Математические структуры и моделирование
2009, вып. 20, с. 121–133

УДК 004.925

ВИЗУАЛИЗАЦИЯ СКАЛЯРНОГО ПОЛЯ НА ОСНОВЕ ДИНАМИЧЕСКОГО ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТИ

А.Ю. Суравикин, В.В. Коробицын

Предложена реализация метода построения изоповерхностей скалярного поля на графическом процессоре. Для построения поверхности применен метод граничных тетраэдров. Описан способ кодирования информации о тетраэдрах в текстурах, обрабатываемых на графическом устройстве. Представлены листинги вершинного и геометрического шейдеров. Приведены результаты тестирования производительности реализации алгоритма.

Введение

При решении задач моделирования различных физических процессов часто приходится оперировать трехмерными массивами данных. Например, при моделировании частиц газов или жидкостей можно представить некоторый ограниченный объем как резервуар, в котором для каждой точки определено значение давления газа [1]. Приведенный в примере объем можно рассматривать как скалярное поле, в каждой точке которого определяется количество частиц. В ходе моделирования газ или жидкость будет перемещаться внутри резервуара, и, следовательно, количество частиц в определенной области будет изменяться. Для подобных экспериментов имеет смысл визуализировать количество жидкости в единице объема.

Существует несколько способов визуализации жидкости. Один из них основан на прямой визуализации трехмерного поля и использует технологии, подобные Volume Ray Marching (Casting), т.е. происходит трассировка лучей сквозь резервуар (объем) и при формировании изображения рассчитывается столкновение луча с объектом или попаданием луча в некую окрестность частицы. От плотности может зависеть цвет или уровень прозрачности жидкости в данной точке. Подробнее о Ray Marching можно узнать в [2], [3] и [4]. В данной статье представлен другой способ: он основан на создании поверхности, которая ограничивает жидкость с заранее заданной плотностью. Этот способ отличается

Омский государственный университет.
E-mail: tizona_del_cid@rambler.ru

тем, что вместо отображения собственно поля с помощью попиксельных алгоритмов используется создание поверхности из примитивов. Следовательно, изменяется область применения: вместо визуализации внутренней структуры жидкости изображается граница раздела сред.

Поскольку алгоритмы визуализации жидкостей и газов достаточно ресурсоёмкие, то целесообразно реализовывать эти алгоритмы на скоростных графических процессорах (GPU). Для реализации алгоритмов на GPU обычно используются шейдерные языки, обеспечивающие функционалом программирования графических процессоров.

Цель работы состоит в реализации алгоритма граничных тетраэдров на графическом процессоре. Для достижения поставленной цели необходимо решить следующие задачи:

1. Изучить литературные источники по теоретическим основам алгоритма.
2. Создать конвертор результатов моделирования жидкости в скалярное поле.
3. Визуализировать скалярное поле с помощью геометрических шейдеров.

Дополнительной задачей является выявление сильных и слабых сторон реализации алгоритма на геометрических шейдерах с точки зрения производительности.

Работа включает в себя как реализацию алгоритма Marching Tetrahedra (граничные тетраэдры), так и создание приложения-основы (так называемого framework) для рендера поверхности и моделирования системы частиц и трехмерного скалярного поля. Подробно описан алгоритм и способ его реализации с использованием вершинной и геометрической стадий, оценена скорость визуализации. Актуальность работы заключается в разработке новой программы с помощью уже известного способа визуализации, который не был до этого достаточно детально описан в литературе.

1. Алгоритм граничных тетраэдров (Marching Tetrahedra Algorithm)

1.1. Изоповерхность

Извлечение изоповерхности — это способ создания полигонального представления непрерывного поля, например таких, которые создаются математическим описанием неявных поверхностей или прямыми измерениями, например медицинским сканированием. Изоповерхность изображает поверхность в объеме, для которого все точки имеют одинаковое значение поля, назовем его изозначением. Для различных изозначений создаваемая поверхность будет также различной (подробно описано в [4] и [5]).

Алгоритм создает изоповерхность, заполняя пространство поля плотно расположенными тетраэдрами. Для каждой вершины тетраэдра значение поля в этой точке сравнивается с изозначением, чтобы получить булево значение. Если

это значение разное для вершин одного тетраэдра, то изоповерхность проходит через объем, ограниченный этим тетраэдром. Если предположить, что поверхность плоская внутри тетраэдра, то сечением тетраэдра этой плоскостью будет либо треугольник, либо четырехугольник.

1.2. Линии с информацией о смежности

Линии со смежностью — новый тип примитивов, который состоит из отрезка и двух дополнительных вершин. Таким образом, в геометрическом шейдере обрабатывается одна такая линия, состоящая из четырех трансформированных вершин.

Отрезок рисуется от вершины $(4i+2)$ к $(4i+3)$ для каждого значения $i = 0, 1, ...n-1$ при общем количестве линий n. Для i-го отрезка вершины $(4i+1)$ и $(4i+4)$ будут смежными соответственно к вершинам $(4i+2)$ и $(4i+3)$.

На рисунке 1 схематично изображен порядок формирования таких примитивов. Подробнее о линиях со смежностью и других новых видах примитивов можно узнать в [9].

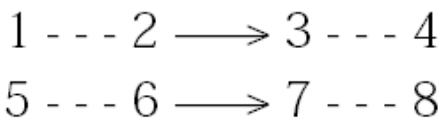

Рис. 1. Линии с дополнительными вершинами. Вершины основных примитивов соединены сплошной линией, прерывистой линией соединены соседние вершины, которые могут использоваться в геометрическом шейдере

1.3. Реализация на графическом процессоре

Извлечение изоповерхности реализуется на графическом процессоре с помощью совместного использования вершинного и геометрического шейдеров (общую информацию о шейдерах можно найти в [10]). Основой будет точная тетраэдризация объема, где четыре вершины каждого тетраэдра передаются как линия с двумя смежными вершинами. Расскажем об этих объектах подробнее.

Вершинный шейдер считывает значения поля в вершинах тетраэдра, сравнивает его с изозначением, а также трансформирует вершину в экранные координаты. Геометрический шейдер обрабатывает результаты работы вершинного шейдера со всех четырех вершин тетраэдра. Он производит операцию логического ИЛИ результатов сравнений с изозначением и формирует четырехбитовое целое значение. Оно определяет геометрию, которую произведет геометрический шейдер. Если все биты установлены в 1 (получим значение 15) или в 0 (получим значение 0), то тетраэдр не пересекает изоповерхность и геометрия не создается. В противном случае, четырехбитное значение используется как координата текстуры, которая содержит таблицу значений, описывающих какую геометрию создаст шейдер.

Таблица 1. Таблица соответствия индекса и номеров ребер тетраэдра

индекс	ребра 0 и 1	ребра 2 и 3
0	0, 0, 0, 0	0, 0, 0, 1
1	3, 0, 3, 1	3, 2, 0, 0
2	2, 1, 2, 0	2, 3, 0, 0
3	2, 0, 3, 0	2, 1, 3, 1
4	1, 2, 1, 3	1, 0, 0, 0
5	1, 0, 1, 2	3, 0, 3, 2
6	1, 0, 2, 0	1, 3, 2, 3
7	3, 0, 1, 0	2, 0, 0, 0
8	0, 2, 0, 1	0, 3, 0, 0
9	0, 1, 3, 1	0, 2, 3, 2
10	0, 1, 0, 3	2, 1, 2, 3
11	3, 1, 2, 1	0, 1, 0, 0
12	0, 2, 1, 2	0, 3, 1, 3
13	1, 2, 3, 2	0, 2, 0, 0
14	0, 3, 2, 3	1, 3, 0, 0

Из таблицы 1 мы получаем восемь значений, которые описывают номера вершин четырех ребер, пересекающих изоповерхность. Значения для четвертого ребра закодированы специальным образом: первая вершина (седьмая колонка значений в таблице) всегда будет ненулевой, если пересечение является четырехугольником. Мы используем эту особенность как условие, генерировать четвертую вершину (а значит, и второй треугольник) или нет. Для создания выходного примитива шейдер решает линейное уравнение на трех или четырех ребрах. Таким образом, происходит интерполяция позиции вершины к точке, где поле пересекает ребро. Это выходные значения для сгенерированных примитивов.

1.4. Настройка по адресам (Swizzling)

В дополнение к реализации прямого извлечения изоповерхности в данном проекте используется техника для оптимизации проходов объема. Вместо того, чтобы производить рендер сетки тетраэдров в построчном порядке, мы изменяем их порядок для улучшения локальности обращений. Техника основана на изменении порядка битов: при проходе объем делится на множество из 8 маленьких объемов (расположение 2 x 2 x 2), каждый из которых содержит шесть тетраэдров. Далее производится проход по этим блокам в обычном построчном порядке. Возможны также другие способы проходов, но используемый метод представляет собой компромисс между сложностью и производительностью.

2. Описание алгоритма

Алгоритм можно разделить на две основные части — инициализацию и рендер, т.е. расчет и вывод изображения. Рассмотрим эти этапы более детально.

2.1. Инициализация

Прежде чем строить изоповерхность, необходимо разбить пространство на ячейки, в узлах которых будет считываться значение поля. Для упрощения расчетов на графическом процессоре ограничим количество узлов числами, являющимися степенями 2. Собственно инициализация состоит из следующих этапов:

1. Создание трехмерной решетки.
2. Загрузка программы для графического процессора.
3. Создание таблицы границ для ячейки.

Программы загружаются с использованием функций графического API, при загрузке передаем параметры, не меняющиеся во время выполнения шейдерных программ. Решетка создается из вершин, расположенных с одинаковыми интервалами по кубу со значениями [-1; 1) по всем измерениям. Индексы создаются для ячеек, в узлах которых расположены созданные вершины. Каждая ячейка состоит из 6 тетраэдров, в каждом тетраэдре 4 индекса. Индекс формируется побитово из целочисленных значений положения ячейки в решетке:

```
index=(int)((x)|((y)<<sizeLog2[0])|((z)<<(sizeLog2[0]+sizeLog2[1])))
```

Здесь x, y, z - положение ячейки, $sizeLog2$ - логарифм по основанию 2 числа ячеек в стороне решетки.

2.2. Рендер

Реализация рендера жидкости описана далее поэтапно в соответствии с порядком обработки данных на графическом конвейере. Описаны вершинный и геометрический шейдеры, этап растеризации и пиксельный шейдер тривиальны.

2.2.1. Вершинный шейдер

Графический процессор рендерит вершинный буфер, загруженный на этапе инициализации и обрабатывает вершины этого буфера-сетки. Вершинная программа приведена в листинге 1. В коде приведены комментарии, объясняющие последовательность действий.

Листинг 1. Программа обработки вершин

```
// Vertex shader
// march_tetra.vert
uniform mat4 WorldViewProj; // Матрица трансформации
uniform sampler3D FieldTex; // Сэмплер текстуры-источника данных

// Маска и смещение битов для сетки
uniform ivec3 SizeMask;
uniform ivec3 SizeShift;
```

```
uniform vec3 scalePos; // Фактор масштабирования
uniform float IsoValue; // Изозначение

void main() //Главная функция
{
    int index = gl_VertexID;//Получаем индекс текущей вершины

    // Получаем позицию считывания данных текстуры из индекса
    vec3 Pos;
    Pos.x = float((index >> SizeShift.x)&SizeMask.x)/(SizeMask.x+1);
    Pos.y = float((index >> SizeShift.y)&SizeMask.y)/(SizeMask.y+1);
    Pos.z = float((index >> SizeShift.z)&SizeMask.z)/(SizeMask.z+1);
    // Считываем значение тесктуры в рассчитанной позиции
    vec4 Field = tex3D(FieldTex, Pos);
    Pos = Pos * 2.0 - 1.0;
    // Масштабируем позицию
    Pos = Pos * scalePos;

    // Трансформируем позицию
    gl_Position = mul(WorldViewProj, vec4(Pos, 1.0));
    // Передаем на следующую стадию значение поля и
    // булево значение сравнения
    gl_TexCoord[0].x =  Field.x;
    gl_TexCoord[0].y = (Field.x > IsoValue) ? 1.0 : 0.0;
}
```

При обработке массива вершин мы используем следующие входные данные:

1. Индекс вершины, т.е. ее номер ($index$). По этому номеру рассчитывается положение вершины в сетке (ячейках скалярного поля) и приводится к интервалу $[0; 1)$ для считывания из текстуры.

2. 3D-текстура $FieldTex$, содержащая значения скалярного поля.

3. Вспомогательные константы для пространственных преобразований: $SizeMask$, $SizeShift$, $scalePos$, а также матрица $WorldViewProj$ (для преобразования позиции вершины в экранные координаты).

4. Изозначение ($IsoValue$), по которому будет строиться поверхность.

Вершинный шейдер возвращает трансформированную вершину ($gl\_Position$), а также значение поля в этой точке ($gl\_TexCoord[0].x$) и булево значение сравнения с IsoValue ($gl\_TexCoord[0].y$). При этом для изменения размеров сетки «резервуара» используется покомпонентное умножение вектора позиции на вектор $scalePos$ (в нашем примере $\{30.0; 30.0; 30.0\}$).

В случае размеров сетки 64x64x64 (в данной реализации размер сетки должен быть степенью 2) значение $SizeMask$ устанавливается в $\{63; 63; 63\}$ ($2^n - 1$,

где степень $n = 6$). $SizeShift$ равен n, в данном случае 6. Переменную $IsoValue$ (изозначение) установим в 0.15 (при этом «вес» каждой частицы равен 0.1). $WorldViewProj$ — матрица преобразования позиции вершины в экранные координаты.

2.2.2. Геометрический шейдер

Обработанные вершины группируются в примитивы, состоящие из четырех вершин (см. выше). Вершины по расположению образуют тетраэдр. Программа обработки представлена в листинге 2.

Листинг 2. Программа обработки примитивов

```
// Geometry shader
// march_tetra.geom
// входные данные: Линии с соседями (тетраэдры)
// выходные данные: 0, 1 или 2 треугольника в зависимости
//                  от сечения изоповерхностью тетраэдра

uniform sampler2DRect edgeTex; //текстура-таблица ребер
uniform float IsoValue; //изозначение

// Функция расчета точек пересечения изоповерхности и заданного ребра
// Используется линейная интерполяция (функция lerp()) между вершинами
// ребра по изозначению и значениям поля в точках, соответствующих
// этим вершинам
vec4 CalcIntersection(vec4 Pos0, vec2 Field0, vec4 Pos1, vec2 Field1)
{
  float t = (IsoValue - Field0.x) / (Field1.x - Field0.x);
  return lerp(Pos0, Pos1, t);
}

// Расчет нормали к плоскости, заданной тремя точками
// Используется векторное произведение двух векторов,
// соединяющих заданные вершины
vec3 calcNormal(vec3 v0, vec3 v1, vec3 v2)
{
  vec3 edge0 = v1 - v0;
  vec3 edge1 = v2 - v0;
  return normalize(cross(edge0, edge1));
}

void main()
{
  // Из полученных булевых значений собираем индекс тетраэдра
  int index = (int(gl_TexCoordIn[0][0].y) << 3) |
              (int(gl_TexCoordIn[1][0].y) << 2) |
```

```
            (int(gl_TexCoordIn[2][0].y) << 1) |
             int(gl_TexCoordIn[3][0].y);

// Если тетраэдр полностью лежит снаружи (index = 0) или внутри
// (index = 15) поверхности, то для него не создаем треугольники
if (index > 0 && index < 15)
{
  //Получаем значения из таблицы ребер для текущего значения index
  ivec4 e0 = ivec4(texelFetch2DRect(edgeTex, ivec2(index,0))*255);
  ivec4 e1 = ivec4(texelFetch2DRect(edgeTex, ivec2(index,1))*255);

  // Создаем треугольник из точек пересечения изоповерхности
  // и текущего тетраэдра
  vec4 p0 = CalcIntersection(
           gl_PositionIn[e0.x], gl_TexCoordIn[e0.x][0].xy,
           gl_PositionIn[e0.y], gl_TexCoordIn[e0.y][0].xy);
  vec4 p1 = CalcIntersection(
           gl_PositionIn[e0.z], gl_TexCoordIn[e0.z][0].xy,
           gl_PositionIn[e0.w], gl_TexCoordIn[e0.w][0].xy);
  vec4 p2 = CalcIntersection(
           gl_PositionIn[e1.x], gl_TexCoordIn[e1.x][0].xy,
           gl_PositionIn[e1.y], gl_TexCoordIn[e1.y][0].xy);

  // Расчет нормали для плоскости созданного треугольника
  vec3 n = calcNormal(p2.xyz, p1.xyz, p0.xyz);
  // Задаем параметры первой вершины и создаем ее
  gl_Position = p0;
  gl_TexCoord[0] = vec4(n, 0.0);
  EmitVertex();
  // Задаем параметры второй вершины и создаем ее
  gl_Position = p1;
  gl_TexCoord[0] = vec4(n, 0.0);
  EmitVertex();
  // Задаем параметры третьей вершины и создаем ее
  gl_Position = p2;
  gl_TexCoord[0] = vec4(n, 0.0);
  EmitVertex();
  // Аналогично создаем второй треугольник,
  // если сечение - четырехугольник
  if (e1.z != 0)
  {
       vec4 p3 = CalcIntersection(
                 gl_PositionIn[e1.z], gl_TexCoordIn[e1.z][0].xy,
                 gl_PositionIn[e1.w], gl_TexCoordIn[e1.w][0].xy);
       n  = calcNormal(p1.xyz, p2.xyz, p3.xyz);
```

```
        gl_Position = p3;
        gl_TexCoord[0] = vec4(n, 0.0);
        EmitVertex();
    }
  }
}
```

В коде геометрического шейдера показано, как происходит создание примитивов, представляющих собой сечения поверхности в каждом тетраэдре. Пример такого сечения показан на рис. 2. Для наглядности пример представлен в двумерном виде.

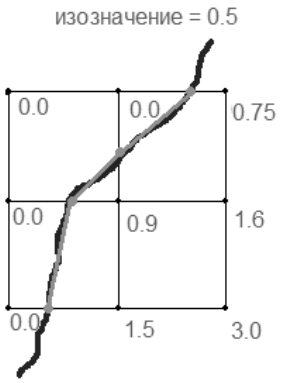

Рис. 2. Сечение поверхностью тетраэдров

Расскажем подробнее о работе геометрического шейдера. Для каждой группы вершин выполняется функция $main()$. При этом размер группы определяется типом входящего примитива. В нашем случае берется 4 вершины, т.к. используемый тип — линии со смежностью. Сначала с помощью побитовых операций формируется индекс ($index$) из булевых значений вершин, затем происходит сравнение индекса с граничными значениями (0 и 15): если индекс равен одному из них, то пропускаем обработку этого примитива, т.к. значение 0 означает, что тетраэдр полностью лежит снаружи объема (ограниченного изоповерхностью), а 15 (1111_2) - что тетраэдр лежит внутри него. Иначе создаем примитивы следующим образом:

1. Получаем массивы номеров ребер $e0$ и $e1$ из таблицы ребер (таблица хранится в отдельной текстуре, хотя можно ее задать через Constant Buffer).

2. Рассчитываем позиции вершин для создаваемого треугольника с помощью линейной интерполяции между вершинами тетраэдров по изозначению поля. Для каждой вершины используется функция $CalcIntersection$, в которой по значению поля в вершинах ребра рассчитывается коэффициент интерполяции, а затем производится собственно интерполяция координат вершин по этому коэффициенту.

3. Расчет нормали к полученному треугольнику. Нормали к плоскости создаваемого треугольника используются для расчета его освещения. В приведенном листинге расчет производится на этапе создания треугольника по трем вершинам (функция *calcNormal*). Однако есть другой способ: использовать скалярное поле, а точнее, градиенты скалярного поля в точках, соответствующих полученным вершинам треугольника. Расчет нормалей по градиентам производится с помощью функции, описанной в Листинге 3.

Листинг 3. Функция расчета нормали как градиента поля

```
// Расчет нормали как градиента поля в точке:
// s - семплер (sampler) текстуры поля,
// t - текстурная координата,
// invFieldTexSize - вектор обратных значений размеров 3D-текстуры поля
vec3 calcGradientNormal(sampler3D s, vec3 t)
{
vec4 step = vec4(invFieldTexSize, 0.0);
vec3 gradient = vec3(
texture3D(s, t + step.xww).x - texture3D(s, t - step.xww).x,
texture3D(s, t + step.wyw).x - texture3D(s, t - step.wyw).x,
texture3D(s, t + step.wwz).x - texture3D(s, t - step.wwz).x);
return normalize(-gradient);
}
```

Используется формула расчета градиента, закодированная таким образом, чтобы достигать высокой производительности на системах с параллельной архитектурой. Недостатком является 6 чтений из текстуры (т.к. это операции с большой латентностью), однако они производятся из соседних ячеек, поэтому будет эффективно использоваться текстурный кэш, который уменьшит задержки на доступ к текстуре.

Вызов данной функции осуществляется для каждой вершины следующим образом:

```
n = calcGradientNormal(FieldTex, lerp(gl_TexCoordIn[e0.x][1].xyz,
   gl_TexCoordIn[e0.y][1].xyz, t));
```

здесь $FieldTex$ - семплер текстуры поля, т.е. переменная, по которой определяется какая текстура и каким образом будет считываться. Вторым параметром идет результат функции *lerp*, т.е. линейной интерполяции значений текстурных координат поля. Остановимся на этом параметре. Это координаты точки в поле, приведенные к интервалу $[0; 1)$, используемые для адресации текстуры. С помощью временной переменной *step* мы можем обращаться к соседним ячейкам текстуры, таким образом рассчитывая градиент поля. Параметры функции интерполяции — значения текстурных координат соответствующих вершин ($gl\_TexCoordIn$[индекс ребра][1] — текстурная координата поля передается из вершинного шейдера) и коэффициент интерполяции t, рассчитанный в функции $CalcIntersection$. Коэффициент интерполяции для позиции вершины и ее

текстурных координат будет одинаковым, поэтому имеет смысл рассчитать его один раз и использовать как при расчете позиции, так и при расчете нормали.

В переменные $gl\_Position$ и $gl\_TexCoord[0]$ заносятся соответственно позиция и нормаль (передается через текстурную координату) для создаваемой вершины, затем происходит вызов функции $EmitVertex$, который формирует вершину. После создания трех вершин, которые сформируют новый треугольник, идет проверка индекса $e1.z$, т.е. первой вершины четвертого ребра тетраэдра. Если значение не равно нулю, то происходит создание четвертой вершины, которая вместе с двумя предыдущими сформирует еще один треугольник. В результате получим один или два треугольника, которые будут аппроксимировать пересечение изоповерхности текущего тетраэдра. Подобная операция производится для всех тетраэдров, таким образом, получим всю изоповерхность.

3. Результаты моделирования

Изображение изоповерхности, построенное в предложенной реализации, приведено на рис. 3. Однако при выводе на экран не производится сглаживание построенной поверхности, поэтому изображение выглядит угловатым. Процедура сглаживания изображения поверхности будет разработана позже.

Рис. 3. Рендер изоповерхности

Необходимо отметить, что этот рисунок является одним кадром из генерируемой анимации процесса движения струи жидкости. Вывод анимации на экран производится в реальном времени, и ее скорость зависит от параметризации модели и производительности графической системы. Для проверки производительности алгоритма мы протестировали его на различных по размеру

массивах данных. В таблице 2 представлены данные производительности, измеренные в кадрах в секунду (Frames Per Second, FPS) для массивов с 32x32x32 по 128x128x128 на двух системах. Так как в реализации понятия размера массива и размера сетки (из которой формируется поверхность) разделены, то можно отдельно регулировать качество визуализации (число треугольников, аппроксимирующих поверхность, напрямую влияет на детализацию) и точность расчетов (чем больше размер текстуры данных, тем выше точность). Это актуально, если дополнительно использовать скалярное поле для собственно моделирования жидкости (расчетов плотности, давления и т.п.). В данной реализации используется упрощенная система моделирования жидкости, основанная на системе частиц, поэтому имеет смысл устанавливать только одинаковые размеры сетки и текстуры.

Производительность реализации алгоритма оценивалась на двух вычислительных системах: система 1 — AMD Phenom X4 9850+, 4GB RAM, NVIDIA GeForce 260 GTX; система 2 — Intel Core 2 Duo 2.26, 4GB RAM, NVIDIA GeForce 9800M GTS.

Таблица 2. Таблица производительности реализации алгоритма на различных размерах массивов данных

Размер поля	Производительность (FPS)	
	система 1	система 2
32x32x32	157,0	112,5
64x64x64	81,5	64,0
128x128x128	17,1	15,5

Из таблицы 2 также видно, что, несмотря на серьезное различие в теоретической производительности видеоподсистем, различия в реальной производительности не настолько значительны и уменьшаются при увеличении размера текстуры. Это сигнализирует о том, что при увеличении нагрузки мы сталкиваемся с т.н. «узким местом» (bottleneck) системы. Способ избавиться от этого еще предстоит выяснить.

Заключение

В работе реализован алгоритм, позволяющий интерактивно отображать результаты моделирования жидкостей, а точнее, границы раздела сред. Визуализация скалярного поля основана на алгоритме граничных тетраэдров. Данный алгоритм был реализован с использованием библиотеки OpenGL. При этом на визуализацию поля разрешением 128x128x128 видеокарте NVIDIA GeForce 9800M GTS требуется около 50–70 мс, что позволяет использовать этот метод в интерактивных приложениях, т.е. при изменении данных скалярного поля можно строить полигональную поверхность несколько раз в секунду.

Возможное улучшение алгоритма состоит в том, чтобы сделать сетку тетраэдров нерегулярной, например, использовать такие структуры как oct-tree, k-d tree и другие. Это позволит увеличивать плотность ячеек с тетраэдрами

в тех местах, где больше плотность частиц (или градиент плотности частиц) поля, и уменьшать там, где частиц нет. Следовательно, работа алгоритма станет более эффективной, т.е. будет улучшено качество получаемой поверхности и уменьшено число требуемых для этого ячеек.

Литература

1. Müller, M. Particle-based fluid simulation for interactive applications / M. Müller, D. Charypar, M. Gross // SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation. – 2003. – P. 154–159.
2. GPU gems 3 / eds. H. Nguyen. – Boston: Pearson Education, 2008. – Доступно также: http://developer.nvidia.com/object/gpu-gems-3.html (5.10.2009).
3. Pawasauskas, J. Volume Visualization With Ray Casting [Электронный ресурс] / J. Pawasauskas // CS563 - Advanced Topics in Computer Graphics (1997). – Режим доступа: http://web.cs.wpi.edu/ matt/courses/cs563/talks/powwie/p1/ray-cast.htm (5.10.2009).
4. NVIDIA OpenGL SDK Guide [Электронный ресурс]. – Режим доступа: http://developer.download.nvidia.com/SDK/10.5/opengl/OpenGL_SDK_Guide.pdf (5.10.2009).
5. Marching Tetrahedra [Электронный ресурс]. – Data Analysis and Assessment Center. – Режим доступа: https://visualization.hpc.mil/wiki/Marching_Tetrahedra (5.10.2009).
6. Bourke, P. Polygonising a Scalar Field [Электронный ресурс] / P. Bourke. – Режим доступа: http://local.wasp.uwa.edu.au/ pbourke/geometry/polygonise/ (5.10.2009).
7. Lorensen, W. Marching cubes: a high resolution 3D surface construction algorithm / W. Lorensen, H. Cline // Computer Graphics. – 1987. – V. 21, N. 4. – P. 163-169.
8. Newman, T.S. A survey of the marching cubes algorithm / T.S. Newman, H. Yi // Computers & Graphics. – 2006. – V. 30, N. 5. – P. 854-879.
9. Brown, P. ARB_geometry_shader4 [Электронный ресурс] / P. Brown, B. Lichtenbelt // OpenGL - The Industry Standard for High Perfomance Graphics. – Режим доступа: http://www.opengl.org/registry/specs/ARB/geometry_shader4.txt (5.10.2009).
10. Определение понятия «Шейдер» [Электронный ресурс]. – Википедия. – Режим доступа: http://ru.wikipedia.org/wiki/Шейдер (5.10.2009).