ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК
КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

ПОСТРОЕНИЕ ИЗОПОВЕРХНОСТЕЙ ПОКАЗАТЕЛЕЙ ФУНКЦИИ ПО 3D ДАННЫМ ДЛЯ GIS СИСТЕМ

Выпускная квалификационная работа
обучающегося по направлению подготовки 02.03.01
Математика и компьютерные науки
очной формы обучения, группы 07001303
Гиричева Арсения Андреевича

Научный руководитель
к.т.н., доцент
Васильев П. В.

БЕЛГОРОД 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
1. АНАЛИЗ ПРОБЛЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ ПРИМЕНЕНИЯ 6
1.1 Основные понятия, структура и особенности существующих методов 6
1.2 Методы построения изонилиний 8
1.3 Постановка задачи 10
2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ, ТЕХНОЛОГИЙ И ПОДХОДА К ВЫЧИСЛЕНИЯМ 11
2.1 Использование среды разработки RAD STUDIO 11
2.2 Glscene 15
2.3. Использование библиотек VCL, GLSCENE 21
3. РЕАЛИЗАЦИЯ СИСТЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ 24
3.1 Реализация алгоритма визуализаци изоповерхности 24
3.2 Реализация алгоритма построения изоповерхности 32
3.3 Программные модули отображения и обработки информации 45
4. АПРОБАЦИЯ И ВНЕДРЕНИЕ АВТОМАТИЗИРОВАННОЙ СИСТЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ 51
ЗАКЛЮЧЕНИЕ 55
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 56
ПРИЛОЖЕНИЕ 58

ВВЕДЕНИЕ

Технологии, позволяющие получать правдоподобные изображения, развиваются быстрыми темпами. Пример тому изолинии в трехмерной графике. Изоповерхности традиционно используется в медицине для просмотра рентгеновских снимков, а также для построения магнитно-резонансных моделей, позволяющих видеть внутренности человека. Самое широкое применение получила, в компьютерной томографии. Изображения с использованием большого количества ультразвуковых или рентгеновских снимков под разными углами анализируются, и создается трехмерный массив плотности различных участков тканей исследуемого объекта. Этот массив представляется "объемной картиной", элементом которой является воксель.

Геологи к примеру создают трехмерные модели разрезов земной коры, основанные на эхолокации. Инженеры рассматривают модели материалов для выявления слабых мест их структуры . Пользователь ПК сталкивается с воксельной, как и вообще с трехмерной графикой, как правило в играх или другой мультимедийной продукции. Для такого рода ПО отображающего трехмерные объекты на экране монитора, является модуль, математический алгоритм просчета координат, цвета и прозрачности пикселов изображения на плоскости экрана.

Если говорить о перспективах применения систем визуализации, то имеются несколько основных областей.

Тренажеры. Требуют эффективной работы с большой базой данных, описывающей моделируемую обстановку в реальном времени. Алгоритмы решающие данные задачи, можно определить как алгоритмы визуализации незамкнутой базы данных. То есть это значит, что база данных не входит целиком в поле зрения. Перспективные и современные системы вооружений и военной техники, имеют принципиально новые требования к тренажерам и соотвецственно к компьютерным системам визуализации (КСВ), которые

используются для генерации изображения виртуальной реальности. К сложным задачам относятся симуляторы для подготовки милотов, а вчастности задания по полетам на сверх малых высотах и огибам рельефов местности.

Медицинские системы. Главное требование это достоверность информации, а так же качество визуализации. Время отображения и обновления изображения уже не так важно, база данных замкнута практически всегда (исключая микронавигацию и микрохирургию). Раньше в распоряжении врачей были лишь рентгеновские снимки, которые дают некоторое представление об исследуемых органах в виде наложения теней на изображения и отличаются плохой контрастностью и отсутствием какой-либо информации о глубине объектов. Использование компьютеров дало возможность развиваться новым направлениям томографической интроскопии, таким как компьютерная томография, магнитная резонансная томография, позитронная эмиссионная томография. Благодаря томографической аппаратуре возможно получить снимки множества сечений тела пациента, которые характеризуют особенности его анатомии, причем с большой четкостью и без наложения изображений органов друг на друга, это важно для медицинских задач, подобных хирургическому планированию, где важно понимать всю ее сложность и видеть дефекты 3D структуры.

Научная визуализация. Похожа на медицинскую, так как характер требований почти такой же. Особое значение имеет вопрос классификации и разделения значений (цвет и тип материала), а также предобработка данных, как и в задаче компьютерной томографии – восстановления функции поглощения её линейных интегралов. Любая теория проверяется на ценность тем, насколько она приложима к решению различных практически задач. Визуализация превращает данные численного моделирования в легко интерпретируемые зрительные образы и позволяет быстро просмотреть и оценить в целом результаты моделирования.

Целью данной работы было модернизировать и адаптировать модуль Isosurfaces для кроссплатформенного использования.

В ходе работы были поставлены следующие задачи:

1) провести анализ проблемы построения изоповерхностей;
2) Выбор инструментальных средств, технологий создания системы.
3) Реализовать алгоритм визуализации с использованием изолиний.
4) Апробация и внедрение полученной системы.

В выпускной квалификационной работе содержится 57 страниц без приложения, 22 рисунка и 5 формул. В процессе создания было использовано 20 литературных источников.

1. АНАЛИЗ ПРОБЛЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ

1.1 Основные понятия, структура и особенности существующих методов

Эффективным средством визуализации 3D комплектов данных является рендеринг изоповерхностей. Изоповерхности могут быть созданы для комплектов 3D данных типа GRIDS и MESH. Изоповерхность есть трехмерный эквивалент изолинии. Изолиния представляет собой некоторую линию постоянного значения, расположенную на изоповерхности. В свою очередь, изоповерхность представляет собой поверхность с постоянным значением показателя, построенную по массиву пространственных данных. Изоповерхности создаются только для видимых и активных ячеек или элементов.

Изоповерхности вычисляются по значениям активного скалярного массива данных для решеток GRIDS или сеток MESH. Диалог Опции изоповерхности становится доступным после нажатия кнопки на инструментальной панели или выполнении команды Опции изоповерхности в меню Карта.

Как и изолинии, изоповерхности являются временными по своей природе. Другими словами, при смене активного геопоказателя старая изоповерхность удаляется и вычисляется новая изоповерхность для текущего показателя. В некоторых случаях, полезно создание изоповерхностей в качестве постоянных объектов. Это можно сделать выбрав опцию Сохранить как сечения в диалоге Опции изоповерхностей. Это приведет к тому, что полученные в результате изоповерхности будут трактоваться как сечения. Они могут быть сохранены на диске, скрыты или удалены. Кроме того, если выбран новый показатель, изоповерхности не удаляются. Фактически,

значения, связанные с массивом нового геопоказателя, интерполируются в сечения изоповерхности и изоповерхности можно показать как цветные изоленты и изолинии. Это позволяет эффектно отображать два показателя одновременно.

ГИС - понятие геоинформационной системы также используется в более узком смысле — как инструмента (программного продукта), позволяющего пользователям искать, анализировать и редактировать как цифровую карту местности, так и дополнительную информацию об объектах.

Трёхмерное пространство – геометрическая модель материального мира, в котором мы находимся. Это пространство называется трёхмерным, так как оно имеет три однородных измерения — высоту, ширину и длину, то есть трёхмерное пространство описывается тремя единичными ортогональными векторами. Понимание трёхмерного пространства людьми, как считается, развивается ещё в младенчестве, и тесно связано с координацией движений человека. Визуальная способность воспринимать окружающий мир органами чувств в трёх измерениях называется глубиной восприятия. В аналитической геометрии каждая точка трёхмерного пространства описывается как набор из трёх величин — координат. Задаются три взаимно перпендикулярных координатных оси, пересекающихся в начале координат. Положение точки задаётся относительно этих трёх осей заданием упорядоченной тройки чисел. Каждое из этих чисел задаёт расстояние от начала отсчёта до точки, измеренное вдоль соответствующей оси, что равно расстоянию от точки до плоскости, образованной другими двумя осями.

Триангуляция – это разбиение геометрического объекта на симплексы. Например, на плоскости это разбиение на треугольники, откуда и название. Разные разделы геометрии используют несколько отличные определения этого термина.

Симплекс – геометрическая фигура, являющаяся n-мерным обобщением треугольника.

GPU – графический процессор, отдельное устройство персонального компьютера или игровой приставки, выполняющее графический рендеринг. Современные графические процессоры очень эффективно обрабатывают и отображают компьютерную графику, благодаря специализированной конвейерной архитектуре они намного эффективнее в обработке графической информации, чем типичный центральный процессор.

Топология — раздел математики, изучающий в самом общем виде явление непрерывности, в частности свойства пространства, которые остаются неизменными при непрерывных деформациях, например, связность, ориентируемость. В отличие от геометрии, в топологии не рассматриваются метрические свойства объектов (например, расстояние между парой точек). Дифференциальная геометрия и дифференциальная топология — два смежных раздела математики, которые изучают гладкие многообразия (обычно с дополнительными структурами). Эти два раздела математики почти неразделимы, при этом часто оба раздела называют дифференциальной геометрией. Различие между этими разделами состоит в наличии или отсутствии локальных инвариантов. В дифференциальной топологии рассматриваются такие структуры на многообразиях, что у любой пары точек можно найти идентичные окрестности, тогда как в дифференциальной геометрии, могут присутствовать локальные инварианты (такие как кривизна) которые могут различаться в точках.

1.2 Методы построения изолиний

Для алгоритма генерации изоповерхности существенно, что сетка состоит из тетраэдров, и скалярное поле линейно на каждом из них. Поэтому каждое ребро сетки пересекается с изоповерхностью не

более, чем в одной точке. Пересечение изоповерхности с тетраэдром — либо треугольник, либо четырехугольник, так как часть изоповерхности внутри каждого тетраэдра — плоская.

Особо следует отметить случай, когда значение поля в некотором узле в точности равно его значению на изоповерхности. При этом возникает

негрубая ситуация, сильно усложняющая весь алгоритм. Разработанные версии алгоритма избегают этой проблемы, добавляя малые слагаемые к тем узловым значениям поля, которые в точности равны значению на изоповерхности.

Алгоритм создания изоповерхности состоит из следующих шагов.

1. Определение диапазона $[f_{min}, f_{max}]$ узловых значений поля f.

2. Добавление малых слагаемых к узловым значениям, совпадающим с заданным на изоповерхности f0. Величина слагаемого выбирается равной εf_0, если f_0 6= 0 и ε $(f_{max} - f_{min})$, если $f_0 = 0$. Величина ε принимается равной 10−7, так как вычисления производятся в числах с плавающей запятой с одинарной точностью. Такое изменение поля не сказывается на видимой геометрии изоповерхности, но существенно упрощает алгоритм, устраняя негрубые ситуации.

3. В цикле по всем ребрам сетки домена выясняется, пересекается ли ребро с изоповерхностью. Если это так, ребру ставится в соответствие очередной номер узла сетки изоповерхности. Также вычисляется параметр от 0 до 1, определяющий положение этого узла на ребре.

4. Создание треугольников изоповерхности. На этом этапе необходимо обойти все тетраэдры, пересекающиеся с изоповерхностью, и сгенерировать для каждого из них обходы одного или двух треугольников, являющихся частью изоповерхности в данном тетраэдре. Чтобы обеспечить согласованность ориентации треугольников на соседних тетраэдрах, достаточно располагать узловыми значениями поля в каждом отдельном тетраэдре (предполагается, однако, что ориентации всех тетраэдров исходной сетки согласованы).

5. Создание узлов изоповерхности. На этом этапе вычисляются координаты узлов изоповерхности, фактически найденных на шаге 3.

6. Построение соответствия между узлами на краю изоповерхности и ребрами сетки домена на его границе. Этот шаг необходим для последующей сшивки кусков изоповерхностей на соседних доменах. Указанное соответствие позволяет при сшивке определить соответствующие друг другу узлы алгебраическим путем, не сравнивая их координаты.

Отметим, что реализация этого алгоритма на GPU нетривиальна, но может быть полностью сведена к последовательности стандартных алгоритмов for_each, transform, partition, sort, scan, gather, scatter, unique, remove_if, copy [1].

1.3 Постановка задачи

Модернизировать и адаптировать модуль Isosurfaces для работы в системе недропользования Geoblock для Win32, а также в Gexoblock для Win64, MacOS, Android, iOS, Linux. Интегрированным программным обеспечением для вычислительной геометрии и визуализации пространственных наборов данных.

Программа может быть использована в науках о Земле, особенно в геологическом моделировании. Визуализация и моделирование сложных объектов в системе недропользования. Оценки запасов руды и оптимизации добычи открытым способом.

2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ТЕХНОЛОГИЙ И ПОДХОДА К ВЫЧИСЛЕНИЯМ

В ходе постановки задачи было определено, что необходимо создать параллельный алгоритм поверхностной и сплошной вокселизации полигональных моделей. После изучения предложенных на рынке инструментов, было решено отдать предпочтение следующему набору:

1. Операционная система не ниже Windows7/8;
2. Среда программирования RAD Studio (язык C++);

2.1 Использование среды разработки RAD Studio

RAD Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. RAD Studio включает в себя несколько компонентов, таких как: Visual C#, Visual C++, Visual Basic .NET и Visual F#.

RAD Studio– выпущена 12 апреля 2012 года вместе с .NET Framework 4.0. RAD Studio включает поддержку языков C# 4.0 и Visual Basic .NET 10.0, а также языка F#, отсутствовавшего в предыдущих версиях.

Платформа .NET Framework — это встроенный компонент Windows, который поддерживает создание и выполнение нового поколения приложений и веб-служб. Основными компонентами .NET Framework являются общеязыковая среда выполнения (CLR) и библиотека классов .NET Framework, включающая ADO.NET, ASP.NET, Windows Forms и Windows Presentation Foundation (WPF)..NET Framework предоставляет среду управляемого выполнения, возможности упрощения разработки и развертывания, а также возможности интеграции со многими языками программирования. Основной идеей при разработке .NET Framework

являлось обеспечение свободы разработчика за счёт предоставления ему возможности создавать приложения различных типов, способные выполняться на различных типах устройств и в различных средах. Вторым принципом стала ориентация на системы, работающие под управлением семейства операционных систем Microsoft Windows. .NET Framework состоит из двух частей:

FCL представляет объектно-ориентированный API-интерфейс, используемый всеми моделями приложений. В ней содержатся описания типов, которые позволяют разработчикам выполнять ввод или вывод, планирование задач в других потоках, создавать графические образы, сравнивать строки и т.п.

.NET Framework предоставляет реальные возможности повторного использования кода, управления ресурсами, многоязыковой разработки, развертывания и администрирования. Вот некоторые преимущества технологии.

2.1.1 Отсутствие проблем с различной совместимостью версий

При использовании динамически подключаемых библиотек может возникнуть такая ситуация, когда компоненты, устанавливаемые для нового приложения, заменяют компоненты старого приложения, и в итоге последнее начинает работать неправильно или вообще перестает работать. Архитектура этой платформы позволяет изолировать прикладные компоненты, это позволяет приложению загрузить те компоненты, с которыми оно создавалось и тестировалось. Таким образом, если приложение работает после установки, то оно работает всегда.

2.1.2 Безопасность

Традиционные системы безопасности обеспечивают управление доступом на основе учетных записей пользователя. Это проверенная модель, но она подразумевает, что любому коду можно доверять в одинаковой степени. Такое допущение оправдано, когда весь код устанавливается с физических носителей или с доверенных корпоративных серверов. Но по мере увеличения объема мобильного кода, например сценариев из Интернета, нужен ориентированный на код способ контроля над поведением приложений. Такой подход реализован в модели безопасности доступа к коду.

Для разработки Windows-приложений среда .NET Framework поддерживает компонентный способ разработки программного обеспечения. Такой метод создания программного обеспечения появился относительно недавно. Его можно охарактеризовать как технологию создания программного обеспечения из готовых блоков. Т.е. программисты пытаются позаимствовать идею у строителей, занимающихся крупнопанельным домостроением. Создание программного обеспечения из компонентов подразумевает, что к проекту они будут добавляться только лишь во время разработки. Кроме того, дело не обойдётся и без их первоначальной настройки.

2.1.3 Триангуляция Делоне и диаграммы Вороного.

Разрешим V быть рядом точек в Rd., быть k-симплексом (0 К d), чьи вершины находятся в V. Описанная сфера является сферой, которая проходит через все вершины, как показано на рис.2.1. Если k = d, имеет уникальную описанную сферу, иначе, в пространстве есть много описанных сфер. Мы говорим, что это - Делоне, если там существует описанная сфера таким образом, что никакая вершина V не находится в нем. Триангуляция Делоне D

и V является симплициальным комплексом, таким образом, что все симплексы, и базовое пространство D - выпуклая оболочка V. Оставленный рисунок 1 иллюстрирует 2-ю Триангуляцию Делоне. 3-ю Триангуляцию Делоне также вызывают Delaunay tetrahedralization. Триангуляция Делоне V уникальна, если V находится в общей позиции, т.е., никакой d +2 точки в V лежат на общей сфере. Иначе, мы говорим, что V содержит степени вырождения, т.е., есть точки d+2 в V, лежат на общей сфере. Степени вырождения могут быть удалены, применив произвольное небольшое возмущение на координаты точек в V.

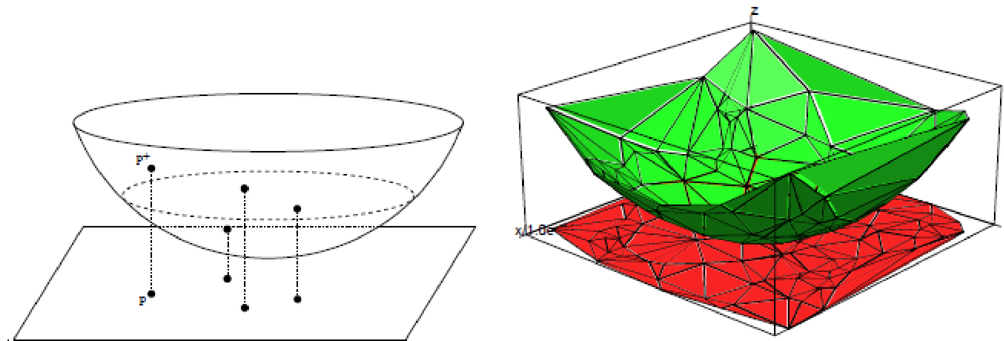

Рис. 2.1 Соотношение между триангуляции Делоне в Rd и выпуклыми Корпус в Rd + 1 (здесь d = 2)

Слева: Некоторые 2d точки и их соответствующие 3d точки подъема.

Справа: Триангуляция Делоне из набора точек и 2d ниже выпуклая оболочка его 3d поднял очки.

Для любой вершины p 2 V, Вороного клетка p множество точек с расстоянием до p не больше, чем на любой другой вершины V, то есть набор клеток (p) = FX 2 Rd; K обозначает евклидова расстояния. Вороного схема V является подразделением Rd в Вороного клеток (некоторые из которых могут быть неограниченными) и лицах. Это г-мерного многогранная комплексная. Если уставки В находится в общем положении, имеется взаимно однозначное соответствие между K-симплексов триангуляции Делоне и (г) □ к - полиэдрами диаграммы Вороного, где 0 один-к-одному, вершины диаграммы Вороного являются окружностей тетраэдров Делоне.

2.2 GLScene

Базовый компонент. Вы можете открыть редактор сцены, дважды щелкнув на нем.

Компонент представляет собой прямоугольную панель, в которой отображается ваша сцена. Ее размеры не ограничивают саму сцену. Чем больше растянута эта панель, тем медленнее прорисовывается сцена. Для отображения сцены в панели необходимо указать в свойствах камеру (TGLCamera), изображение с которой будет прорисовываться в вашем GLSceneViewer и собственно саму сцену (TGLScene). Вы можете установить здесь важное свойство – контекст рендеринга через свойство Buffer. Однако значения этого свойства по умолчанию в большинстве случаев вполне достаточно.

Material Library – это компонент для хранения материалов. Это библиотека материалов. Вы можете получить доступ к материалам через их индекс или имя сохраненного материала. В этом пункте немного объясню, как материалы обрабатываются в GLScene. Каждый объект, на который может быть наложен материал, имеет одноименное свойство – material. Вы можете редактировать материал непосредственно в инспекторе объектов. По двойному клику на значке многоточия напротив свойства material – открывается редактор материалов (Material Editor). Редактор материалов имеет три вкладки и окно с примером в виде куба с наложенным вами материалом. Три вкладки – это:

1. Front properties – редактирует качество материала лицевых граней объекта. Диффузный цвет (Diffuse) является наиболее важным. Он определяет цвет освещенных частей объекта. Окружающий цвет (Ambient) определяет цвет затененных частей объекта. Зеркальный цвет (Specular) «отвечает» за цвет отражений и бликов. Цвет эмиссии (Emission) определяет цвет свечения GLScene руководство новичка, Jat-Studio, 2009 объекта. Но пока только запомните, что для изменения основного цвета изменять нужно

именно диффузный цвет (Diffuse). Другие свойства материала как отражения или блики могут быть достигнуты более реалистично с помощью использования шейдеров.

2. Back properties – отличие от front properties в том, что эти свойства «отвечают» за материал невидимых граней объекта. Эти грани объекта становятся видимыми, только если материал прозрачен или отключен отбор (culling) задних граней.

3. Texture – используется для наложения на объект текстуры. Для включения видимости текстуры необходимо отключить свойство disabled. Это свойство поумолчанию включено, что означает, что объект не использует никакой текстуры. Объект в этом случае будет окрашен в соответствии с настройками двух предыдущих вкладок (Back и Front properties). Если вы хотите применить текстуру, то отключите блокирующий флажок disabled и загрузите изображение. GLScene поддерживает форматы jpg, tga, bmp. При использовании двух первых форматов необходимо сначала в разделе кода uses добавить модули jpeg или tga соответственно. Для реалистичного освещения вы должны установить для свойства Texture Mode текстуры значение tmModulate. Помните, что свет должен осветить объект, чтобы материал стал виден. Другая важная вещь – размеры текстуры должны быть кратны двум: 2,4,8,16,32,64,128,256,512,1024 и т. д. Причем длина и ширина текстуры могут и не совпадать. Вы, например, можете использовать текстуру размером 32x512. Если же вы будете использовать текстуру нестандартного размера, то она будет отображена медленнее, так как GLScene придется привести ее размеры к стандарту.

Внизу редактора материалов есть выпадающий список для выбора режима смешивания – blending mode. Здесь вы можете определить, как материал будет смешиваться или перекрываться другими материалами. Непрозрачный режим смешивания (bmOpaque) непрозрачный объект. Прозрачный режим (bmTransparent) позволить видеть сквозь объект. Объект может быть однородно прозрачным, или прозрачность может быть задана

текстурой. Совокупное смешивание (bmAdditive) комбинирует цвет объекта с цветом объектов позади него. Хотя для каждого объекта можно настроить свой материал, но настойчиво рекомендуется хранить все материалы в библиотеке материалов (GLMaterialLibrary). Особенно если объектов много и некоторые используют одну и ту же текстуру. Например, вы используете 100 кубиков и для каждого загружаете одну и ту же текстуру – в памяти разместятся 100 одинаковых текстур. Но вы можете загрузить эту текстуру один раз в MaterialLibrary и затем ссылаться на нее каждый раз, когда она необходима. Делается это с помощью кода GLCube->Material->MaterialLibrary для обращения к библиотеке материалов или же GLCube->Material->LibMaterialName для обращения к имени материала, который вам нужен. Осторожно! Объекты имеют свойство MaterialLibrary вы же долны использовать Material.MaterialLibrary. Не путайте их! Библиотека материалов имеет еще одну удобную функцию: AddTextureMaterial. В этой функции определяется имя нового материала и загружаемое в него изображение (текстура). Новый материал добавляется к библиотеке так: Texture.Disabled := False; и Texture.Modulation := tmModulate;

```
unit Unit1;

interface

uses
  Windows,
  Messages,
  SysUtils,
  Variants,
  Classes,
  Graphics,
  Controls,
  ExtCtrls,
  StdCtrls,
  ComCtrls,
  Buttons,
  Forms,
  Dialogs,
  GLWin32Viewer,
  GLCrossPlatform,
  GLBaseClasses,
  GLScene,
  GLObjects,
  GLCoordinates,
  GLColor,
  GLVectorGeometry,
  GLMesh,
  GLVectorFileObjects,
  GLState,
  GLGeomObjects,
  GLExtrusion,
  GLMarchingCubes,
  GLSimpleNavigation,
  GLMaterial;
```

Рис. 2.2 Пример используемых библиотек в GLScene

Большинство приложений, использующих GLScene, отрисовываются (рендерятся) в режиме реального времени. При этом время имеет большое значение и используют базовый набор библиотек как показано на рис. 2.2. Поэтому появляется необходимость в некотором менеджере времени. И это не самый простой компонент. Сначала, все что мы должны знать – это сколько времени будет рендериться сцена. Камера может быть направлена на сложные геометрические объекты с большим количеством полигонов, и, при вращении камеры все они должны перерисовываться. Причем ваша GLScene руководство новичка, Jat-Studio, 2009 программа может выполняться как на старой и медленной системе, так и на новейшей системе с высокой производительностью. Этот момент невозможно предугадать заранее. Если вы хотите использовать свою сцену в течение долгого времени – используйте GLCadencer. Этот компонент позаботится о необходимой синхронизации обновления объектов в сцене от кадра к кадру. Но сначала вы должны

настроить свойства этого компонента. Процесс перерисовки (рендеринга) кадра приводит к возникновению события Progress компонента GLScene. Каждый объект GLScene имеет событие onProgress, где можно запрограммировать некоторые действия программы, выполняющиеся каждый раз при перерисовке (рендеринге) сцены. Двойным кликом на объекте в инспекторе объектов к основному коду добавляется заготовка реакции на событие onProgress. Процедура Progress передает через параметры одну важную переменную – deltaTime. Это период времени в секундах, который прошел после рендеринга последнего кадра. Если этот параметр слишком велик, то значит, что сцена медленно рендерится и «тормозит». Идеальное количество отрендеренных кадров – 30 в секунду. При этом deltaTime равен 0,033333. Если вам необходимо провести какие-либо вычисления связанные со временем – включайте переменную deltaTime в ваши уравнения. Например, если вы хотите переместить куб вдоль оси X со скоростью 10 пунктов в секунду, то код будет выглядеть примерно так: GLCube.Position.X := GLCube.Position.X + 10 * deltaTime. Cadencer имеет свойство enabled. С его помощью можно просто включить или выключить компонент в нужный момент. Когда он выключен сцена будет заморожена. Cadencer может работать в нескольких различных режимах (GLCadencer.Mode). cmASAP – значение по умолчанию, сцена будет обрабатываться всякий раз с максимальным приоритетом, по сравнению с другими процессами. cmIdle – сцена будет обрабатываться только если завершены другие процессы и с cmManual вы сможете управлять запуском обработки сцены вручную. Другая интересная особенность – Cadencer.minDeltaTime. С помощью этого свойства вы можете установить время, только по истечении которого, начнется обработка сцены, даже если сцена уже отрендерена. Этим вы сможете несколько разгрузить систему. Cadencer.maxDeltaTime – напротив не позволит cadencer выполниться быстрее установленного времени.

. Без света сцена темная и нецветная. Свет делает ее светлой и яркой. Максимум можно иметь восемь источников света. Любой свет кроме параллельного имеет предел дальности свечения. От источника света к границе его свечения свет от этого источника постепенно уменьшается. Это называется ослаблением света. Свет от LightSource не создает теней. Если, например источник света направлен на сферу, а за ней находится плоскость, то на плоскости тени мы не увидим. Свет проходит через сферу, нисколько ее не замечая. Вы должны использовать другие методики, чтобы получить тени. Например, Lightmaps, Z-Shadows или Shadow Volumes. Существует три типа света:

1. Omni Light – источник света находится в определенной точке. Лучи от него расходятся радиально по всем направлениям. Вы можете, например, представить электрическую лампочку, висящую где-нибудь на проводе.

2. Spot Light – единичный луч света или конус направленного света. Вы можете изменить ширину и угол света. Если изменить угол на 360°, то свет станет типа Omni. Пример Spot Light – свет фонарика.

3. Parallel Light – однородная масса параллельных лучей с одинаковым направлением, которые светят от некоторой плоскости в бесконечность. Изменение позиции параллельного источника света не имеет никакого эффекта. Параллельный свет обычно используют для симуляции равномерного освещения.

GLFreeForm Этот объект используется довольно часто. Он способен загружать геометрию из различных форматов сетки (mesh). Чтобы формат смог использоваться, нужно добавить в раздел uses одноименный модуль формата файла. Например, чтобы использовать формат *.3ds* необходимо добавить модуль GLFile3DS. Для загрузки геометрии используется функция LoadFromFile конкретного GLFreeForm. Координаты тестуры обычно включены в файл. Некоторые форматы поддерживают мультитекстурирование (использование множества текстур одновременно для одного объекта) объектов. Вы должны установить используемые

текстуры в список Mesh list. Для успешной загрузки текстуры геометрия должна загрузиться перед ней.

GLSkyDome (купол неба) создает градиентный цвет, полосы которого расположены горизонтально. Вы можете использовать столько полос, сколько захотите. Вы можете добавить маленькие точки (звезды) в список Stars. Эти звезды могут сверкать.

GLLensFlare моделирует эффект, который возникает при взгляде камеры на сильный источник света в реальном мире. Этот эффект добавит больше реализма вашей сцене. GLLensFlare помещается обычно в ту же позицию, что и сам источник света. Кольца и полосы создаются при взгляде прямо на GLLensFlare. В настройках можно указать их количество, размер и качество. GLLensFlare не создает эффекта, если находится позади другого объекта, по отношению к камере.

2.3. Использование библиотек VCL, FMX, CGAL, GLScene

VSL (Библиоте́ка визуа́льных компоне́нтов) — объектно-ориентированная библиотека для разработки программного обеспечения, разработанная компанией Borland для поддержки принципов визуального программирования. VCL входит в комплект поставки Delphi, C++ Builder и Embarcadero RAD Studio и является, по сути, частью среды разработки, хотя разработка приложений в этих средах возможна и без использования VCL. VCL предоставляет огромное количество готовых к использованию компонентов для работы в самых разных областях программирования, таких, например, как интерфейс пользователя (экранные формы и элементы управления — т. н. «контролы», «контроли»), работа с базами данных, взаимодействие с операционной системой, программирование сетевых приложений и прочее

Библиотека алгоритмов вычислительной геометрии (CGAL) - это программная библиотека алгоритмов вычислительной геометрии. Хотя в

основном написанные на C ++, в настоящее время доступны и привязки Scilab и привязки, созданные с помощью SWIG (поддерживающие Python и Java).

Программное обеспечение доступно по схеме двойного лицензирования. При использовании для другого программного обеспечения с открытым исходным кодом, он доступен под лицензиями с открытым исходным кодом (LGPL или GPL в зависимости от компонента). В других случаях коммерческая лицензия может быть приобретена в рамках различных вариантов для академических / исследовательских и промышленных клиентов.

GLScene — графический движок для создания кросс-платформенных приложений на языках программирования Delphi, Pascal и C, и использующий библиотеку OpenGL в качестве основного интерфейса при создании приложений.

GLScene позволяет создавать сцены, содержащие трехмерные модели. Множество объектов и дополнительных визуальных компонентов VCL помогает программистам создавать 3D-приложения для Delphi, C++Builder и Lazarus.

Поддерживаемые форматы файлов моделей: 3ds, obj, vrml, smd, md2, md3, nmf, oct, lwo, b3d, gl2, gls, ms3d, Nurbs, lod, и некоторые другие.

Сохраняемые форматы файлов моделей: glsm, obj и smd.

Поддерживаемая физика: ODE, Newton Game Dynamics. Также есть небольшой собственный движок расчёта столкновений с учётом законов сохранения импульса DCE.

FireMonkey - это кроссплатформенная GUI-инфраструктура, разработанная Embarcadero Technologies для использования в Delphi, C ++ Builder и AppMethod с C ++ или Object Pascal для создания кросс-платформенных приложений для Windows, macOS, iOS и Android.

FireMonkey - это кроссплатформенная структура пользовательского интерфейса и позволяет разработчикам создавать пользовательские

интерфейсы, которые работают в Windows, MacOS, iOS и Android. Он написан, чтобы по возможности использовать GPU, а приложения используют преимущества аппаратного ускорения, доступные в Direct2D в Windows Vista, Windows 7 и Windows 8, OpenGL на macOS, OpenGL ES на iOS и Android, а также на платформах Windows, где Direct2D Недоступен (например, Windows XP), он возвращается к GDI +.

Приложения и интерфейсы, разработанные с FireMonkey, разделены на две категории: HD и 3D. Приложение HD является традиционным двумерным интерфейсом; Это называется HD, потому что FireMonkey - это полностью векторная библиотека UI и масштабируется без потери определения. Второй тип, 3D-интерфейс, обеспечивает среду 3D-сцены, полезную для разработки визуализации. Эти два элемента могут быть свободно перемешаны с 2D-элементами (обычными элементами управления пользовательского интерфейса, такими как кнопки) в трехмерной сцене, как наложением или в 3D-пространстве, так и 3D-сценами, интегрированными в обычный 2D-интерфейс HD. В инфраструктуре встроена поддержка эффектов (таких как размытость и свечение, а также других) и анимации, что позволяет легко создавать современные интерфейсы в стиле WPF. Он также поддерживает родные темы, поэтому приложение FireMonkey, хотя обычно использует элементы управления FireMonkey, может быть очень близко к native на каждой платформе. Собственные элементы управления можно использовать в macOS, iOS и Android через сторонние библиотеки.

FireMonkey - это не только визуальный каркас, но и полноценная среда разработки программного обеспечения, и многие функции, доступные в VCL. Основные различия:

- Кроссплатформенная совместимость
- Векторная графика элементов интерфейса

Любой визуальный компонент может быть дочерним элементом любого другого визуального компонента, что позволяет создавать гибридные компоненты.

3. РЕАЛИЗАЦИЯ СИСТЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ

3.1 Реализация алгоритма визуализации изоповерхности

Некоторые алгоритмы используют методы обработки изображений для поиска структур в трехмерных данных [1,32,30,29] или для фильтрации исходных данных. Данные MR, в частности, требуют обработки изображения для выбора подходящей структуры. Построение поверхности, тема этой статьи, предполагает создание модели поверхности из трехмерных данных. Модель обычно состоит из трехмерных элементов объема (вокселов) или многоугольников. Пользователи выбирают желаемую поверхность, задавая значение плотности. Этот шаг может также включать создание разрезанных или укупоренных поверхностей. Создав поверхность, последний шаг отображает эту поверхность с помощью методов отображения, которые включают в себя лучевой кастинг, затенение глубины и затенение цвета. Существует несколько подходов к решению проблемы 3D-поверхности. Ранняя техника [23] начинается с контуров строящейся поверхности и соединяет контуры на согласованных срезах с треугольниками. К сожалению, если на срезе существует более одного контура поверхности, возникают неоднозначности при определении контуров для соединения [14]. Интерактивное вмешательство пользователя может преодолеть некоторые из этих неоднозначностей [8]; Однако в клинической среде взаимодействие с пользователем должно быть сведено к минимуму.

Другой подход, разработанный Г. Германом и коллегами [19], создает поверхности из куберилл. Куберилл - это «рассечение пространства на равные кубики (называемые вокселями) тремя ортогональными наборами параллельных плоскостей» [7]. «Хотя существует множество способов

отображения модели куберилл, наиболее реалистичные результаты возникают, когда градиент, Вычисляемый по кубериллам по соседству, используется для нахождения тени точки на модели [15]. Миггер [25] использует представление октодерева, чтобы сжать хранилище трехмерных данных, что позволяет быстро манипулировать и отображать воксели.

Farrell [12] использует лучевой кастинг, чтобы найти 3D-поверхность, но вместо того, чтобы затенять изображение с помощью серой шкалы, использует оттенок для отображения поверхности. В другом методе лучевого литья Хохне [22] после определения поверхности вдоль луча вычисляет градиент вдоль поверхности и использует этот градиент, масштабируемый «подходящим» значением, чтобы создать серые шкалы для изображения.

Другой подход, используемый в клинике Майо [26], отображает объем плотности, а не поверхность. Этот метод фактически создает обычный теневой граф, который можно рассматривать под произвольными углами. Движение усиливает трехмерный эффект, полученный с использованием объемной модели.

Каждый из этих методов построения и устранения поверхностей имеет недостатки, поскольку они отбрасывают полезную информацию в исходных данных. Связанные контурные алгоритмы отбрасывают межслоевую связность, существующую в исходных данных. Метод куберилл, использующий пороговое значение для представления поверхности в виде блоков в 3D-пространстве, пытается восстановить информацию о затенении из блоков. Методы лучевого литья либо используют только глубинную штриховку, либо пытаются приблизить затенение с ненормализованным градиентом. Так как они отображают все значения, а не только видимые с данной точки зрения, то модели объема полагаются на движение для создания трехмерного ощущения.

Наш подход использует информацию из исходных трехмерных данных, чтобы получить взаимосвязь между фрагментами, местоположение поверхности и поверхностный градиент. Полученная треугольная модель

может быть отображена на обычных графических системах отображения, используя стандартные алгоритмы рендеринга.

В нашем подходе к проблеме поверхностного строительства есть два основных шага. Сначала мы определяем поверхность, соответствующую заданному пользователем значению, и создаем треугольники. Затем, чтобы обеспечить качественное изображение поверхности, мы вычисляем нормали к поверхности в каждой вершине каждого треугольника.

Маршевые кубы используют подход «разделяй и властвуй» для определения поверхности в логическом кубе, созданном из восьми пикселей; Четыре из двух соседних секций.

Алгоритм определяет, как поверхность Так как в каждом кубе имеется восемь вершин, а внутри и снаружи два состояния, то только 28 = 25 6 способов поверхность может пересечь куб. Перечисляя эти 256 случаев, мы создаем таблицу для поиска пересечений поверхностных кромок, учитывая маркировку вершин кубов. В таблице указаны ребра, введенные для каждого случая.

Триангуляция 256 случаев возможна, но утомительна и подвержена ошибкам. Две различные симметрии куба уменьшают проблему с 256 случаев до 14 шаблонов. Во-первых, топология триангулированной поверхности остается неизменной, если отношение значений поверхности к кубам меняется на обратное. Завершающие случаи, когда вершины, превышающие значение поверхности, заменяются теми, которые меньше значения, эквивалентны. Таким образом, нужно рассматривать только случаи с нулевым числом до четырех вершин, превышающим поверхностное значение, уменьшая число случаев до 128. Используя второе свойство симметрии, вращательную симметрию, мы редуцировали задачу до 14 шаблонов путем инспекции.

Простейшая модель, 0, встречается, если все значения вершин выше (или ниже) выбранного значения и не создает треугольников. Следующий шаблон, 1, встречается, если поверхность отделяется от вершины от

остальных семи, в результате получается один треугольник, определяемый тремя пересечениями кромок. Другие узоры образуют многогранные треугольники. Перестановка этих 14 основных моделей с использованием дополнительной и вращательной симметрии приводит к 256 случаям.

Мы создаем индекс для каждого случая, исходя из состояния вершины. Используя нумерацию вершин, восьмибитовый индекс содержит по одному биту для каждой вершины.

Этот индекс служит указателем на таблицу ребер, которая дает все пересечения ребер для заданной конфигурации куба.

Используя индекс, чтобы определить, к какому краю пересекает поверхность, мы можем интерполировать пересечение поверхности по краю. Мы используем линейную интерполяцию, но экспериментировали с более высокой степенью интерполяции. Так как алгоритм производит по крайней мере один и целых четыре треугольника на куб, поверхности более высокой степени показывают небольшое улучшение по сравнению с линейной интерполяцией.

Последний шаг в марширующих кубах вычисляет единичную нормаль для каждой вершины треугольника. Алгоритмы рендеринга используют этот обычай для получения изображений, затененных Гуро. Поверхность постоянной плотности имеет нулевую градиентную составляющую вдоль тангенциального направления поверхности; Следовательно, направление вектора градиента, является нормальным к поверхности. Мы можем использовать этот факт для определения вектора нормали к поверхности, 7i, если величина градиента отлична от нуля. К счастью, на поверхности, представляющей интерес между двумя типами ткани разных плотностей, градиентный вектор отличен от нуля. Вектор градиента, g, является производной функции плотности

$$S(x, y, z) = V \sim f(x, y, z). \quad (3.1)$$

Чтобы оценить вектор градиента на интересующей поверхности, сначала оценим векторы градиента в вершинах куба и линейно интерполируем градиент в точке пересечения. Градиент в вершине куба оценивается

Итак, марширующие кубы создают поверхность из трехмерного набора данных следующим образом:

1. Прочтите четыре фрагмента в память.

2. Сканируйте два кусочка и создайте куб из четырех соседей на одном срезе и четырех соседей на следующем фрагменте.

3. Вычислите индекс для куба, сравнив восемь значений плотности в вершинах куба с поверхностной константой.

4. Используя индекс, найдите список ребер из предварительно рассчитанной таблицы.

5. Используя плотности в каждой вершине ребра, найдите пересечение поверхностного края с помощью линейной интерполяции.

6. Вычислите единичную нормаль в каждой вершине куба, используя центральные различия. Интерполируйте нормаль к каждой вершине треугольника.

7. Выведите вершины треугольников и нормали вершин.

3.1.2 Алгоритм тетра-кубов

Идея граничных кубов следующая, данные представляются послойно, где логические кубы состоят из восьми соседних точек; по четыре из двух соседних планов.

Следующий шаг состоит из получения тетраэдров из существующих логических кубов (гексаэдров), здесь алгоритм использует корреляцию между гексаэдром и тетраэдром. Каждый куб может быть разделен на группу из пяти тетраэдров, где вершины тетраэдров совпадают с соответствующими исходными вершинами куба.

После создания тетраэдров алгоритм пересекает поверхность с каждым из пяти тетраэдров в группе и перемещается к следующей группе тетраэдров, которые были получены из соседнего гексаэдра.

Как в граничных кубах, такой же порядок поиска пересечения с тетраэдром, если значение данных превосходит или равно значению изоповерхности конструируемой нами, мы присваиваем вершине "1", которая соответствует вершине расположенной внутри или на поверхности. Вершинам тетраэдра со значением меньше изоповерхности присваивается "0" и они расположены вне поверхности. Те ребра тетраэдра, которые имеют вершины внутри и снаружи, пересекаются изоповерхностью. Это позволяет нам найти форму поверхности с тетраэдрами двоичным способом. Так как у каждого тетраэдра четыре вершины и два возможных значения для каждой из них, то существует ровно 24 способов пересечения тетраэдра изоповерхностью, включая обе ситуации, когда тетраэдр полностью внутри или снаружи изоповерхности, поэтому нет пересечений с ребрами.

Однако, использование симметрии тетраэдра позволяет свести эти случаи всего лишь к трем, как показано на рис. 3.1.

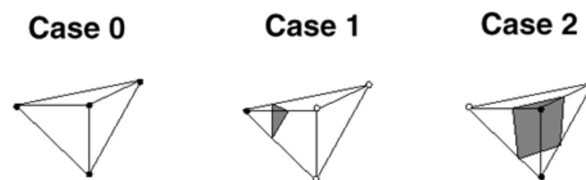

Рис. 3.1 Минимум возможных пересечений тетраэдров

С этими тремя случаями мы создаем таблицу просмотра пересечений поверхности и ребер, где содержимое таблицы - ребра пересекаемые в каждом случае. Далее следующая шаг граничных кубов - мы создаем индекс для каждого случая, основываясь на состоянии вершин. Каждая вершина представляется двоичной позицией в этом индексе, который описывает случай, обрабатываемый нами.

Этот индекс используется как указатель в таблице ребер, что дает все пересечения ребер для данной конфигурации тетраэдра. Использовав индекс

для нахождения ребер пересекающихся с изоповерхностью, мы используем линейную интерполяцию для нахождения точки пересечения на ребре. В заключение, после окончания интерполяции для каждого ребра, создаются треугольники содержащиеся в поверхности, путем соединения всех точек пересечения в данной ячейке. Эта работа повторяется, от ячейки к ячейке, до завершения генерирования изоповерхности.

Отдавая должное безусловной симметрии использования новых базовых ячеек - тетраэдров, надо учесть проблему связей между группами пяти тетраэдров получаемых из соседних областей гексаэдров. Эта проблема в том, что соседние тетраэдры в различных соседних гексаэдрах должны делить идентичные грани с идентичными ребрами и вершинами. Эта проблема не возникает, если некоторая группа тетраэдров получается из смежных кубов на рис. 3.2.

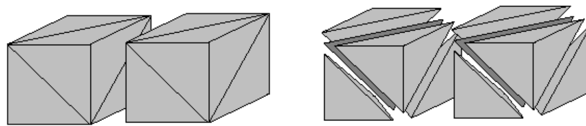

Рис. 3.2 Неправильная связь смежных кубов

Для разрешения этой ошибочной ситуации, мы можем создать другую группу тетраэдров, используемых в смежном кубе. Эта новая группа получается при повороте предыдущей на девяносто градусов. Другими словами, мы реализовали "зигзаг" или "шахматный порядок" альтернативных групп тетраэдров по направлению трех осей для решения этой проблемы (см. рис. 3.3).

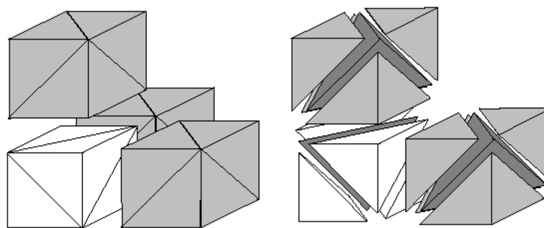

Рис. 3.3 Зигзагообразная связь

Двоичный подход используемый для рассмотрения случаев приводит к двусмысленности в реализации граничных кубов. Этого не возникает в реализации алгоритма тетра-кубов из-за геометрических различий между гексаэдрами и тетраэдрами, что мы можем видеть в сравнении на рис. 3.3, который изображает некоторую двусмысленную ситуацию на двух различных базовых ячейках.

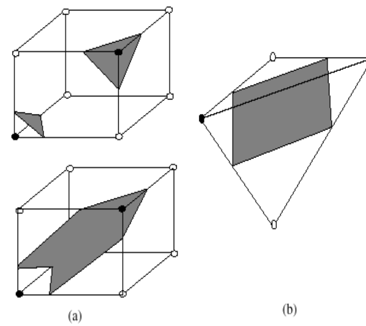

Рис. 3.4 Пример двух возможных поверхностей внутри куба

Пример двух возможных поверхностей внутри куба, основанных на двух "черных" вершинах, которые определяют двойственность. Аналогичный, описанному на рис. 3.4, случай использования тетраэдра как базовой ячейки. Не возникает двусмысленности, только одна поверхность возможна.

Так как отсутствует двусмысленность используем тетраэдры как базовые ячейки, ускорение реализации для алгоритма граничных кубов можно совершить комбинированием алгоритмов граничных кубов и тетра-кубов. Кубы будут использоваться в качестве базовых ячеек, сохраняется обход кубов с "шахматной расцветкой". При получении двусмысленности применяется алгоритм тетра-кубов для пересечения поверхности с этим проблематичным кубом, после деления его на правильную группу пяти тетраэдров. Алгоритм граничных кубов будет применяться для всех остальных беспроблемных гексаэдров. Используя эту идею, алгоритм тетра-кубов представляет согласованное и элегантное правило для разрешения двусмысленности присутствующей в алгоритме граничных кубов. Важно

заметить, что двусмысленности нет если мы применяем линейную интерполяцию для вычислений совместно с тетраэдрами (см. рис. 3.4).[12] метод, который обрабатывает случаи, где интерполяция выполняется на исходном кубе.

Подход нерегулярных решеток: В отличие от подхода граничных кубов, который получает в качестве данных только значение плотности, неявно подразумевая регулярную решетку кубов, наш подход получает не только значение плотности для каждой точки, но также и 3D координаты для них. Данный подход не подразумевает регулярной решетки; фактически он создает нерегулярную решетку тетраэдров, так как координаты взятые из данных могут быть совершенно не регулярными. Поэтому алгоритм тетра-кубов получает как данные координаты для каждой точки, которые могут быть совершенно не регулярными, и он еще может создавать нерегулярные решетки тетраэдров. Эти нерегулярные координаты используются алгоритмом в качестве нерегулярных вершин тетраэдров, основываясь на этих вершинах, алгоритм тера-кубов, создает высокоточную 3D поверхность на нерегулярной решетке.

2.2 Реализация алгоритма построения изоповерхности

Отображение изоповерхностей - стандартная методика научной визуализации объемных данных, и алгоритм Marching Cubes (MC) [16] обычно используется для построения изоповерхностей, которые представлены как треугольные сетки. Основным недостатком этого метода является то, что он создает сетки со многими маленькими и плохо сформированными треугольниками. Такие сетки требуют улучшения с прореживанием, сглаживанием или перекомпоновкой. Эти алгоритмы последующей обработки могут быть очень дорогими с точки зрения времени и потребления памяти, особенно если ячейки являются большими. И с

разрешением современных сканирующих устройств выходная сетка MC может легко состоять из миллионов треугольников.

Поэтому мы предлагаем понизить масштаб набора данных тома и создать иерархию томов, как описано в разделе 3. Затем мы используем MC для извлечения изоповерхности на самом грубом разрешении и подгонки сетки к изо-поверхностям на Более точные уровни иерархии томов позже. Так как количество треугольников в извлеченной сетке квадратично зависит от разрешения объема, выполнение MC на самом грубом уровне дает сетку с низкой сложностью, которая может быть оптимизирована эффективно. Мы применяем стратегию для улучшения сетки MC, удаляя короткие края, чтобы получить базовую сетку с небольшими и хорошо сформированными треугольниками.

Как только эта базовая сетка построена, мы используем ее в качестве первоначального предположения для аппроксимации изоповерхности на следующем более тонком уровне объема и итерации этого процесса подбора до тех пор, пока мы не придем к изоповерхностной реконструкции по отношению к исходным данным. Наша процедура подгонки обсуждается в разделе 4 и учитывает три аспекта.

Во-первых, вершины сетки необходимо проецировать на изоповерхность, поскольку мы хотим пробовать эту поверхность. Во-вторых, оператор релаксации требуется для равномерного распределения точек выборки по поверхности и для обеспечения правильной формы треугольников в конечной сетке. В-третьих, мы адаптивно подразделяем сетку для того, чтобы аппроксимировать изоповерхность в пределах заданной пользователем точности и захватить мелкие детали. Таким образом, мы наконец получаем полурегулярную сетку с иерархической структурой, которая может быть использована многими алгоритмами мультиразрешения, такими как рендеринг уровня детализации [3], прогрессивная трансмиссия [10, 14], редактирование с несколькими разрешениями [31] , Вейвлет-декомпозиции и реконструкции [17, 22].

В качестве применения метода мы восстановили поверхности археологических артефактов, подобных тем, которые показаны на рис. 3.5.

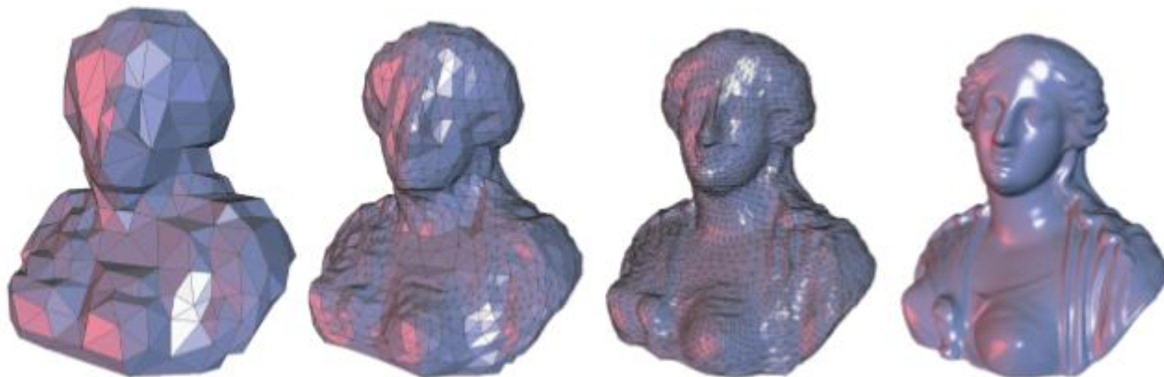

Рис. 3.5 Первые три уровня и окончательный результат нашего иерархического алгоритма выделения изоповерхности

Стандартный подход для извлечения изо-поверхностей из объемных данных - это хорошо известный алгоритм Марширующих Кубов [16]. Алгоритм проходит через все ячейки правильной шестигранной сетки и независимо вычисляет изоповерхность для каждой ячейки. Чтобы избежать двусмысленностей, было предложено несколько модификаций [19, 20], а расширение для восстановления поверхностей с резкими характеристиками из объемов расстояний было представлено [13]. Чтобы улучшить производительность, в нескольких алгоритмах [6, 27, 29] используются адаптивные иерархии набора данных тома

Задача преобразования произвольно триангулированной сетки в полурегулярную сетку называется пересозданием. В подходе вершины распределены по заданной триангуляции, а базовая сетка построена путем роста геодезических плиток Вороного вокруг вершин. Параметризация данной триангуляции внутри базовых треугольников вычисляется с помощью гармонических отображений, которые минимизируют локальное искажение. Затем ремиз определяется путем униформного разбиения каждого базового треугольника и отображения вершин в 3-пространство с помощью параметризации [15]. Построить базовую сетку путем сокращения

сетки на основе обрушения краев и инкрементально вычислить параметризацию исходной триангуляции внутри треугольников оставшейся сетки. Этот процесс приводит к локально гладкой параметризации. Чтобы достичь глобальной гладкости, двоичные точки перемещаются по варианту схемы подразделения Loop и отображаются в 3-пространство [12]. Описывают подход к обертыванию для перекомпоновки. Идея заключается в размещении полурегулярной сетки вокруг исходной поверхности. Аналогично физической термоусадочной обмотке путем вытягивания воздуха между обеими поверхностями полурегулярная сетка усаживается на поверхность. Кроме того, для равномерного распределения вершин по поверхности используется сила релаксации.

Прямое извлечение полурегулярных сеток из объемных данных рассматривается в нескольких работах. [2] используйте для извлечения начальной изоповерхности, которая укрупняется с помощью алгоритма упрощения сетки, основанного на [7]. Затем они используют модифицированный подход к обертыванию для вычисления своей конечной полурегулярной сетки на основе схемы четырехстороннего разбиения. Метод непосредственного вытягивания грубой основной сетки из объема был представлен [30]. Они вычисляют контуры поверхности по данным объема и связывают их так, что они образуют грубую сетку, которая топологически эквивалентна желаемой изо-поверхности. Конечная полурегулярная сетка строится с использованием многомасштабного решателя на основе силы с внешней силой, перемещающей вершины к изоповерхности, и внутренней силой, расслабляющей вершины сетки.

$$N_x,\ n_y\ и\ n_z,$$

$$G_0 = \{(x_i, y_j, z_k): 0 < i < n_x,\ 0 < j < n_y,\ 0 < k < n_z\} \tag{3.2}$$

$$C\ Xi = x_o + ih_x,\ y_j = y_o + jh_y,\ z_k = z_o + kh_z.$$

Чтобы упростить обозначение, мы далее предполагаем согласованный размер сетки $h = h_x = h_y = h_z$. Тогда иерархия f_0, f_1, f_2, ..., где каждая f^l определена на сетке Gl с размером сетки 2lh и размерами [2-ln$_x$ ~ 2-n$_z$], может тогда вычисляется путем итеративного понижения дискретизации объемных данных в два раза. Этот процесс обычно реализуется путем свертывания функции f^{l-1} с подходящим фильтром, а затем выборки отфильтрованного сигнала для получения f^l.

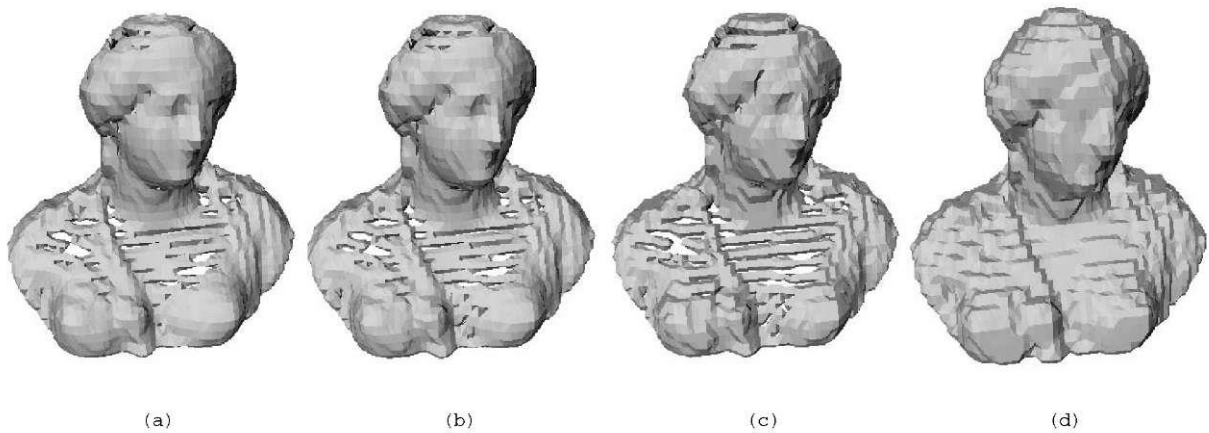

Рис. 3.6 Изоповерхность M2 (900) с использованием фильтровального фильтра (a), фильтра Гаусса (b), медианного фильтра (c) и дилатации (d) для вычисления f^2

Мы предполагаем, что серые значения объекта, которые мы хотим реконструировать, больше, чем серые значения окружающих вокселей. Мы протестировали несколько фильтров, включая бокс, Гаус и медианный фильтр, но нашли, что оператор дилатации лучше всего работает в рамках наших исследований, как показано на рис. 3.6. Этот оператор выбирает наибольшее значение серого из группы из восьми вокселов На уровне l – 1, которые объединяются для образования соответствующего вокселя с двойной длиной ребра на уровне l и определяют Где i, j и k кратны 2l. Учитывая изо-значение v, мы можем теперь извлечь аппроксимацию M^l (v) соответствующей изоповерхности из набора данных с пониженной дискретизацией f^l со стандартным алгоритмом марширующих кубов [16].

Как показано на рис. 3.6, что использование фильтров нижних частот имеет тенденцию вымывать тонкие слои воксела, которые представляют материал объекта. Это может привести к топологическим отверстиям, как показано на рисунках 2 (a) – (c). Напротив, оператор расширения имеет растущий эффект и для фиксированного значения iso M^l может фактически быть доказано, что он охватывает сетки.

Для того чтобы эффективно создать базовую сетку с несколькими треугольниками, мы запускаем алгоритм марширующих кубов на грубом томе, который вычисляется путем понижающей дискретизации данных. Поскольку количество треугольников, генерируемых марширующими кубами, квадратично зависит от числа вокселей в каждом измерении, уменьшение объема в n раз снижает сложность извлеченной сетки на n^2. Предположим, что данные объема представляются в виде дискретной серой функции значения f_0: $G_0 \rightarrow IN$, определяемой на регулярной сетке размерности

$$N_x,\ n_y\ u\ n_z,$$

$$G_0 = \{(x_i,y_j,z_k):0\leq i\leq n_x, 0\leq j\leq n_y, 0\leq k\leq n_z\} \quad (3.3)$$

Затем может быть вычислен Итеративно уменьшать объемные данные в два раза. Этот процесс обычно реализуется путем свертывания функции f^{l-1} с подходящим фильтром, а затем выборки фильтрованного сигнала для получения f^l. Мы предполагаем, что серые значения объекта, которые мы хотим восстановить, больше, чем серые значения окружающих вокселов. Мы протестировали несколько фильтров, в том числе коробку, Гаусс и медианный фильтр, но обнаружили, что оператор дилатации лучше всего работает в рамках наших исследований. Этот оператор выбирает наибольшее значение серого из кластера из восьми вокселов на уровне L^{-1}, которые объединяются, чтобы сформировать соответствующий воксель с двойной длиной ребра на уровне l и определяет:

$$f^l(x_i,y_j,z_k) \quad = \quad max\, i^{fl-1}(x_i,\, y_j,\, z_k) \qquad (3.4)$$

где i, j и k кратны 2^l. Учитывая изо-значение v, мы можем теперь извлечь аппроксимацию M^l (v) соответствующей изоповерхности из набора данных с пониженной дискретизацией f^l со стандартным алгоритмом маршевых кубов [16]. Использование фильтров нижних частот имеет тенденцию вымывать тонкие слои воксела, которые представляют материал объекта. Это может привести к топологическим отверстиям, как показано на рисунках 2 (a) – (c). Напротив, оператор расширения имеет растущий эффект, и для фиксированного значения iso M может фактически быть доказано, что он охватывает сетки

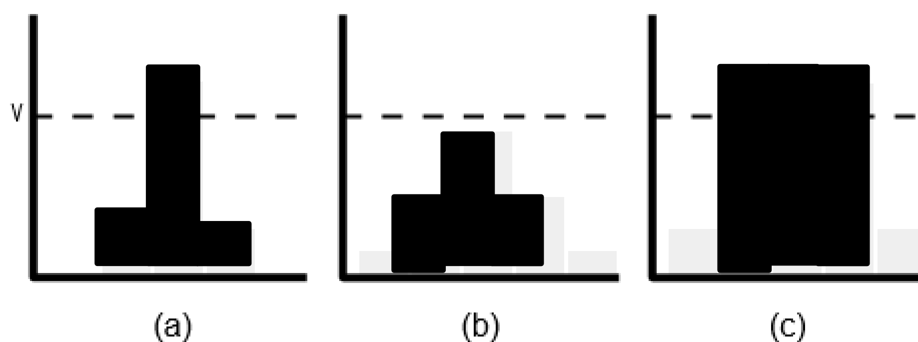

Рис. 3.7 Фильтрация входного сигнала: (a) исходные воксели, (b) воксели после фильтрации нижних частот, (c) воксели после расширения

M^{l-1}, M^{l-2}, ..., M^0, извлеченных из более низких уровней, как показано на рис. 3.7. Хотя этот метод может изменять топологию изоповерхности, поскольку малые дырки могут исчезать в результате расширения в целом , Он подходит для рассматриваемых множеств данных, поскольку они топологически простые. Типичным явлением алгоритма маршевых кубов является то, что некоторые из сгенерированных треугольников очень малы. В самом деле, всякий раз, когда разность δ серого значения вокселя и заданного значения i_s v мала, алгоритм отсекает угол лежащей ниже сетки и создает треугольник, размер которого пропорционален δ. Из-за их крошеч- ности

разумно предположить, что эти треугольники не содержат значи- тельной геометрической информации. Поскольку мы, в конечном счете, нацелены на создание триангулированной изоповерхности с равномерно распределенными вершинами, мы выполняем этап децимации перед дальнейшей обработкой сетки. Чтобы удалить все ребра, которые короче некоторой пороговой длины $\alpha 2^{l}h$ с $\alpha > 0$, мы сначала заменим все треугольники тремя короткими ребрами на одну вершину в их барицентре (см. рис. 3.8 (а)), а затем свернуть остальные Короткие края к их средним точкам (см. рис. 3.8 (б)). Мы обнаружили, что $\alpha = 0,5$ является хорошим выбором, и эта простая стратегия уменьшает количество треугольников примерно на 20%.

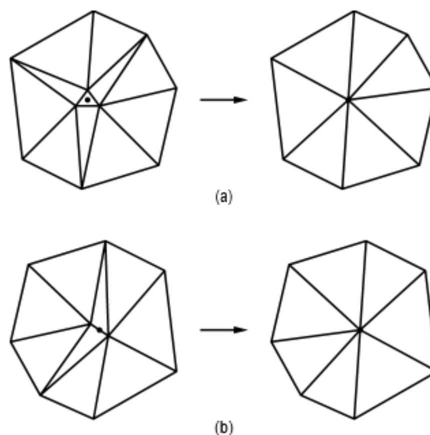

Рис. 3.8 Удаление коротких ребер из извлеченной изоповерхности с помощью алгоритма прореживания с двумя проходами

3.2.1 Алгоритм построения изолиний

Из-за растущего эффекта оператора дилатации вершины базовой сетки не лежат на изоповерхности на уровне 0, которую мы на самом деле хотим восстановить, и нам приходится сжимать сетку на этой поверхности. Чтобы повысить надежность и производительность этого алгоритма, мы используем ранее созданную иерархию томов путем итеративного фитирования сетки на следующий более низкий уровень. Сначала мы перемещаем вершины на

изоповерхность на уровне l -1, затем на уровень на уровне l-2 и так далее, пока мы не достигнем уровня 0. Обратите внимание, что это всегда гарантирует, что вершины текущей сетки Быть близко к изо-поверхности, а именно в пределах расстояния 2 воксела. Это помогает избежать самопересечений триангуляции после проецирования вершин, которые могут возникнуть, если расстояние слишком велико, как упомянуто в [12]. Существенным шагом нашего иерархического алгоритма выделения изоповерхности является адаптивная привязка текущей сетки к изоповерхности объема на определенном уровне l. Такая изоповерхность определяется как $S^l(v)$.

$$S^l(v)=\{(x,y,z): \sim f^l\ (x,y,z)=v\} \tag{3.5}$$

где ~ f^l: $[G^l]$ → IR – непрерывное расширение f^l, трилинейно интерполирующее значения f^l (G^l).

Поскольку изоповерхности S^{l+1} и S^l различны, вершины текущей сетки не будут лежать на S^l, и нам нужен метод их проецирования на эту поверхность. В принципе это можно сделать, найдя первое пересечение луча, исходящего из этой вершины, в определенном направлении с S^l, но остается вопрос, как определить направление этого луча. Например, мы могли бы использовать градиент функции серого значения f^0, как это часто делается при визуализации объема [11, 21, 28]. Хотя этот выбор хорошо работает в медицинских приложениях, мы считаем его неприемлемым для наших данных по следующей причине. Объекты, которые мы хотим восстановить, сделаны из довольно однородного материала. Если объемные данные имели бесконечно малую разрешающую способность, в идеале это был бы набор бинарных данных с нулевым значением серого в тех вокселях, которые представляют окружающий объект воздух и зависящее от материала постоянное значение серого во всех других вокселях. Поэтому градиент функции серого значения либо равен нулю, либо не определен. На практике

это предложение не выполняется, поскольку любое сканирующее устройство имеет только конечное разрешение и чувствительно к измерению ошибок. Однако мы обнаружили, что градиент серого значения слишком шумный для наших целей. Другим вариантом является градиент функции расстояния d^l: $[G^l] \rightarrow IR$, которая дает кратчайшее расстояние между точками на изоповерхности S^l [8, 9]. Для объемов такая функция расстояния обычно определяется значениями в узлах сетки GL и трилинейной интерполяцией, как и ~ f^l, а значения в узлах сетки определяются быстрым маршевым методом [23]. Градиент расстояния оказался лучшим выбором, чем градиент серого значения, но он также имеет некоторые потенциальные недостатки. Во-первых, он не определен должным образом повсюду, так как он прерывистый вдоль средней оси S^1, и поэтому оценка градиента крайне нестабильна вблизи медиальной оси. Однако, поскольку текущая сетка гарантированно близка к S^1, это не является проблемой для наших вычислений, но более серьезным недостатком является то, что градиент расстояния не способен перемещать вершины внутри вогнутой области изоповерхности, как показано на рис. 3.9

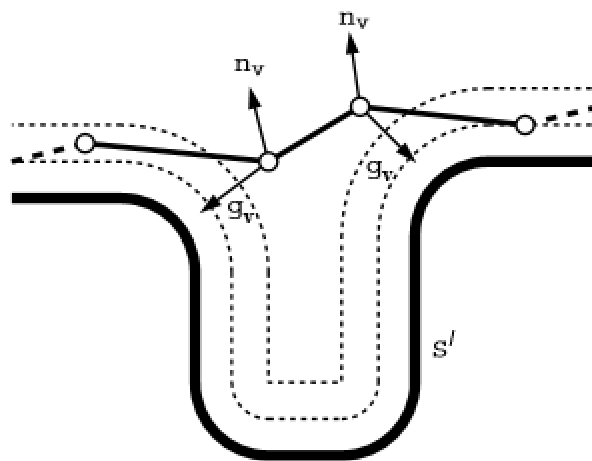

Рис. 3.9 Изолинии S^l с изоразмерными линиями (пунктир) и градиентами расстояния g_v и нормалями n в двух вершинах сетки

Поэтому мы перемещаем вершину v вдоль направления нормального вектора nv в точке v, которая может быть найдена либо путем усреднения

нормалей смежных с v треугольников, либо, как это было сделано, нормализацией вектора нормали кривизны [4].

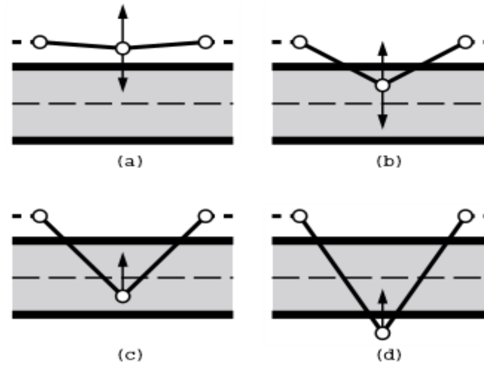

Рис. 3.10 Градиент расстояния и нормальный вектор в вершине в четырех различных ситуациях

Когда нормаль вычисляется, нам нужно определить, можно ли найти пересечение с изоповерхностью в положительном или отрицательном направлении. На рис. 3.10 показаны различные случаи, которые могут произойти, и мы используем функцию расстояния и ее градиент, чтобы распознать их. Обычно вершина находится вблизи изоповерхности, и мы можем просто использовать знак функции расстояния, чтобы определить, на какой стороне поверхности она лежит. Если dl (v)> 0, то вершина лежит «вне» и мы перемещаем ее в противоположном направлении ее нормального вектора (a). Если dl (v) <0, то он находится «внутри» и должен быть сдвинут вдоль нормального направления (b). Оба случая имеют то общее, что нормальный вектор nv и градиент расстояния gv указывают в противоположных направлениях. Также может случиться так, что nv и gv ориентированы аналогично (c) и (d), указывая, что вершина выходит за среднюю ось и приближается к «неправильной» изо-поверхности. Это может произойти, если шаг исправления вставляет новые вершины в сетку с сильно изогнутыми.

Области, где объект также очень тонкий, так что две части изоповерхности близки. В этом случае мы рассмотрим луч вдоль

положительного направления нормали и найдем первое (c) или второе (d) пересечение с изоповерхностью. Чтобы различать случаи (a, b) и (c, d), мы определяем знак скалярного произведения <nv | gv>. Хотя градиент расстояния не определен на медиальной оси, никаких проблем не будет, поскольку в обоих случаях (b) и (c) вершина будет перемещаться вдоль положительного направления нормали. Чтобы проверить, было ли найдено правильное пересечение в случае (d), мы снова используем знак скалярного произведения <nv | gv>.

3.2.2 Доказательство правильности алгоритма

Учитывая регулярную выборку сетки скалярного поля и изоповерхности, мы утверждаем, что наш алгоритм создает поверхность, которая аппроксимирует истинную изоповерхность для этой изовалы. Наша претензия состоит из двух частей. Во-первых, мы показываем, что наш алгоритм действительно создает поверхность, а не просто набор произвольно связанных симплексов. Более конкретно, он строит триангулированное (d-1) - многообразие с границей в Rd. Заметим, что действительная изоповерхность не обязательно должна быть многообразием, но во многих приложениях желательно, чтобы поверхность была многообразием. Второе утверждение состоит в том, что поверхность, построенная по нашему алгоритму, аппроксимирует точную изоповерхность от скалярной функции. Без условий на лежащее в основе скалярное поле невозможно восстановить точную изоповерхность из дискретной выборки функции или даже произвести поверхность, близкую к некоторой метрике, к точной изоповерхности. Точная изоповерхность должна пересекать ребра сетки с одним положительным и одним отрицательным концом, хотя она может пересекать другие. Мы утверждаем и доказываем, что наша приближенная поверхность пересекает ровно этот набор ребер сетки с одним положительным и одним отрицательным концом.

Даже в трех измерениях существует множество топологически различных поверхностей, пересекающих один и тот же набор ребер сетки. Без лежащей в основе скалярной функции невозможно узнать, какая поверхность гомотопна (непрерывно деформируема) до точной изоповерхности, хотя предсказания могут быть сделаны на основе некоторых упрощающих предположений [9]. Как и в случае алгоритма Modified Marching Cubes [8], мы не утверждаем, что наша изоповерхность гомотопна точной изоповерхности. Наш алгоритм возвращает набор симплексов, которые, как мы утверждаем, образуют поверхность. Чтобы доказать наше утверждение, сначала покажем, что эти симплексы правильно пересекаются на их гранях, образуя так называемый симплициальный комплекс. Затем, используя основные свойства симплициальных комплексов, мы покажем, что этот симплициальный комплекс образует поверхность. Чтобы точнее сформулировать наши результаты, приступим к некоторым определениям. Множество точек M в Rd является (d-1) -мерным многообразием с краем, если окрестность каждой точки в M гомеоморфна либо Rd-1, либо замкнутому полупространству Rd-1. Интуитивно многообразие с краем представляет собой множество точек, которые ведут себя локально как часть (d-1) -мерного евклидова пространства или граница (d-1) -мерного евклидова полупространства. Множество T симплексов определяет симплициальный комплекс, если непустое пересечение любых двух или более симплексов T является гранью каждого из этих симплексов. Например, непустое пересечение любых двух трахеров является либо (треугольной) гранью, либо ребром или вершиной двух тетраэдров, а непустое пересечение любых трех тетраэдров является ребром или вершиной всех трех.

3.3 Программные модули отображения и обработки информации

3.3.1 Проектирование системы визуализации

Визуализация единой сетки очень проста, хотя и несколько неэффективна по времени.

Проблемы с большими детализированными уровнями возникают из-за количества полигонов, с которыми приходится иметь дело. Рисование всех полигонов в каждом кадре неэффективно. Для увеличения скорости вы можете визуализировать только те полигоны, которые находятся в поле зрения, а чтобы игра работала еще быстрее, исключить сканирование каждого полигона сцены, определяющее видим ли он.

Решение заключается в разделении трехмерной модели (представляющей уровень) на небольшие фрагменты (называемые узлами, nodes), содержащие несколько полигонов. Затем узлы упорядочиваются в специальную структуру (дерево, tree) которую можно быстро просканировать для определения того, какие узлы видимы. Потом нужно визуализировать видимые узлы. Можно определить, какие узлы видимы, используя пирамиду видимого пространства. Теперь вместо того, чтобы сканировать тысячи полигонов, вы сканируете небольшой набор узлов, чтобы определить, что рисовать. Видите, насколько просто это улучшение процесса рисования? Созданный специально для этой книги, он может взять любую сетку и разделить ее на узлы, используемые для быстрой визуализации сеток (например, сеток, используемых в качестве уровней вашей игры). К разделению узлов вернемся чуть позже, а перед тем, как продолжить, необходимо прояснить несколько моментов. Узел представляет группу полигонов и в то же время представляет область трехмерного пространства. Каждый узел связан с другими отношениями родитель-потомок. Это означает, что узел может содержать другие узлы, и каждый последующий узел является частью, меньшей чем его родитель. Весь трехмерный мир в

целом считают корневым узлом (root node) — самым верхним узлом, с которым полигоны сгруппировать их; затем, начав с корневого узла, вы сможете быстро перебрать каждый узел дерева. Чтобы создать узлы и построить древовидную структуру вы проверяете каждый полигон сетки. Не беспокойтесь; это делается только один раз и влияние этого процесса на быстродействие можно не учитывать. Ваша цель — решить, как упорядочивать узлы в дереве. связаны все другие узлы.Вот трюк для узлов и деревьев: зная, какие содержатся в узле трехмерного пространства, вы можете Воксельная модель данных разбивает пространство на множествагеометрических элементов (называемых вокселями), смежных и не атрибутивные значения. По типу воксельных элементов, модель делится на Каждый полигон сетки заключается в прямоугольник. Прямоугольник определяет протяженность полигона по всем направлениям. Если ограничивающий прямоугольник полигона находится в узле трехмерного пространства (полностью или частично), значит полигон относится к узлу.

Полигон может относиться к нескольким узлам, поскольку протяженность перекрывающихся между собой, где каждому вокселю присваиваются полигона может распространяться на несколько узлов.

- модель 3D решетки с одинаковым размером кубических элементов, которая является расширением растровой 2D модели на 3D случай;
- модель октодерева с делением каждой стороны куба пополам, которая является расширением 2D модели квадродерева на случай 3D. Для улучшения эффективности, также рассматриваются точечные октодеревья, улучшенные линейные октодеревья, как например модель октодерева с множественным разложением на составные части;
- модель тетраэдрических элементов, также называемая моделью TEN (нерегулярная сеть тетраэдров), которая является расширением TIN 2D модели для пространства 3D;

- модель вертикальных треугольных призм, с нерегулярными треугольными призмами в качестве объемных элементов, сечения которых могут быть представлены на вертикальных разрезах между линиями сеток TIN;
- GTP модель (генерализованные треугольные призмы), которая составлена из модели треугольных призм. Но так как скважины часто отклоняются от вертикали, они не образуют стандартные треугольные призмы, а формируют так называемые генерализованные треугольные призмы. Генерализованная треугольная призма может быть преобразована в два тетраэдра.

Пространственное подразбиение является основой воксельной модели. Точки роста зерен путём трансформации 3D расстояний для генерации 3D полиэдров Делоне, а затем переключения к солидам на основе 3D полиэдров Вороного. Представили динамический метод, и исследовали построение 3D тетраэдров или 3D диаграмм Вороного напрямую из множества дискретных 3D точека так модель многоуровнего октодерева и обобщенную модель, состоящую из треугольных призм.

Преимуществом 3D моделей на базе вокселей являются:

- простая структура данных и простые операции, особенно для стандартной модели с кубическими вокселями;
- она может описывать не только границы твердотельных структур, но и неоднородные свойства поля внутри объектов с точностью до размеров (гранулярности) вокселей;
- некоторые пространственные операции, такие как перекрытие или буферизация выполняются достаточно просто.

Недостатками являются:

- количество данных возрастает в кубе при увеличении разрешения;
- модель не включает концепцию объекта, а объект представляет собой коллекцию вокселей, имеющих одинаковые свойства

- нет прямой концепции топологических связей, поэтому некоторые пространственные операции, такие как анализ связности, трудно реализуемы в воксельной модели.

3.3.2 Визуализатор Voxler

Моделирование данных происходит очень быстро. С помощью обширного инструментария 3D моделирований Voxler, можно легко визуализировать многокомпонентные данные для геологических и геофизических моделей шлейфов, загрязнения облака, точек лидара, модели скважин или рудных моделей депозитного тела.

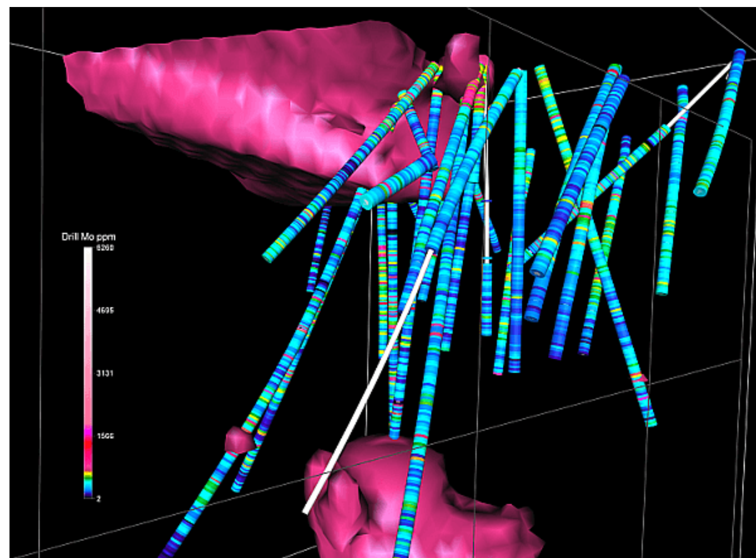

Рис. 3.12 Пример работы Voxler

Так же возможно проводить исследования глубин при помощи данных. Просмотр модели Voxler с любого угла, чтобы определять аномалии и выявлять закономерности и тенденции, как показано на рис. 3.12.

Voxler имеет улучшенное распознавание трехмерных данных. Преобразует в комплексные модели, которые упрощают процесс принятия решений.

Представленые данные с многочисленными опциями настройками Voxler позволяет контролировать каждый аспект модели вплоть до

мельчайших деталей, так чтобы можно было передать сложные модели в удобной для понимания форме.

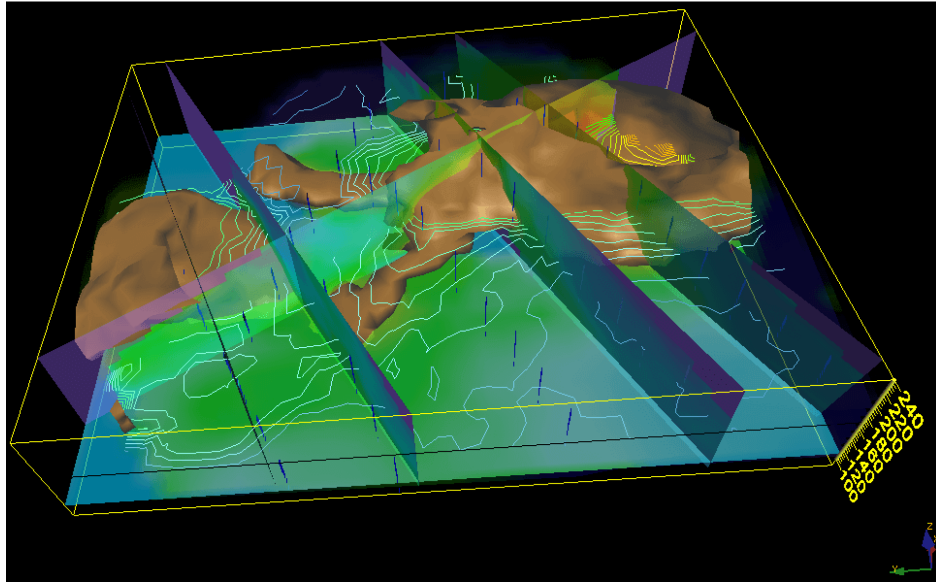

Рис. 3.13 Пример работы Voxler по данным

Если модель построенная верно и точно. Voxler будет производить однородную 3D решетку из регулярно или нерегулярно расположенных данных, как показано на рис. 3.12. Погружение в детали данных с метрических расчетов, включая порядковых статистик данных (минимум, нижний квартили, медиана, верхний квартиль, максимум, диапазон, средних частот, диапазон между квартиль), статистика момента (среднее, стандартное отклонение, дисперсия, коэффициент вариации , сумма), другие статистические данные (среднеквадратичное, медианное абсолютное отклонение), и статистика определения местоположения (подсчет, плотность, ближайшая расстояние, максимальное расстояние, среднее расстояние, среднее расстояние).

Voxler Интерполяция функции:

- Методы Гридинг: Inverse Расстояние и Локально полиномиальная
- Фильтр, исключить данные
- Выполнение математических операций на решетке
- Transform, слияние, ресамплинг решетку

- Извлечение подмножества или кусочек решетки

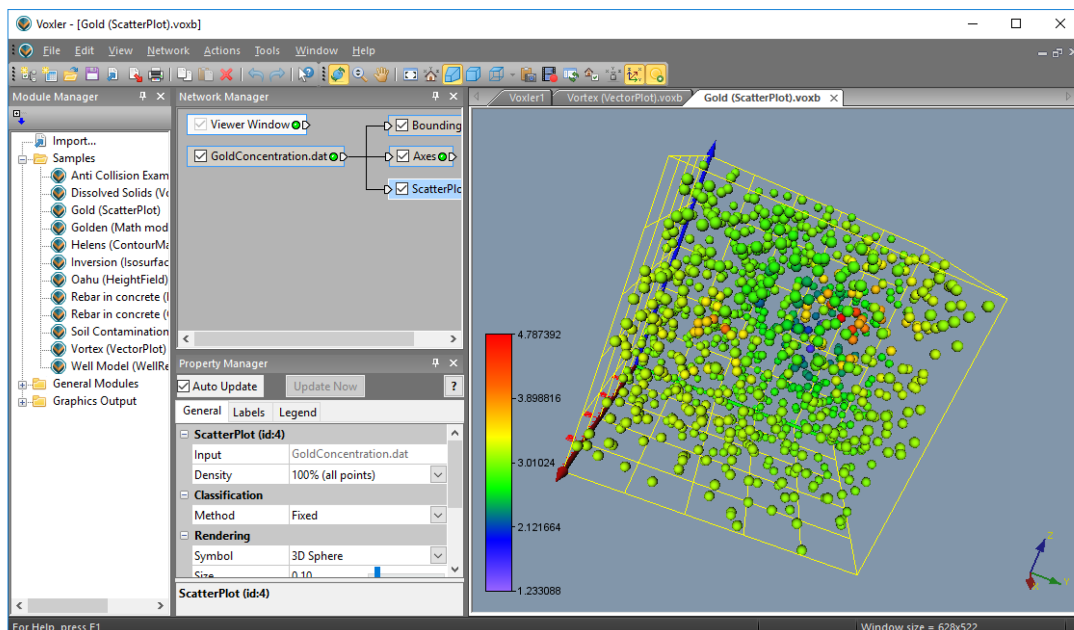

Рис. 3.13 Пример работы Voxler по данным построение Grid3D

Данные является основой для успешного моделирования проекта, как показано на рис. 3.13. С выпуском последней версией Voxler 3D отображения визуализации программного обеспечения включает в себя функции: новое окно рабочего листа, которое позволяет редактировать в реальном времени импортированные данные. Также можно использовать Voxler как ценный инструмент контроля качества данных для коррекции данных облака и сквозных данных.

4. АПРОБАЦИЯ И ВНЕДРЕНИЕ АВТОМАТИЗИРОВАННОЙ СИСТЕМЫ ПОСТРОЕНИЯ ИЗОПОВЕРХНОСТЕЙ

Для тестирования приложения используется 8-ядерный процессор AMD FX-8350 4.5 МГц и видеокарта ATI Radeon 7870 2Гб, RAM 8Гб.

После запуска приложения, выбора модели. Исходная модель в виде отображается следующим образом (см. рис. 4.1):

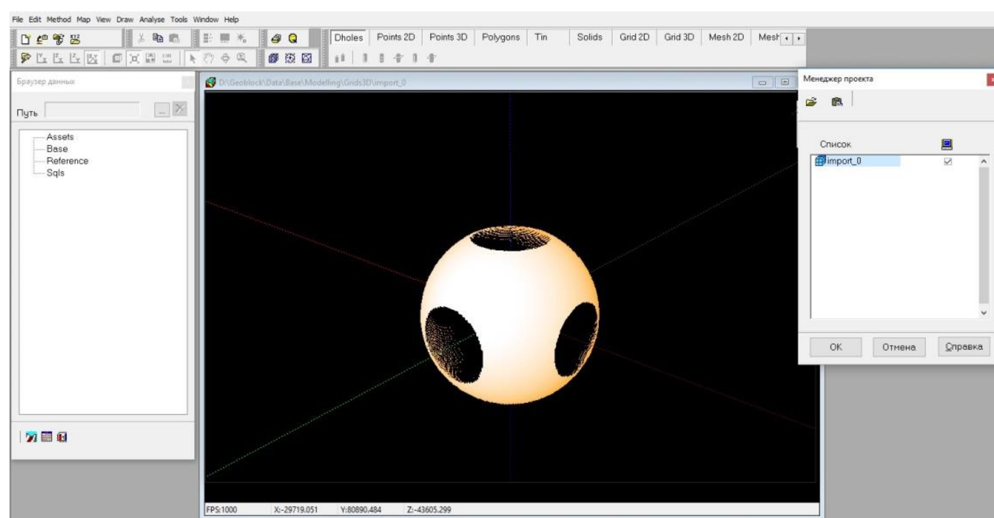

Рис. 4.1 Модель в исходном виде отображается следующим образом

Так же мы можем взаимодействовать с моделью, осматривая и приближая её со всех возможных ракурсов.

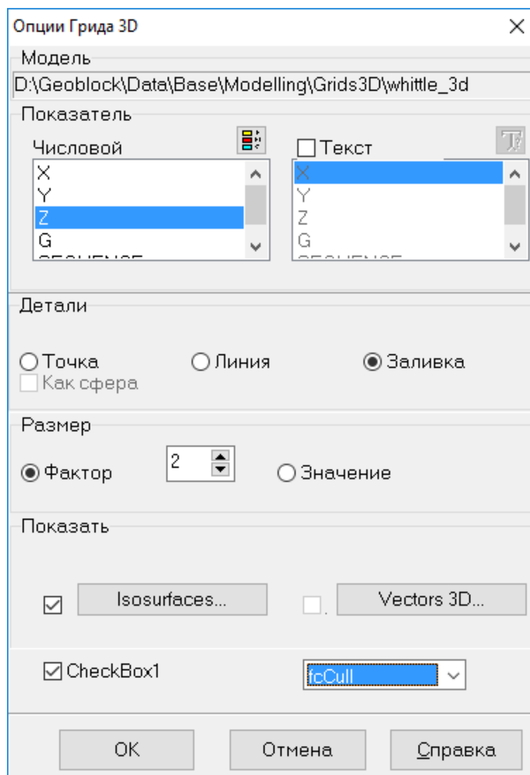

Рис 4.2 Опции для отображения изоповерхностей

Опции параметров грид 3D на которой можно выбрать отображение точек, линий или полной заливки выбранной модели как показано на рис. 4.2. После того, как нужные параметры отображения будут выбраны. Модель будет выведена на экран и визуализирована в нужной форме.

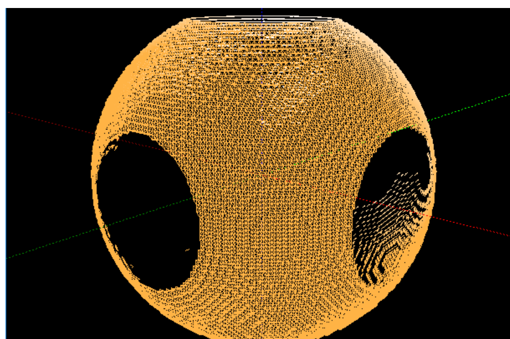

Рис. 4.3 Пример изоповерхностей без заливки

Визуализация модели с отключенными изоповерхностями изображена на рис. 4.3 отображается в сетки с ребрами.

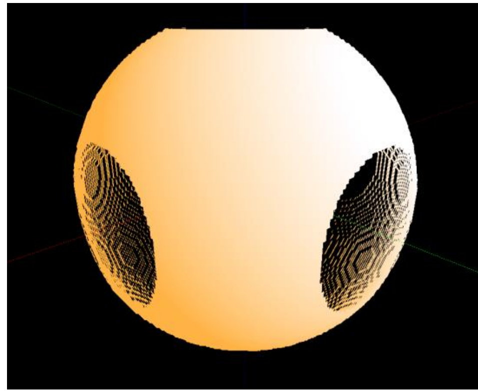

Рис. 4.4 Пример работы ChekBox заливки изоповерхностей

На рис. 4.4 отображены изолинии с использованием переключателя ChekBox, что позволяет увидеть нашу 3D модель не влияя на производительность ПК.

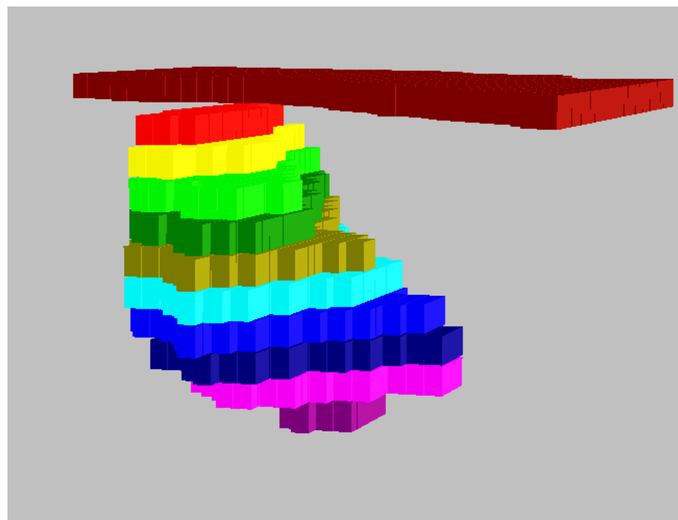

Рис 4.5 Пример тестовой модели.

Доработаный модуль позволяет отображать разной сложности модели, как показано на рис. 4.5 при этом отображение происходит без задержек.

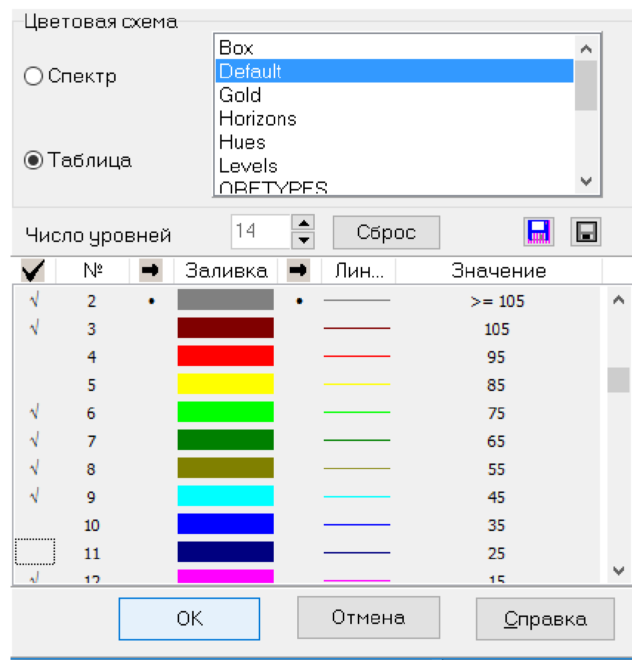

Рис. 4.5 Диалоговое окно уровня заливки

Также есть диалоговое окно выбора уровня заливки показанно на рис. 4.5. Галочкой отмечаем нужные уровни для отображения и нажимаем кнопку «ОК». Далее в окне проекта отобразятся оставшиеся уровни изображенные на рис. 4.6.

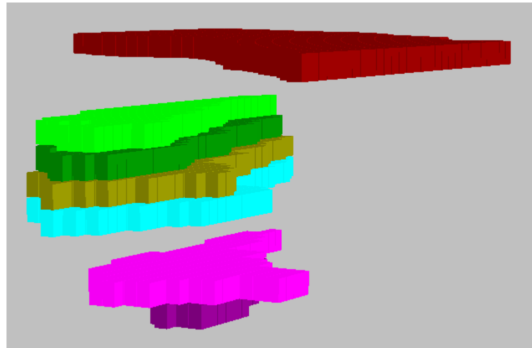

Рис 4.6 Пример отображения выбраных уровней заливки

Полученую модель сохраняем в формате Mesh2D для быстрого вызоваю. После просмотра результата можно сделать вывод, что приложение для визуализации изоповерхности выполняет все свои функции корректно.

ЗАКЛЮЧЕНИЕ

В данном проекте был разработан GPU-совместимый модуль для визуализации изоповерхностей. Этот модуль обеспечивают эффективный и удобный способ хранения данных, а также защищённую от искажений сетку поверхности. Это могут быть данные о цвете, как при применении сетки палитры, или данные для динамического моделирования текстур, что достаточно часто применяется для демонстрации динамических водных поверхностей. Данная визуализация может быть эффективно и быстро осуществлена на современном оборудовании. Для ускорения быстродействия визуализации было предоставлено решение для фильтрации текстур, чтобы избежать лишнего наложения. Тем не менее, так 3D текстуры весьма предпочтительны в некоторых ситуациях и было показано, как текстура может быть динамически преобразована в 3D текстуру без артефактов.

Объемная графика имеет преимущества по сравнению с полигональной, будучи независимой с любой точки зрения и нечувствительной к сцене и сложности объекта. Она подходит для представления проб или смоделированных данных и их перемешивание с геометрическими объектами, и поддерживает визуализацию внутренних структур объектов. Проблемы, связанные с представлением объема буфера, такие как объем памяти, процессорное время, сглаживание, а также отсутствие геометрического представления, появились в качестве альтернативы технологии векторной графики и могут быть решены в подобных отношениях. До настоящего времени прогресс компьютерной техники и систем памяти, в сочетании с желанием раскрыть внутреннюю структуру объемных объектов состоит в том, что визуализация объемов и объемная графика могут перерасти в основные тенденции в области компьютерной визуализации данных.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Н.Н.Калиткин, И.П.Пошивайло. О вычислении простых и кратных корней нелинейного уравнения // Матем. моделирование. 2008, т.20, №7, с.57-64.
2. Постников М.М. Устойчивые многочлены. – М.: Наука, 1981, 176 с.
3. Маккей А. «Введение в платформу .NET 4.0 с Visual Studio 2010». Пер. с англ. — М.: Издательский дом "Вильямс", 2010. — 505 с.: ил.
4. В.В.Лабор "Си Шарп: Создание приложений для Windows". Изд. «Харвест», 2003 г.
5. Карли Ватсон "C#". Изд. «Лори». 2003 г.
6. Рихтер Дж. CLR Via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# [текст]: мастер-класс / перевод с английского М. издательство «Русская Редакция» СПб.: Питер, 2007 – 656 с.
7. Симон Робинсон "C# для профессионалов". В двух томах. Том 1. Изд. "Лори". 2003 г.
8. Симон Робинсон "C# для профессионалов". В двух томах. Том 2. Изд. "Лори". 2003 г.
9. Герберт Шилдт "Полный справочник по C#". Изд. "Вильямс". 2004 г.
10. Джесс Либети "Создание .NET приложений. Программирование на C#". 2-е издание. Изд. "Символ". 2003 г.
11. Д.Сеппа "Программирование на Microsoft ADO.NET 2.0. Мастер класс". Пер. с англ. Изд. "Питер". 2007 г.
12. Вильямс А. Системное программирование в Windows 2000 /Пер. с англ. П. Анджан.-СПб.- М.- Харьков - Минск: Питер, 2001.-621 с.: ил. + CD-ROM.
13. В.П.Румянцев. Азбука программирования в Win 32 API. – 3-е изд.: - Москва, «Горячая линия - телеком», 2005.

14. Маркушевич А. И. Введение в теорию аналитических функций: Учеб. пособие для физ.-мат. фак. пед. ин-тов. – М.: Просвещение, 1977. – 320 с., ил.
15. Маркушевич А. И. Теория аналитических функций. – 2-е изд., испр. и доп.: В 2-х т. Т. 1. – М.: Наука, 1967. – 486 с., черт.
16. Энциклопедический словарь юного математика: Для сред. и ст. шк. возраста /Сост. Савин А. П. – 2-е изд., испр. и доп. – М.: Педагогика, 1989. – 349 с., ил.
17. Энциклопедия элементарной математики. Кн. 4. Геометрия. – М.: Физматгиз, 1963. – 568 с., ил.
18. Энциклопедия элементарной математики. Кн. 5. Геометрия. – М.: Наука, 1966. – 624 с., ил.
19. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В. Основы криптографии. Учебное пособие. М., Гелиос АРВ, 2005.
20. Клайн К., Клайн Д., Хант Бр. XML. Справочник. 2-е издание / Пер. с англ. – М.: КУДИЦ-ОБРАЗ, 2006 – 832 с.
21. Конноли Т., Бегг К. «Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3–е издание». Пер. с англ. — М.: Издательский дом "Вильямс", 2003. — 1440 с.: ил.
22. Маклаков С. В. «BPwin и ERwin: CASE-средства для разработки информационных систем». — Диалог–МИФИ, 2001. — 256 с.
23. А. Н. Тихонов, А. Б. Васильева, А. Г. Свешников. Дифференциальные уравнения. — Наука, ФИЗМАТЛИТ, 1998. — 232 с.
24. В. Н. Масленникова. Дифференциальные уравнения в частных производных. — Издательство Российского Университета дружбы народов, 1997. — 448 с.
25. Martz, Paul. 2006. OpenGL(R) Distilled. Addison-Wesley Professional.

ПРИЛОЖЕНИЕ

```
unit GLIsosurface;

// uncomment next line to memorize vertex Density value to further use
// (i.e. mesh color generation)
{.$Define UseDensity}

interface

uses
  GLVectorGeometry,
  GLMesh,
  GLVectorFileObjects,
  GLVectorTypes,
  GLTypes;


const
  ALLOC_SIZE = 65536;

type
  TSingle3DArray = array of array of array of Single;
  TVertexArray = array of TVector3f;
  TIntegerArray = array of Integer;

  TGLMarchingCube = class(TObject)
  private
    FIsoValue: TGLScalarValue;
    // sliceSize:Longword;

    PVertsX: PIntegerArray;
    PVertsY: PIntegerArray;
    PVertsZ: PIntegerArray;

    _Nverts: Integer;
    _Ntrigs: Integer;
    _Sverts: Integer;
    _Strigs: Integer;

    PVertices: PGLVertexArray;
    PTriangles: PGLTriangleArray;

    _i, _j, _k: Longword;

    _Cube: array [0 .. 7] of TGLVoxel;
    _lut_entry: Byte;
    // _case:Byte;
    // _config:Byte;
```

```
  // _subconfig:Byte;

  procedure Init_temps;
  procedure Init_all;
  procedure Init_space;

  procedure Clean_temps;
  procedure Clean_all(keepFacets: Boolean = False);
  procedure Clean_space;
  procedure Test_vertex_addiction;

protected
  FOriginalMC: Boolean; // now only original MC is implemented
  FSizeX: Integer;
  FSizeY: Integer;
  FSizeZ: Integer;
  FxMin: Single;
  FxMax: Single;
  FyMin: Single;
  FyMax: Single;
  FzMin: Single;
  FzMax: Single;
  FStepX: Single;
  FStepY: Single;
  FStepZ: Single;

  VoxelData: PGLVoxelData;

  procedure Process_cube;
  { function test_face(face:byte):Boolean;
    function test_interior(s:Byte):boolean }

  procedure Compute_Intersection_Points;
  procedure Add_Triangle(trig: array of Integer; N: Byte; v12: Integer = -1);

  function Add_x_vertex: Integer;
  function Add_y_vertex: Integer;
  function Add_z_vertex: Integer;
  function Add_c_vertex: Integer;

  function Get_x_grad(i, j, k: Integer): Single;
  function Get_y_grad(i, j, k: Integer): Single;
  function Get_z_grad(i, j, k: Integer): Single;

  function Get_x_vert(i, j, k: Integer): Integer;
  function Get_y_vert(i, j, k: Integer): Integer;
  function Get_z_vert(i, j, k: Integer): Integer;

  procedure Set_x_vert(a_val, i, j, k: Integer);
  procedure Set_y_vert(a_val, i, j, k: Integer);
  procedure Set_z_vert(a_val, i, j, k: Integer);
```

```
  function GetVoxelValue(i, j, k: Integer): TGLScalarValue;
  procedure SetVoxelValue(i, j, k: Integer; HfValue: TGLScalarValue);

  function GetVoxelData(i, j, k: Integer): TGLVoxel;
  function Voxel(i, j, k: Integer): PGLVoxel;

  function Calc_u(v1, v2: Single): Single; virtual;
public
  ScalarField: TGLScalarField;

  constructor Create; overload; virtual;
  constructor Create(SizeX, SizeY, SizeZ: Integer;
    AIsoValue: TGLScalarValue = 0.0; xMin: Single = -0.5; xMax: Single = 0.5;
    yMin: Single = -0.5; yMax: Single = 0.5; zMin: Single = -0.5;
    zMax: Single = 0.5); overload; virtual;

  procedure ReDim(ASizeX, ASizeY, ASizeZ: Integer;
    xMin, xMax, yMin, yMax, zMin, zMax: Single); virtual;

  destructor Destroy; override;

  procedure Run; overload;
  procedure Run(IsoValue: TGLScalarValue); overload;

  function Internal(AValue: TGLScalarValue): Boolean; virtual;

  procedure FillVoxelData; overload; virtual;
  procedure FillVoxelData(AIsoValue: TGLScalarValue;
    AScalarField: TGLScalarField = nil); overload; virtual;
  procedure FillVoxelData(AIsoValue: TGLScalarValue;
    AScalarField: TGLScalarFieldInt); overload; virtual;

  procedure CalcVertices(Vertices: TGLVertexList; Alpha: Single = 1);
  procedure CalcMeshObject(AMeshObject: TMeshObject; Alpha: Single = 1);

  property IsoValue: TGLScalarValue read FIsoValue write FIsoValue;
  // TODO SetIsoValue to Run
end;

// TIsoSurfaceExtractor
//
{ 3D isosurface extractor class. This class allows to calculate and exctract
  isosurfaces from scalar field voxel models using a given isovalue.
}
TIsoSurfaceExtractor = class(TObject)
private
  Data: TSingle3DArray;
  Dimensions: array ['x' .. 'z'] of Integer;

  function BuildIndex(var ADatavals: array of Single; Isovalue: Single): word;
  function Interpolate(V0, V1: TAffineVector;
    var Val0, Val1, Isovalue: Single; isPolished: boolean): TVertex;
```

```
  public
    constructor Create(); overload;
    constructor Create(Xdim, Ydim, Zdim: Integer;
      var AData: TSingle3DArray); overload;
    destructor Destroy();

    procedure AssignData(Xdim, Ydim, Zdim: Integer; var AData: TSingle3DArray);

    procedure MarchingCubes(Isovalue: Single; out Vertices: TVertexArray;
      out Triangles: TIntegerArray; isPolished: boolean);
    procedure MarchingTetrahedra(Isovalue: Single; out Vertices: TVertexArray;
      out Triangles: TIntegerArray; isPolished: boolean);
  end;

// Sphere surface
function SFSphere(X, Y, Z: Single): TGLScalarValue;
// Minkowski space (http://mathworld.wolfram.com)
function SFMinkowski(X, Y, Z: Single): TGLScalarValue;
// Klein Bottle (http://mathworld.wolfram.com)
function SFKleinBottle(X, Y, Z: Single): TGLScalarValue;
// Chmutov-surface-1 (http://mathworld.wolfram.com)
function SFChmutov1(X, Y, Z: Single): TGLScalarValue;
// Chmutov-surface-2 (http://mathworld.wolfram.com)
function SFChmutov2(X, Y, Z: Single): TGLScalarValue;
// Toroidal surface (phantasy!)
function SFToroidal(X, Y, Z: Single): TGLScalarValue;
// Double torus Surface (phantasy!)
function SFDoubleTorus(X, Y, Z: Single): TGLScalarValue;

const
  DemoScalarField: array [0 .. 6] of
  record
    // xMin, xMax, yMin, yMax, zMin, zMax:Single; // default -0.5..0.5
    ScalarField: TGLScalarField;
    IsoValue: TGLScalarValue
  end = ((ScalarField: SFSphere; IsoValue: 0.3), (ScalarField: SFMinkowski;
  IsoValue: 0.0), (ScalarField: SFKleinBottle; IsoValue: 0.0),
  (ScalarField: SFChmutov1; IsoValue: 3.0), (ScalarField: SFChmutov2;
  IsoValue: 3.0), (ScalarField: SFToroidal; IsoValue: 3.0),
  (ScalarField: SFDoubleTorus; IsoValue: 0.015));

  // ------------------------------------------------------------------------
  // ------------------------------------------------------------------------
  // ------------------------------------------------------------------------
implementation

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

const
```

```
// Classic Cases for Marching Cube TriTable
//
(*
      4----4------5
     /|          /|
    7 |         5 |
   /  |        /  |
  7-----6----6    |
  |   8       |   9
  |   |       |   |
  |   0----0--|---1
  11 /        10 /
  | 3         | 1
  |/          |/
  3-----2-----2
*)


// Marching Cube EdgeTable
//
const

MC_EDGETABLE: array [0 .. 11, 0 .. 1] of Integer = ((0, 1), (1, 2), (2, 3),
  (3, 0), (4, 5), (5, 6), (6, 7), (7, 4), (0, 4), (1, 5), (2, 6), (3, 7));

// Marching Tetrahedra TriTable
//
(*
      + 0
       /|\
      / | \
     /  |  0
    3   |   \
   /    2    \
  /     |     \
 +----4--------+ 1
3 \     |     /
   \    |    /
    5   |   1
     \  |  /
      \ | /
       \|/
      + 2
*)
MT_TRITABLE: array [0 .. 15, 0 .. 6] of Integer =
  ((-1, -1, -1, -1, -1, -1, -1), (2, 3, 0, -1, -1, -1, -1),
  (4, 1, 0, -1, -1, -1, -1), (2, 4, 1, 3, 4, 2, -1),
  (5, 2, 1, -1, -1, -1, -1), (5, 3, 0, 1, 5, 0, -1), (5, 2, 0, 4, 5, 0, -1),
  (3, 4, 5, -1, -1, -1, -1), (5, 4, 3, -1, -1, -1, -1),
  (0, 5, 4, 0, 2, 5, -1), (0, 5, 1, 0, 3, 5, -1), (1, 2, 5, -1, -1, -1, -1),
  (2, 4, 3, 1, 4, 2, -1), (0, 1, 4, -1, -1, -1, -1),
  (0, 3, 2, -1, -1, -1, -1), (-1, -1, -1, -1, -1, -1, -1));
```

```
  // Marching Tetrahedra EdgeTable
  //
  MT_EDGETABLE: array [0 .. 5, 0 .. 1] of Integer = ((0, 1), (1, 2), (2, 0),
    (0, 3), (1, 3), (2, 3));

  // Marching Tetrahedra CubeSplit
  //
  MT_CUBESPLIT: array [0 .. 5, 0 .. 3] of Integer = ((0, 5, 1, 6), (0, 1, 2, 6),
    (0, 2, 3, 6), (0, 3, 7, 6), (0, 7, 4, 6), (0, 4, 5, 6));

// Test surface functions
//

function SFSphere(X, Y, Z: Single): TGLScalarValue;
begin
  Result := sqr(X) + sqr(Y) + sqr(Z)
end;

function SFToroidal(X, Y, Z: Single): TGLScalarValue;
const
  FScale = 7;
  a = 2.5;
begin
  X := FScale * X;
  Y := FScale * Y;
  Z := FScale * Z;
  Result := (sqr(sqrt(sqr(X) + sqr(Y)) - a) + sqr(Z)) *
    (sqr(sqrt(sqr(Y) + sqr(Z)) - a) + sqr(X)) *
    (sqr(sqrt(sqr(Z) + sqr(X)) - a) + sqr(Y));
end;

function SFDoubleTorus(X, Y, Z: Single): TGLScalarValue;
const
  FScale = 2.25;
begin
  X := FScale * X;
  Y := FScale * Y;
  Z := FScale * Z;
  Result := PowerInteger(X, 8) + PowerInteger(X, 4) - 2 * PowerInteger(X, 6) - 2
    * sqr(X) * sqr(Y) + 2 * PowerInteger(X, 4) * sqr(Y) +
    PowerInteger(Y, 4) + sqr(Z)
end;

function SFChmutov1(X, Y, Z: Single): TGLScalarValue;
const
  FScale = 2.5;
begin
  X := FScale * X;
  Y := FScale * Y;
  Z := FScale * Z;
  Result := 8 * (sqr(X) + sqr(Y) + sqr(Z)) - 8 *
```

```
  (PowerInteger(X, 4) + PowerInteger(Y, 4) + PowerInteger(Z, 4));
end;

function SFChmutov2(X, Y, Z: Single): TGLScalarValue;
const
 FScale = 2.5;
begin
 X := FScale * X;
 Y := FScale * Y;
 Z := FScale * Z;
 Result := 2 * (sqr(X) * sqr(3 - 4 * sqr(X)) + sqr(Y) * sqr(3 - 4 * sqr(Y)) +
  sqr(Z) * sqr(3 - 4 * sqr(Z)));
end;

function SFKleinBottle(X, Y, Z: Single): TGLScalarValue;
const
 FScale = 7.5;
begin
 X := FScale * X;
 Y := FScale * Y;
 Z := FScale * Z;
 Result := (sqr(X) + sqr(Y) + sqr(Z) + 2 * Y - 1) *
  (sqr(sqr(X) + sqr(Y) + sqr(Z) - 2 * Y - 1) - 8 * sqr(Z)) + 16 * X * Z *
  (sqr(X) + sqr(Y) + sqr(Z) - 2 * Y - 1);
end;

function SFMinkowski(X, Y, Z: Single): TGLScalarValue;
const
 FScale = 7;
begin
 X := FScale * X;
 Y := FScale * Y;
 Z := FScale * Z;
 Result := (sqr(X) - sqr(Y) - sqr(Z) - 2) * (sqr(X) - sqr(Y) - sqr(Z) + 2) *
  (sqr(X) - sqr(Y) - sqr(Z) - 4) * (sqr(X) - sqr(Y) - sqr(Z) + 4) *
  (sqr(X) - sqr(Y) - sqr(Z));
end;

 { ------------------------------------------------------------------------
  Class IsoSurfaceExtractor
  Purpose: Extract an Isosurface from volume dataset for given Isovalue
  ------------------------------------------------------------------------ }

 // Build Index depending on whether the edges are outside or inside the surface
 //
function TIsoSurfaceExtractor.BuildIndex(var ADatavals: array of Single;
 Isovalue: Single): word;
var
 i: Integer;
 val: word;
begin
 val := 1;
```

```
  Result := 0;
  for i := 1 to Length(ADatavals) do
  begin
    if ADatavals[i - 1] <= Isovalue then // Edge inside surface
      Result := Result + val;
    val := val * 2;
  end;
end;

// Compute intersection point of edge and surface by linear interpolation
//
function InterpolateRugged(V0, V1: TAffineVector;
  var Val0, Val1, Isovalue: Single): TVertex;
var
  Diff: Single;
  i: Integer;
begin
  if Val0 <= Isovalue then
  begin
    Diff := Val0 / (Val0 + Val1);
    for i := 0 to 2 do
      Result.V[i] := V0.V[i] + Diff * (V1.V[i] - V0.V[i]);
  end
  else
  begin
    Diff := Val1 / (Val0 + Val1);
    for i := 0 to 2 do
      Result.V[i] := V1.V[i] + Diff * (V0.V[i] - V1.V[i]);
  end;
end;

function InterpolatePolished(V0, V1: TAffineVector;
  var Val0, Val1, Isovalue: Single): TVertex;
var
  w0, w1: Single;
begin
  if (Val0 = Val1) then
    w1 := 0.5
  else
    w1 := (Isovalue - Val0) / (Val1 - Val0);
  w0 := 1.0 - w1;
  Result.X := w0 * V0.X + w1 * V1.X;
  Result.Y := w0 * V0.Y + w1 * V1.Y;
  Result.Z := w0 * V0.Z + w1 * V1.Z;
end;

function TIsoSurfaceExtractor.Interpolate(V0, V1: TAffineVector;
  var Val0, Val1, Isovalue: Single; isPolished: boolean): TVertex;
begin
  if isPolished then
    Result := InterpolatePolished(V0, V1, Val0, Val1, Isovalue)
  else
```

```
    Result := InterpolateRugged(V0, V1, Val0, Val1, Isovalue)
  end;

  // Launch Marching Tetrahedra
  //
  procedure TIsoSurfaceExtractor.MarchingTetrahedra(Isovalue: Single;
    out Vertices: TVertexArray; out Triangles: TIntegerArray; isPolished: boolean);
  var
    i, j, k: Integer;
    index: word;
    CubeVertices: array of TAffineVector;
    Tetrahedron: array [0 .. 3] of TAffineVector;
    DataTetra: array [0 .. 3] of Single;

    // Add Triangle to List
    procedure AppendTri();
    var
      edge: byte;
      Ver1, Ver2, Ver3: TVertex;
      VListlength: Integer;
      TListlength: Integer;
    begin
      edge := 0;
      while MT_TRITABLE[index, edge] <> -1 do
      begin
        Ver1 := Interpolate(Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge],
0]
          ], Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge], 1]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge], 0]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge], 1]], Isovalue,
isPolished);
        Ver2 := Interpolate(Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge +
1],
          0]], Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge + 1], 1]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge + 1], 0]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge + 1], 1]], Isovalue,
isPolished);
        Ver3 := Interpolate(Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge +
2],
          0]], Tetrahedron[MT_EDGETABLE[MT_TRITABLE[index, edge + 2], 1]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge + 2], 0]],
          DataTetra[MT_EDGETABLE[MT_TRITABLE[index, edge + 2], 1]], Isovalue,
isPolished);
        VListlength := Length(Vertices) + 3;
        TListlength := Length(Triangles) + 3;
        SetLength(Vertices, VListlength);
        SetLength(Triangles, TListlength);
        Vertices[VListlength - 3] := Ver1;
        Vertices[VListlength - 2] := Ver2;
        Vertices[VListlength - 1] := Ver3;
        Triangles[TListlength - 3] := VListlength - 3;
        Triangles[TListlength - 2] := VListlength - 2;
```

```
    Triangles[TListlength - 1] := VListlength - 1;
    edge := edge + 3;
  end;
 end;

// Split Cube in 6 Tetrahedrons and process each tetrahedron
//
 procedure SplitCube();
 var
  i, j: Integer;
 begin
  for i := 0 to 5 do
  begin
   for j := 0 to 3 do
   begin
    Tetrahedron[j] := CubeVertices[MT_CUBESPLIT[i, j]];
    DataTetra[j] := Data[Trunc(Tetrahedron[j].X),
     Trunc(Tetrahedron[j].Y), Trunc(Tetrahedron[j].Z)];
   end;
   index := BuildIndex(DataTetra, Isovalue);
   AppendTri();
  end;
 end;

begin
(*
   1----2
  /|   /|
  0----3 |
  | 5--|-6
  |/   |/
  4----7
*)
 SetLength(CubeVertices, 8);
 for k := 0 to Dimensions['z'] - 2 do
 begin
  for j := 0 to Dimensions['y'] - 2 do
  begin
   for i := 0 to Dimensions['x'] - 2 do
   begin
    CubeVertices[0] := AffineVectorMake(i, j, k);
    CubeVertices[1] := AffineVectorMake(i, j, k + 1);
    CubeVertices[2] := AffineVectorMake(i + 1, j, k + 1);
    CubeVertices[3] := AffineVectorMake(i + 1, j, k);
    CubeVertices[4] := AffineVectorMake(i, j + 1, k);
    CubeVertices[5] := AffineVectorMake(i, j + 1, k + 1);
    CubeVertices[6] := AffineVectorMake(i + 1, j + 1, k + 1);
    CubeVertices[7] := AffineVectorMake(i + 1, j + 1, k);

    SplitCube();
   end; // for k
  end; // for j
```

```
  end; // for i
end; // ccMT

constructor TIsoSurfaceExtractor.Create;
begin
  inherited;
end;

constructor TIsoSurfaceExtractor.Create(Xdim, Ydim, Zdim: Integer;
  var AData: TSingle3DArray);
begin
  Create();
  AssignData(Xdim, Ydim, Zdim, AData);
end;

destructor TIsoSurfaceExtractor.Destroy;
begin
  inherited;
end;

procedure TIsoSurfaceExtractor.AssignData(Xdim, Ydim, Zdim: Integer;
  var AData: TSingle3DArray);
begin
  Dimensions['x'] := Xdim;
  Dimensions['y'] := Ydim;
  Dimensions['z'] := Zdim;

  Data := AData;
end;

// Launch Marching Cubes
//
procedure TIsoSurfaceExtractor.MarchingCubes(Isovalue: Single;
  out Vertices: TVertexArray; out Triangles: TIntegerArray; isPolished: boolean);
var
  i, j, k: Integer;
  index: word;
  CubeVertices: array [0 .. 7] of TAffineVector;
  CubeData: array [0 .. 7] of Single;

  procedure AppendTri();
  var
    edge: byte;
    Ver1, Ver2, Ver3: TVertex;
    VListlength: Integer;
    TListlength: Integer;
  begin
    edge := 0;
    while MC_TRITABLE[index, edge] <> -1 do
    begin
      Ver1 := Interpolate(CubeVertices[MC_EDGETABLE[MC_TRITABLE[index, edge],
0]
```

```
          ], CubeVertices[MC_EDGETABLE[MC_TRITABLE[index, edge], 1]],
          CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge], 0]],
          CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge], 1]], Isovalue,
isPolished);
         Ver2 := Interpolate(CubeVertices[MC_EDGETABLE[MC_TRITABLE[index,
          edge + 1], 0]], CubeVertices[MC_EDGETABLE[MC_TRITABLE[index, edge + 1],
          1]], CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge + 1], 0]],
          CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge + 1], 1]], Isovalue,
isPolished);
         Ver3 := Interpolate(CubeVertices[MC_EDGETABLE[MC_TRITABLE[index,
          edge + 2], 0]], CubeVertices[MC_EDGETABLE[MC_TRITABLE[index, edge + 2],
          1]], CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge + 2], 0]],
          CubeData[MC_EDGETABLE[MC_TRITABLE[index, edge + 2], 1]], Isovalue,
isPolished);
         VListlength := Length(Vertices) + 3;
         TListlength := Length(Triangles) + 3;
         SetLength(Vertices, VListlength);
         SetLength(Triangles, TListlength);
         Vertices[VListlength - 3] := Ver1;
         Vertices[VListlength - 2] := Ver2;
         Vertices[VListlength - 1] := Ver3;
         Triangles[TListlength - 3] := VListlength - 3;
         Triangles[TListlength - 2] := VListlength - 2;
         Triangles[TListlength - 1] := VListlength - 1;
         edge := edge + 3;
        end;
      end;

    begin
      (*
         7----6
        /|   /|
       3----2 |
       | 4--|-5
       |/   |/
       0----1
      *)
      for i := 0 to Dimensions['x'] - 2 do
      begin
        for j := 1 to Dimensions['y'] - 1 do
        begin
          for k := 0 to Dimensions['z'] - 2 do
          begin
            CubeVertices[0] := AffineVectorMake(i, j, k);
            CubeVertices[1] := AffineVectorMake(i + 1, j, k);
            CubeVertices[2] := AffineVectorMake(i + 1, j - 1, k);
            CubeVertices[3] := AffineVectorMake(i, j - 1, k);
            CubeVertices[4] := AffineVectorMake(i, j, k + 1);
            CubeVertices[5] := AffineVectorMake(i + 1, j, k + 1);
            CubeVertices[6] := AffineVectorMake(i + 1, j - 1, k + 1);
            CubeVertices[7] := AffineVectorMake(i, j - 1, k + 1);
            CubeData[0] := Data[i, j, k];
```

```
        CubeData[1] := Data[i + 1, j, k];
        CubeData[2] := Data[i + 1, j - 1, k];
        CubeData[3] := Data[i, j - 1, k];
        CubeData[4] := Data[i, j, k + 1];
        CubeData[5] := Data[i + 1, j, k + 1];
        CubeData[6] := Data[i + 1, j - 1, k + 1];
        CubeData[7] := Data[i, j - 1, k + 1];

        Index := BuildIndex(CubeData, Isovalue);
        AppendTri();
      end; // for k
    end; // for j
  end; // for i
end;


// TMarchingCube
//

function TGLMarchingCube.add_c_vertex: Integer;
var
  u: Single;
  vid: Integer;
  procedure VertexAdd(iv: Integer);
  begin
    with PVertices^[_Nverts] do
    begin
      u := u + 1;
      P := VectorAdd(P, PVertices[iv].P);
      N := VectorAdd(N, PVertices[iv].N);
{$IFDEF UseDensity}
      Density := Density + PVertices[iv].Density;
{$ENDIF}
    end
  end;

begin
  test_vertex_addiction;

  u := 0;
  with PVertices^[_Nverts] do
  begin
    P := NullVector;
    N := NullVector;
{$IFDEF UseDensity}
    Density := 0;
{$ENDIF}
  end;
  Inc(_Nverts);

  // Computes the average of the intersection points of the cube
  vid := Get_x_vert(_i, _j, _k);
```

```
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_y_vert(_i + 1, _j, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_x_vert(_i, _j + 1, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_y_vert(_i, _j, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_x_vert(_i, _j, _k + 1);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_y_vert(_i + 1, _j, _k + 1);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_x_vert(_i, _j + 1, _k + 1);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_y_vert(_i, _j, _k + 1);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_z_vert(_i, _j, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_z_vert(_i + 1, _j, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_z_vert(_i + 1, _j + 1, _k);
  if (vid <> -1) then
    VertexAdd(vid);
  vid := Get_z_vert(_i, _j + 1, _k);
  if (vid <> -1) then
    VertexAdd(vid);

  ScaleVector(PVertices^[_Nverts].P, 1 / u);
  NormalizeVector(PVertices^[_Nverts].N);
{$IFDEF UseDensity}
  PVertices^[_Nverts].Density := PVertices^[_Nverts].Density / u;
{$ENDIF}
  Result := _Nverts - 1;
end;

procedure TGLMarchingCube.add_triangle(trig: array of Integer; N: Byte;
  v12: Integer = -1);

var
  tv: array [0 .. 2] of Integer;
  t, tmod3: Integer;

begin
```

```
  for t := 0 to 3 * N - 1 do
  begin
    tmod3 := t mod 3;
    case trig[t] of
      0: tv[tmod3] := Get_x_vert(_i, _j, _k);
      1: tv[tmod3] := Get_y_vert(_i + 1, _j, _k);
      2: tv[tmod3] := Get_x_vert(_i, _j + 1, _k);
      3: tv[tmod3] := Get_y_vert(_i, _j, _k);
      4: tv[tmod3] := Get_x_vert(_i, _j, _k + 1);
      5: tv[tmod3] := Get_y_vert(_i + 1, _j, _k + 1);
      6: tv[tmod3] := Get_x_vert(_i, _j + 1, _k + 1);
      7: tv[tmod3] := Get_y_vert(_i, _j, _k + 1);
      8: tv[tmod3] := Get_z_vert(_i, _j, _k);
      9: tv[tmod3] := Get_z_vert(_i + 1, _j, _k);
      10: tv[tmod3] := Get_z_vert(_i + 1, _j + 1, _k);
      11: tv[tmod3] := Get_z_vert(_i, _j + 1, _k);
      12: tv[tmod3] := v12
    end;

    if (tv[tmod3] = -1) then
      Break;

    if (tmod3 = 2) then
    begin
      if (_Ntrigs >= _Strigs) then
      begin
        _Strigs := 2 * _Strigs;
        ReallocMem(PTriangles, _Strigs * SizeOf(TGLTriangle));
      end;

      with PTriangles^[_Ntrigs] do
      begin
        v1 := tv[0];
        v2 := tv[1];
        v3 := tv[2];
      end;
      Inc(_Ntrigs);

    end
  end
end;

function TGLMarchingCube.calc_u(v1, v2: Single): Single;
begin
  if (abs(FIsoValue - v1) >= 0.00001) then
    Result := 1
  else if (abs(FIsoValue - v2) >= 0.00001) then
    Result := 0
  else if (abs(v1 - v2) >= 0.00001) then
    Result := (FIsoValue - v1) / (v2 - v1)
  else
```

```
    Result := 0.5
end;

function TGLMarchingCube.add_x_vertex: Integer;
var
  u: Single;
begin
  test_vertex_addiction;
  u := calc_u(_Cube[0].Density, _Cube[1].Density);

  with PVertices^[_Nverts] do
  begin
    P.X := _Cube[0].P.X + u * FStepX;
    P.Y := _Cube[0].P.Y;
    P.Z := _Cube[0].P.Z;

    N.X := (1 - u) * Get_x_grad(_i, _j, _k) + u * Get_x_grad(_i + 1, _j, _k);
    N.Y := (1 - u) * Get_y_grad(_i, _j, _k) + u * Get_y_grad(_i + 1, _j, _k);
    N.Z := (1 - u) * Get_z_grad(_i, _j, _k) + u * Get_z_grad(_i + 1, _j, _k);
    NormalizeVector(N);
{$IFDEF UseDensity}
    Density := _Cube[1].Density
{$ENDIF}
  end;
  Inc(_Nverts);
  Result := _Nverts - 1;
end;

function TGLMarchingCube.add_y_vertex: Integer;
var
  u: Single;
begin
  test_vertex_addiction;
  u := calc_u(_Cube[0].Density, _Cube[3].Density);

  with PVertices^[_Nverts] do
  begin
    P.X := _Cube[0].P.X;
    P.Y := _Cube[0].P.Y + u * FStepY;
    P.Z := _Cube[0].P.Z;

    N.X := (1 - u) * Get_x_grad(_i, _j, _k) + u * Get_x_grad(_i, _j + 1, _k);
    N.Y := (1 - u) * Get_y_grad(_i, _j, _k) + u * Get_y_grad(_i, _j + 1, _k);
    N.Z := (1 - u) * Get_z_grad(_i, _j, _k) + u * Get_z_grad(_i, _j + 1, _k);
    NormalizeVector(N);
{$IFDEF UseDensity}
    Density := _Cube[3].Density
{$ENDIF}
  end;
  Inc(_Nverts);
  Result := _Nverts - 1;
end;
```

```
function TGLMarchingCube.add_z_vertex: Integer;
var
  u: Single;
begin
  test_vertex_addiction;

  u := calc_u(_Cube[0].Density, _Cube[4].Density);

  with PVertices^[_Nverts] do
  begin
    P.X := _Cube[0].P.X;
    P.Y := _Cube[0].P.Y;
    P.Z := _Cube[0].P.Z + u * FStepZ;;

    N.X := (1 - u) * Get_x_grad(_i, _j, _k) + u * Get_x_grad(_i, _j, _k + 1);
    N.Y := (1 - u) * Get_y_grad(_i, _j, _k) + u * Get_y_grad(_i, _j, _k + 1);
    N.Z := (1 - u) * Get_z_grad(_i, _j, _k) + u * Get_z_grad(_i, _j, _k + 1);
    NormalizeVector(N);
{$IFDEF UseDensity}
    Density := _Cube[4].Density
{$ENDIF}
  end;
  Inc(_Nverts);
  Result := _Nverts - 1;
end;

procedure TGLMarchingCube.clean_all(keepFacets: Boolean = False);
begin
  clean_temps;
  clean_space;
  if (not keepFacets) then
    FreeMem(PVertices);
  FreeMem(PTriangles);
  PVertices := nil;
  PTriangles := nil;
  _Nverts := 0;
  _Ntrigs := 0;
  _Sverts := 0;
  _Strigs := 0;
end;

procedure TGLMarchingCube.clean_space;
begin
  if (VoxelData <> nil) then
  begin
    FreeMem(VoxelData);
    VoxelData := nil
  end;
  FSizeX := 0;
  FSizeY := 0;
  FSizeZ := 0
```

```
end;

procedure TGLMarchingCube.clean_temps;
begin
  FreeMem(PVertsX);
  PVertsX := nil;
  FreeMem(PVertsY);
  PVertsY := nil;
  FreeMem(PVertsZ);
  PVertsZ := nil;
end;

procedure TGLMarchingCube.compute_intersection_points;
var
  k, j, i: Integer;
begin
  _Cube[0] := getVoxelData(0, 0, 0);
  _Cube[1] := getVoxelData(1, 0, 0);
  _Cube[3] := getVoxelData(0, 1, 0);
  _Cube[4] := getVoxelData(0, 0, 1);
  { _step_x:= _Cube[1].P[0] - _Cube[0].P[0] ;
    _step_y:= _Cube[3].P[1] - _Cube[0].P[1] ;
    _step_z:= _Cube[4].P[2] - _Cube[0].P[2] ; }

  for k := 0 to FSizeZ - 2 do
  begin
    _k := k;
    for j := 0 to FSizeY - 2 do
    begin
      _j := j;
      for i := 0 to FSizeX - 2 do
      begin
        _i := i;
        _Cube[0] := getVoxelData(_i, _j, _k);
        _Cube[1] := getVoxelData(_i + 1, _j, _k);
        _Cube[3] := getVoxelData(_i, _j + 1, _k);
        _Cube[4] := getVoxelData(_i, _j, _k + 1);

        if (Internal(_Cube[0].Density)) then
        begin
          if (not Internal(_Cube[1].Density)) then
            set_x_vert(add_x_vertex(), _i, _j, _k);
          if (not Internal(_Cube[3].Density)) then
            set_y_vert(add_y_vertex(), _i, _j, _k);
          if (not Internal(_Cube[4].Density)) then
            set_z_vert(add_z_vertex(), _i, _j, _k);
        end
        else
        begin
          if (Internal(_Cube[1].Density)) then
            set_x_vert(add_x_vertex(), _i, _j, _k);
          if (Internal(_Cube[3].Density)) then
```

```
      set_y_vert(add_y_vertex(), _i, _j, _k);
     if (Internal(_Cube[4].Density)) then
      set_z_vert(add_z_vertex(), _i, _j, _k);
    end
   end
  end
 end
end;

procedure TGLMarchingCube.ReDim(ASizeX, ASizeY, ASizeZ: Integer;
 xMin, xMax, yMin, yMax, zMin, zMax: Single);
begin
 clean_all;
 FSizeX := ASizeX;
 FSizeY := ASizeY;
 FSizeZ := ASizeZ;
 FxMin := xMin;
 FxMax := xMax;
 FyMin := yMin;
 FyMax := yMax;
 FzMin := zMin;
 FzMax := zMax;

 FStepX := (FxMax - FxMin) / (FSizeX - 1);
 FStepY := (FyMax - FyMin) / (FSizeY - 1);
 FStepZ := (FzMax - FzMin) / (FSizeZ - 1);

 VoxelData := nil;
 PVertsX := nil;
 PVertsY := nil;
 PVertsZ := nil;
 _Nverts := 0;
 _Ntrigs := 0;
 _Sverts := 0;
 _Strigs := 0;
 PVertices := nil;
 PTriangles := nil;

 Init_all;
 // FillVoxelData;
end;

constructor TGLMarchingCube.Create;
begin
 FOriginalMC := True; // now only original MC is implemented
 FIsoValue := 0;
 ScalarField := nil;
 //          SFSphere;//SFKleinBottle;//SFMinkowski;//          SFChmutov2;//
SFChmutov1;//SFDoubleTorus;// SFToroidal;

 VoxelData := nil;
 PVertsX := nil;
```

```
  PVertsY := nil;
  PVertsZ := nil;
  _Nverts := 0;
  _Ntrigs := 0;
  _Sverts := 0;
  _Strigs := 0;
  PVertices := nil;
  PTriangles := nil;
end;

constructor TGLMarchingCube.Create(SizeX, SizeY, SizeZ: Integer;
  AIsoValue: TGLScalarValue = 0.0; xMin: Single = -0.5; xMax: Single = 0.5;
  yMin: Single = -0.5; yMax: Single = 0.5; zMin: Single = -0.5;
  zMax: Single = 0.5);
begin
  FOriginalMC := True; // now only original MC is implemented
  FIsoValue := AIsoValue;
  ScalarField := SFSphere;
  //SFKleinBottle;
  //SFMinkowski;
  //SFChmutov2;
  //SFChmutov1;
  //SFDoubleTorus;
  //SFToroidal;
  ReDim(SizeX, SizeY, SizeZ, xMin, xMax, yMin, yMax, zMin, zMax);
  FillVoxelData;
end;

destructor TGLMarchingCube.Destroy;
begin
  clean_all;
  inherited;
end;

function TGLMarchingCube.getVoxelValue(i, j, k: Integer): TGLScalarValue;
begin
  Result := VoxelData^[i + j * FSizeX + k * FSizeX * FSizeY].Density
end;

function TGLMarchingCube.getVoxelData(i, j, k: Integer): TGLVoxel;
begin
  Result := VoxelData^[i + j * FSizeX + k * FSizeX * FSizeY]
end;

function TGLMarchingCube.Get_x_grad(i, j, k: Integer): Single;
begin
  if (i > 0) then
    if (i < FSizeX - 1) then
      Result := (getVoxelValue(i + 1, j, k) - getVoxelValue(i - 1, j, k)) / 2
    else
      Result := getVoxelValue(i, j, k) - getVoxelValue(i - 1, j, k)
  else
```

```
    Result := getVoxelValue(i + 1, j, k) - getVoxelValue(i, j, k)
end;

function TGLMarchingCube.Get_x_vert(i, j, k: Integer): Integer;
begin
  Result := PVertsX^[i + j * FSizeX + k * FSizeX * FSizeY]
end;

function TGLMarchingCube.Get_y_grad(i, j, k: Integer): Single;
begin
  if (j > 0) then
    if (j < FSizeY - 1) then
      Result := (getVoxelValue(i, j + 1, k) - getVoxelValue(i, j - 1, k)) / 2
    else
      Result := getVoxelValue(i, j, k) - getVoxelValue(i, j - 1, k)
  else
    Result := getVoxelValue(i, j + 1, k) - getVoxelValue(i, j, k)
end;

function TGLMarchingCube.Get_y_vert(i, j, k: Integer): Integer;
begin
  Result := PVertsY^[i + j * FSizeX + k * FSizeX * FSizeY]
end;

function TGLMarchingCube.Get_z_grad(i, j, k: Integer): Single;
begin
  if (k > 0) then
    if (k < FSizeZ - 1) then
      Result := (getVoxelValue(i, j, k + 1) - getVoxelValue(i, j, k - 1)) / 2
    else
      Result := getVoxelValue(i, j, k) - getVoxelValue(i, j, k - 1)
  else
    Result := getVoxelValue(i, j, k + 1) - getVoxelValue(i, j, k)
end;

function TGLMarchingCube.Get_z_vert(i, j, k: Integer): Integer;
begin
  Result := PVertsZ^[i + j * FSizeX + k * FSizeX * FSizeY]
end;

procedure TGLMarchingCube.Init_all;
begin
  Init_temps;
  Init_space;

  if (PVertices <> nil) then
    FreeMem(PVertices);
  if (PTriangles <> nil) then
    FreeMem(PTriangles);
  _Nverts := 0;
  _Ntrigs := 0;
  _Sverts := ALLOC_SIZE;
```

```
  _Strigs := ALLOC_SIZE;

  GetMem(PVertices, _Sverts * SizeOf(TGLVertex));
  GetMem(PTriangles, _Strigs * SizeOf(TGLTriangle));
end;

procedure TGLMarchingCube.Init_space;
begin
  VoxelData := AllocMem(FSizeX * FSizeY * FSizeZ * SizeOf(TGLVoxel));
end;

procedure TGLMarchingCube.Init_temps;
var
  spaceSize: Longword;
begin
  spaceSize := FSizeX * FSizeY * FSizeZ;
  GetMem(PVertsX, spaceSize * SizeOf(Integer));
  GetMem(PVertsY, spaceSize * SizeOf(Integer));
  GetMem(PVertsZ, spaceSize * SizeOf(Integer));

  FillChar(PVertsX^, spaceSize * SizeOf(Integer), -1);
  FillChar(PVertsY^, spaceSize * SizeOf(Integer), -1);
  FillChar(PVertsZ^, spaceSize * SizeOf(Integer), -1);
end;

function TGLMarchingCube.Internal(AValue: TGLScalarValue): Boolean;
begin
  Result := AValue <= FIsoValue
end;

procedure TGLMarchingCube.process_cube;
var
  nt: Byte;
begin
  if (FOriginalMC) then
  begin
    nt := 0;
    while (MC_TRITABLE[_lut_entry][3 * nt] <> -1) do
      Inc(nt);
    add_triangle(MC_TRITABLE[_lut_entry], nt);
    Exit;
  end;
  {
    TODO complete algorithm with various tiling...
  }
end;

procedure TGLMarchingCube.Run;
var
  i, j, k, P: Integer;
begin
  if (PVertsX = nil) then
```

```
  begin
    Init_temps;
    _Nverts := 0;
    _Ntrigs := 0;
  end;
  Compute_Intersection_Points;
  for k := 0 to FSizeZ - 2 do
  begin
    _k := k;
    for j := 0 to FSizeY - 2 do
    begin
      _j := j;
      for i := 0 to FSizeX - 2 do
      begin
        _i := i;
        _lut_entry := 0;
        for P := 0 to 7 do
        begin
          _Cube[P] := getVoxelData(i + ((P xor (P shr 1)) and 1),
            j + ((P shr 1) and 1), k + ((P shr 2) and 1));
          // _Cube[p]:= getVoxelData( i+((p^(p>>1))&1), j+((p>>1)&1), k+((p>>2)&1) ) ;
          if (Internal(_Cube[P].Density)) then
            _lut_entry := _lut_entry or (1 shl P);
        end;
        process_cube;
      end
    end
  end;
  clean_temps;
end;

procedure TGLMarchingCube.Run(IsoValue: TGLScalarValue);
begin
  FIsoValue := IsoValue;
  Run
end;

procedure TGLMarchingCube.setVoxelValue(i, j, k: Integer; HfValue:
TGLScalarValue);
begin
  VoxelData^[i + j * FSizeX + k * FSizeX * FSizeY].Density := HfValue
end;

procedure TGLMarchingCube.set_x_vert(a_val, i, j, k: Integer);
begin
  PVertsX^[i + j * FSizeX + k * FSizeX * FSizeY] := a_val
end;

procedure TGLMarchingCube.set_y_vert(a_val, i, j, k: Integer);
begin
  PVertsY^[i + j * FSizeX + k * FSizeX * FSizeY] := a_val
end;
```

```
procedure TGLMarchingCube.set_z_vert(a_val, i, j, k: Integer);
begin
  PVertsZ^[i + j * FSizeX + k * FSizeX * FSizeY] := a_val
end;
procedure TGLMarchingCube.test_vertex_addiction;
begin
  if _Nverts >= _Sverts then
  begin
    _Sverts := 2 * _Sverts;
    ReallocMem(PVertices, _Sverts * SizeOf(TGLVertex))
  end;
end;
function TGLMarchingCube.voxel(i, j, k: Integer): PGLVoxel;
begin
  if (k >= FSizeZ) or (j >= FSizeY) or (i >= FSizeX) then
    Result := nil
  else
    Result := @VoxelData^[i + j * FSizeX + k * FSizeX * FSizeY]
end;
procedure TGLMarchingCube.FillVoxelData;
var
  iX, iY, iZ: Integer;
  X, Y, Z: Single;
begin
  for iX := 0 to FSizeX - 1 do
  begin
    X := FxMin + iX * FStepX;
    for iY := 0 to FSizeY - 1 do
    begin
      Y := FyMin + iY * FStepY;
      for iZ := 0 to FSizeZ - 1 do
        with VoxelData^[iX + iY * FSizeX + iZ * FSizeX * FSizeY] do
        begin
          Z := FzMin + iZ * FStepZ;
          MakeVector(P, X, Y, Z);
          Density := ScalarField(X, Y, Z);
          if Internal(Density) then
            Status := bpInternal
          else
            Status := bpExternal
        end;
    end;
  end;
end;

procedure TGLMarchingCube.FillVoxelData(AIsoValue: TGLScalarValue;
  AScalarField: TGLScalarField = nil);
begin
  FIsoValue := AIsoValue;
  if Assigned(AScalarField) then
    ScalarField := AScalarField;
  FillVoxelData;
```

```
end;
procedure TGLMarchingCube.FillVoxelData(AIsoValue: TGLScalarValue;
  AScalarField: TGLScalarFieldInt);
var
  iX, iY, iZ: Integer;
  X, Y, Z: Single;
begin
  FIsoValue := AIsoValue;
  for iX := 0 to FSizeX - 1 do
  begin
    X := FxMin + iX * FStepX;
    for iY := 0 to FSizeY - 1 do
    begin
      Y := FyMin + iY * FStepY;
      for iZ := 0 to FSizeZ - 1 do
        with VoxelData^[iX + iY * FSizeX + iZ * FSizeX * FSizeY] do
        begin
          Z := FzMin + iZ * FStepZ;
          MakeVector(P, X, Y, Z);
          Density := AScalarField(iX, iY, iZ);
          if Internal(Density) then
            Status := bpInternal
          else
            Status := bpExternal
        end;
    end;
  end;
end;
procedure TGLMarchingCube.CalcVertices(Vertices: TGLVertexList;
  Alpha: Single = 1);
var
  i: Integer;
  vx1, vx2, vx3: TGLVertexData;
  function GetNrmColor(Nrm: TAffineVector): TVector;
  begin
    Result.V[0] := 0;
    if Nrm.V[0] > 0.0 then
      Result.V[0] := Result.V[0] + Nrm.V[0];
    if Nrm.V[1] < 0.0 then
      Result.V[0] := Result.V[0] - 0.5 * Nrm.V[1];
    if Nrm.V[2] < 0.0 then
      Result.V[0] := Result.V[0] - 0.5 * Nrm.V[2];
    Result.V[1] := 1;
    if Nrm.V[0] < 0.0 then
      Result.V[1] := Result.V[1] - 0.5 * Nrm.V[0];
    if Nrm.V[1] > 0.0 then
      Result.V[1] := Result.V[1] + Nrm.V[1];
    if Nrm.V[2] < 0.0 then
      Result.V[1] := Result.V[1] - 0.5 * Nrm.V[2];
    Result.V[2] := 0;
    if Nrm.V[0] < 0.0 then
      Result.V[2] := Result.V[2] - 0.5 * Nrm.V[0];
```

```
    if Nrm.V[1] < 0.0 then
      Result.V[2] := Result.V[2] - 0.5 * Nrm.V[1];
    if Nrm.V[2] > 0.0 then
      Result.V[2] := Result.V[2] + Nrm.V[2];
    Result.V[3] := 0.3
  end;
  function GetColor(H: TGLScalarValue): TVector;
  begin
    Result := VectorMake(0.890, 0.855, 0.788, Alpha)
    { if H <= 10 then Result:= VectorMake(0.922, 0.957, 0.980, 1.000)  //<=10
      else if H <= 50 then Result:= VectorMake(0.541, 0.027, 0.027, 1.000) // 10..50
      else if H <= 300 then Result:= VectorMake(0.941, 0.910, 0.859, 1.000) //50..300
      else if H <= 2000 then Result:= VectorMake(0.965, 0.969, 0.973, 1.000) //350.. 2000
      else if H <= 4000 then Result:= VectorMake(0.890, 0.855, 0.788, 1.000) //2000..4000
      else Result:= VectorMake(0.9, 0.9, 0.6, 1.0) }
  end;
begin
  Vertices.Clear;
  Vertices.Capacity := 3 * _Ntrigs;

  for i := 0 to _Ntrigs - 1 do
    with PTriangles^[i] do
    begin
      vx1.coord := PVertices^[v1].P;
      vx1.normal := PVertices^[v1].N;
      vx2.coord := PVertices^[v2].P;
      vx2.normal := PVertices^[v2].N;
      vx3.coord := PVertices^[v3].P;
      vx3.normal := PVertices^[v3].N;
{$IFDEF UseDensity}
      vx1.Color := GetColor(PVertices^[v1].Density); // GetNrmColor(vx1.normal);
      vx2.Color := GetColor(PVertices^[v2].Density); // GetNrmColor(vx2.normal);
      vx3.Color := GetColor(PVertices^[v3].Density); // GetNrmColor(vx3.normal);
{$ELSE}
      vx1.Color := VectorMake(0.890, 0.855, 0.788, Alpha);
      vx2.Color := VectorMake(0.890, 0.855, 0.788, Alpha);
      vx3.Color := VectorMake(0.890, 0.855, 0.788, Alpha);
{$ENDIF}
      // Vertices.AddVertex3(vx1, vx2, vx3); seems to be correct the next line
      Vertices.AddVertex3(vx3, vx2, vx1);
    end;
end;
procedure TGLMarchingCube.CalcMeshObject(AMeshObject: TMeshObject; Alpha: Single);
var
  i: Integer;
  vx1, vx2, vx3: TGLVertexData;
begin
  AMeshObject.Clear;
  AMeshObject.Vertices.Capacity := _Nverts;
  AMeshObject.Normals.Capacity := _Nverts;
  AMeshObject.Colors.Capacity := _Nverts;
```

```
      with TFGVertexIndexList.CreateOwned(AMeshObject.FaceGroups) do
      begin
        Mode := fgmmTriangles;
        for i := 0 to _Nverts - 1 do
        begin
          AMeshObject.Vertices.Add(PVertices^[i].P);
          AMeshObject.Normals.Add(PVertices^[i].N);
          // AMeshObject.Normals.Add(VectorScale(PVertices^[i].N, -1));
          // AMeshObject.Colors.Add(VectorMake(0.890, 0.855, 0.788, Alpha));
        end;

        for i := 0 to _Ntrigs - 1 do
          // with PTriangles^[i] do VertexIndices.Add(v1, v2, v3);
          with PTriangles^[i] do
            VertexIndices.Add(v3, v2, v1);
      end;
    end;
end.
```