подход сразу же сильно выходит за поставленные временные рамки, поэтому мы рассмотрим способы ускорения изменения геометрии.

Мы предлагаем разделить изначальную трехмерную матрицу (в которой и будет построена геометрия) в алгоритме Marching Cubes на под матрицы. Для каждой под матрицы мы проведем по 2

этапа compute шейдеров, которые мы уже описали выше. Таким образом, мы будем иметь по одному вершинному буферу на под матрицу. Преимущество такого метода в том, что при запросе на изменение геометрии, нам не нужно будет перестраивать всю геометрию, все что нам нужно будет – это определить, геометрию какой под матрицы хотят изменить и перестроить только эту под матрицу.

Мы не можем бездумно увеличивать количество под матриц, так как из-за этого, мы потратим больше времени на определение того, геометрию какой под матрицы пытаются изменить. Количество под матриц должно определяться сложностью операции по определению под матрицы, которую нужно изменить. Так, если операция по определению того, какую под матрицу пользователь хочет изменить является сложной операцией и может занять значительное время, то количество под матриц должно быть меньше. Если операция по определению того, какую под матрицу пользователь хочет изменить является простой операцией и выполняется быстро, то количество под матриц можно увеличить.

В общем случае количество под матриц нужно определять экспериментальным путем в зависимости от имеющихся инструментов определения под матриц.

ЗАКЛЮЧЕНИЕ

Рассмотренное в этой работе улучшение оригинального метода Marching Cubes позволяет использовать все ресурсы компьютера (оба процессора) и пользуется возможностью ускорения на графическом процессоре. Предложенный алгоритм также разрешает проблемы с памятью, которые могут возникнуть при создании массивной геометрии при этом не сильно увеличивает длительность работы алгоритма в общем.

В добавок, мы также предложили использовать разделение на под матрицы как еще одна мера ускорения вычислений и изменений геометрии в реальном времени.

Предложенный алгоритм найдет хорошее применение в компьютерных играх космической тематики, которые используют процедурную генерацию для создания бесконечных и уникальных вселенных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. William E. Lorensen, Harvey E. Cline, “Marching Cubes: A High Resolution 3d Surface Construction Algorithm“, ACM SIGGRAPH Computer Graphics, vol. 21, issue 4.

2. Алгоритм Нелдера-Мида [Электронный ресурс] // Страница веб-сайта Wikipedia, описывающая алгоритм Нелдера-Мида. – 2021. – Режим доступа: https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method.

3. Создание разрушаемых мешей [Электронный ресурс] // Статья веб-сайта Habr, в которой описывается алгоритм Marching Cubes. – 2021. – Режим доступа: https://habr.com/ru/post/358658/.

4. Топология, сетка [Электронный ресурс] // Веб-сайт объясняющий значение терминов топология и сетка. – 2021. – Режим доступа: https://3dyuriki.com/2015/03/07/topologiya-retopologiya-mesh-setka-3d-slovar-spravochnik/.

5. Ken Perlin, “An image synthesizer“, ACM SIGGRAPH Computer Graphics, vol. 19. issue 3.

6. Изоповерхность [Электронный ресурс] // Страница веб-сайта Wikipedia, описывающая определение изоповерхности. – 2021. – Режим доступа: https://en.wikipedia.org/wiki/Isosurface.

7. Natalya Tatarchuk, Jeremy Shopf, Christopher DeCoro, “Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline“, ACM SIGGRAPH Computer Graphics, vol. 9, pp. 122-137.

8. Thesis: Cubical Marching Squares Implementation [Электронный ресурс] // Страница описывающая поверхностный пример реализации алгоритма Cubical Marching Squares. – 2021. – Режим доступа: https://grassovsky.wordpress.com/2014/09/09/cubical-marching-squares-implementation/.

9. Chien-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, Ming Ouhyoung, “Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data“, *Eurographics*, vol. 24, issue 3.

10. Yuanchen Zhu, “Uniform Remeshing with an Adaptive Domain: A New Scheme for View-Dependent Level-of-Detail Rendering of Meshes“, IEEE Transactions on Visualization and Computer Graphics, vol. 11, issue 3.

11. Графический API [Электронный ресурс] // Страница веб-сайта Wikipedia, описывающая определение термина API. – 2021. – Режим доступа: https://ru.wikipedia.org/wiki/API.

12. DirectX [Электронный ресурс] // Страница веб-сайта Wikipedia, описывающая графический API DirectX. – 2021. – Режим доступа: https://en.wikipedia.org/wiki/DirectX.

13. Compute shader overview [Электронный ресурс] // Страница из документации DirectX, описывающая compute шейдеры. – 2021. – Режим доступа: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader.

14. HLSL [Электронный ресурс] // Страница веб-сайта Wikipedia, описывающая язык HLSL. – 2021. – Режим доступа: https://en.wikipedia.org/wiki/High-Level_Shading_Language.

15. Interlocked Add [Электронный ресурс] // Страница документации DirectX, описывающая функцию Interlocked Add. – 2021. – Режим доступа: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/interlockedadd.

Третьяков Александр Андреевич – магистр 1-го курса, Пермский национальный исследовательский политехнический университет, тел. 9194616855, e-mail: flonednb@gmail.com.

PROCEDURAL GENERATION OF MASSIVE 3D GEOMETRY USING IMPROVED MARCHING CUBES ALGORITHM

A. A. Tretyakov
Perm National Research Polytechnic University, Perm

Abstract – Procedural generation, or the creation of content while a program is running, is a complex area that requires not only an understanding of 3D graphics, but also graphics programming skills, which often boils down to learning how GPUs work. Because of this complexity, developers often use off-the-shelf content creation tools. Such tools generalize and simplify work by providing a large pre-built set of functions that can be used without knowing programming at all. Unfortunately, generalization often reduces flexibility and introduces new constraints. Statistics show that using procedural generation to create massive 3D geometry is impossible when using ready-made tools with already prepared functions. Such tools do not allow the huge scales of massive geometry to be brought to life due to various constraints. In addition, existing 3D geometry creation algorithms often do not account for the application of these algorithms to create massive 3D geometry such as planets. The Marching Cubes algorithm considered in this work also does not take into account the use of the algorithm for creating massive geometry, which is why the use of this algorithm for such purposes will have many limitations and many disadvantages. But this algorithm was not chosen by chance, it is very popular and we will talk why. This work focuses on modifying the existing Marching Cubes algorithm to apply it to massive geometry. This algorithm will find application in computer games with a space theme, our algorithm allows to create massive 3D geometry of planetary scales even on a low-end computers without special resource costs. In addition, our algorithm allows to change the generated geometry in real time, without time delays, which is so important for computer games.

Index terms: procedural generation, acceleration, gpu, marching cubes.

REFERENCES

1. William E. Lorensen, Harvey E. Cline, “Marching Cubes: A High Resolution 3d Surface Construction Algorithm“, ACM SIGGRAPH Computer Graphics, vol. 21, issue 4.
2. Nelder-Mead algorithm, 2021, https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method.
3. Creating destructible meshes, 2021, https://habr.com/ru/post/358658/.
4. Topology, mesh, 2021, https://3dyuriki.com/2015/03/07/topologiya-retopologiya-mesh-setka-3d-slovar-spravochnik/.
5. Ken Perlin, “An image synthesizer“, ACM SIGGRAPH Computer Graphics, vol. 19. issue 3.
6. Isosurface, 2021, https://en.wikipedia.org/wiki/Isosurface.
7. Natalya Tatarchuk, Jeremy Shopf, Christopher DeCoro, “Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline“, ACM SIGGRAPH Computer Graphics, vol. 9, pp. 122-137.
8. Thesis: Cubical Marching Squares Implementation, 2021, https://grassovsky.wordpress.com/2014/09/09/cubical-marching-squares-implementation/.
9. Chien-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, Ming Ouhyoung, “Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data,“ *Eurographics*, vol. 24, Issue 3.
10. Yuanchen Zhu, “Uniform Remeshing with an Adaptive Domain: A New Scheme for View-Dependent Level-of-Detail Rendering of Meshes“, IEEE Transactions on Visualization and Computer Graphics, vol. 11, issue 3.
11. API, 2021, https://ru.wikipedia.org/wiki/API.
12. DirectX, 2021, https://en.wikipedia.org/wiki/DirectX.
13. Compute shader overview, 2021, https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader.
14. HLSL, 2021, https://en.wikipedia.org/wiki/High-Level_Shading_Language.
15. Interlocked Add, 2021, https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/interlockedadd.

Tretyakov Aleksandr Andreevich – 1st year master, Perm National Research Polytechnic University, 9194616855, e-mail: flonednb@gmail.com