ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖИНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА МЕТОДА ПРЕДСТАВЛЕНИЯ ВОКСЕЛЬНЫХ ДАННЫХ В ВИДЕ ОКТО-ДЕРЕВА ДЛЯ УСКОРЕНИЯ ВИЗУАЛИЗАЦИИ БЛОЧНЫХ МОДЕЛЕЙ

Выпускная квалификационная работа
обучающегося по направлению подготовки 010200.62,
Математика и компьютерные науки
очной формы обучения, группы 07001303
Шаповалова Максима Витальевича

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

БЕЛГОРОД 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ…………………………………………………………………..3
1. АНАЛИЗ ПРОБЛЕМЫ ПРЕДСТАВЛЕНИЯ ВОКСЕЛЬНЫХ ДАННЫХ В ВИДЕ ОКТО-ДЕРЕВА…………………………………………..7
1.1. Основные понятия, структура и особенности существующих методов……………………………………………………………………8
1.2. Методы представления воксельных данных в виде окто-дерева…12
1.3. Постановка задачи…………………………………………...21
2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ, ТЕХНОЛОГИЙ И ВЫРАБОТКА ПОДХОДА К ВЫЧИСЛЕНИЯМ……………………….23
2.1. Проектирование модуля для системы недропользовани…………23
2.2. Использование среды разработки RAD Studio……………………24
2.3. Использование библиотек VCL, FMX, CGAL, GLScene…………30
3. Реализация автоматизированной системы вокселизации полигональных моделей …………………………………………………….31
3.1. Проектирование системы визуализации……………..…………31
3.2 Реализация алгоритма визуализации с использованием структуры октодерева ………………………………………………………………39
3.3 Geoblock система для недропользования………………………..43
4. АПРОБАЦИЯ МЕТОДА ПРЕДСТАВЛЕНИЯ ВОКСЕЛЬНЫХ ДАННЫХ В ВИДЕ ОКТО-ДЕРЕВА ДЛЯ УСКОРЕНИЯ ВИЗУАЛИЗАЦИИ БЛОЧНЫХ МОДЕЛЕЙ……………………………45
ЗАКЛЮЧЕНИЕ…………………………………………………………...49
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ………...51
ПРИЛОЖЕНИЕ……………………………………………………………..55

ВВЕДЕНИЕ

Данная работа представляет собой программную реализацию метода представления воксельных данных в виде окто-дерева для ускорения визуализации блочных моделей. Вопрос эффективного способа представления изображений некоторой структурой данных остается актуальным в течении долгого времени. Ответ на данный вопрос всегда является неким компромиссом между объемом памяти, необходимым для представления изображения и временем доступа к элементам изображения.

Одной из наиболее изученных и хорошо зарекомендовавших себя структур данных для представления 2D-изображений является квадротомическое дерево или квадродерево (quadtree). Благодаря естественной иерархической структуре и способу организации квадродеревья сочетают в себе значительную экономию объемов памяти с эффективностью доступа к элементам изображения. Идеология квадродеревьев применяется не только для представления растровых изображений, но и используется для эффективной организации больших баз любых пространственных данных, состоящих как из растровых, так и векторных изображений. Аналогичные принципы лежат в основе структуры данных для представления трехмерных изображений и других трехмерных данных – октотомического дерева или октодерева (octree).

Одно из популярных применений октодеревьев – ускорение алгоритмов трассировки лучей. Вместо отслеживания пересечения лучей, исходящих из каждого пиксела с каждым объектом сцены при проверках используется квадротомированное представление сцены. Вначале проверяется пересечение луча с октантами, соответствующими дочерним узлам корня октодерева (пересечение с корневым октантом всегда существует). В случае пересечения луча и октанта проверки осуществляются уже для его дочерних октантов. Этот процесс продолжается до тех пор, пока

не будет найдено пересечение луча и листового узла. Листовой узел либо содержит объект сцены, пересечение луча с которым требуется обработать, либо не содержит таковых. Возникновение последнего случая означает, что луч покидает сцену или достигает ее фона. Такой способ проверки пересечений намного быстрее классических приемов трассировки лучей, поскольку проверке подвергаются не все объекты, а лишь небольшая их часть. Естественно, если сцена содержит немного объектов, то выигрыш во времени будет небольшим или вовсе будет отсутствовать. Выигрыш становится заметным, когда количество объектов сцены превышает глубину октодерева приблизительно в четыре раза. На рисунке показано максимальное количество тестов, которое может потребоваться при использовании октодерева.

Другое применение октодеревьев находят при квантовании цветов. Квантование цветов - это метод, применяемый для определения достаточно близкого цветового значения при представлении изображения меньшим количеством цветов, чем содержит его оригинал. Для представления трехмерных изображений и трехмерных графических сцен структурой данных, основанной на сходных принципах, является октотомическое дерево или октодерево (octree). Подобно квадродереву октотомическое дерево является иерархической структурой, но представляет изображение на уровне объектов или вокселов, а не пикселов.

Октотомические деревья являются естественным распространением концепции квадродерева для представления трехмерного пространства. Подобно тому, как при построении квадродерева область разбивается на четыре части при построении октодерева трехмерный объект подразделяется на восемь кубов (октантов). Если какой-либо из октантов является однородным, т.е. он располагается либо целиком внутри объекта, либо целиком снаружи, разбиение заканчивается. Иначе, если октант не является однородным, т.е. октант пересекается граничной поверхностью объекта, он разбивается далее на восемь подоктантов. Процесс разбиения завершается,

когда все листовые узлы октодерева станут однородными, возможно с некоторой погрешностью.

В качестве простейшего варианта октотомической структуры данных часто называют октодерево регионов. Октодерево имеет те же недостатки, что и квадродерево, в том смысле, что оно является лишь приближением изображения. Октодерево требует значительных вычислительных затрат на построение, поскольку требуется просмотреть большое количество элементов данных.

Октодеревья обладают той же структурой, что и квадродеревья за исключением того, что каждый родительский узел дерева имеет не четыре дочерних узла, а восемь. Сходство структуры квадродерева и октодерева можно увидеть на рисунке.

Принципы работы с октодеревьями почти ничем не отличаются от способов работы с квадродеревьями. Разница заключается в том, что исходными данными при построении октодеревьев являются трехмерные объекты, а не двумерные изображения. Объекты соотносятся с листовыми узлами октодерева, которые формируются в том случае, когда объект либо целиком лежит внутри, либо целиком вне октанта, соответствующего листовому узлу. Корневой узел октодерева соответствует полной трехмерной сцене. Все узлы кроме листовых имеют по восемь дочерних узлов (октантов).

Одним из главных достоинств октодерева является возможность представления объектов почти произвольной формы вне зависимости от того выпуклый ли он, вогнутый или многосвязный. Точность представления определяется размерами наименьшего октанта разбиения. Это свойство октодеревьев находит, пожалуй, свое основное применение в приложениях построения реалистичных изображений путем трассировки лучей, при которой точность представления объектов является критической величиной для достижения качественных результатов.

С помощью октотомического дерева очень просто производятся расчёты таких величин как площадь поверхности, объем, центр масс.

Октодеревья обладают всеми преимуществами и эффективностью, свойственными любой древесной структуре.

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

В ходе работы были поставлены следующие задачи:

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

В дипломной работе содержится 54 страниц без приложения, 19 рисунка и 1 формула. В процессе создания было использовано 30 литературных источника.

1. АНАЛИЗ ПРОБЛЕМЫ ПРЕДСТАВЛЕНИЯ ВОКСЕЛЬНЫХ ДАННЫХ В ВИДЕ ОКТО-ДЕРЕВА

Текстурирование – очень эффективный способ обогатить внешний вид детализированных полигональных моделей. Текстуры могут не только хранить информацию о цвете, но и о нормалях для отображения рельефа и различных атрибутов затенения, что может создать привлекательный эффект поверхности. Тем не менее, наложение текстур требует параметризации сетки, связав 2D координаты текстуры каждой из её вершин. Искажения и швы часто делают этот процесс очень сложным, особенно на сложных сетках.

2D-параметризации можно избежать, определив структуру внутри объема ограждающих граней объекта. Ранние исследовательские работы показали, как иерархические 3D-структуры данных, названные текстуры октодеревьев, могут быть использованы для эффективного хранения информации о цвете по поверхности сетки без текстурных координат.

Это дает разработчику два преимущества. Во-первых, цвет сохраняется только там, где поверхность пересекает объем, что позволяет снизить требования к памяти. Во-вторых, поверхность регулярно сэмплируется и в результате текстура не страдает от каких-либо искажений. Кроме рисования сетки картина любое приложение, которое требует хранения информации на сложной поверхности, может извлечь выгоду из этого подхода.

В следующих подразделах описывается, как реализовать работу с текстурами октодерева на современных графических процессорах. Октодеревья будут храниться непосредственно в текстурной памяти. Также будут выявлены чёткие компромиссы между производительностью, эффективностью хранения и качества рендеринга. После описания реализации это будет показано на двух различных интерактивных приложениях:

Приложение, отображающее только поверхности моделей. В частности, будут рассмотрены различные способы фильтрации текстур. Также будут показано, как текстуры, определенные в октодереве могут быть преобразованы в стандартные текстуры, возможно даже во время выполнения.

Физическое моделирование жидкости, протекающей вдоль поверхности. Моделирование полностью работает на графическом процессоре.

1.1. Основные понятия, структура и особенности существующих методов

Квадрадером является структура данных дерева, в которой каждый внутренний узел имеет ровно четыре потомка. Квадранты - двумерный аналог октерей и чаще всего используются для разбиения двумерного пространства путем рекурсивного разбиения его на четыре квадранта или области. Данные, связанные с листовой ячейкой, варьируются в зависимости от приложения, но листовая ячейка представляет собой «блок интересной пространственной информации».

Разделенные области могут быть квадратными или прямоугольными или могут иметь произвольные формы.

Октодерево - это структура данных дерева, в которой каждый внутренний узел имеет ровно восемь дочерних элементов. Октодерево наиболее часто используются для разбиения трехмерного пространства путем рекурсивного разбиения его на восемь октантов. Октодерево являются трехмерным аналогом квадратов. Октодерево часто используются в 3D-графике.

Каждый узел в октодереве подразделяет пространство, которое он представляет, на восемь октантов. В точечном регионе (PR) octree узел хранит явную трехмерную точку, которая является «центром» подразделения

для этого узла; Точка определяет один из углов для каждого из восьми потомком. В основанном на матрице (MX) октодереве точка подразделения является неявно центром пространства, которое представляет узел. Корневой узел PR-октодека может представлять бесконечное пространство; Корневой узел октодерева MX должен представлять конечное ограниченное пространство, так что неявные центры определены правильно. Обратите внимание, что Октодерево - это не то же самое, что деревья k-d: деревья k-d, разделенные по размеру и окты, разделенные вокруг точки. Кроме того, деревья k-d всегда являются двоичными, что не относится к октодереву. Используя поиск по глубине, узлы должны быть, пройдены, и только требуемые поверхности должны быть просмотрены.

Полигон – это геометрическая фигура, определяется как замкнутая ломаная. Существуют три различных варианта определения полигонов:

- плоская замкнутая ломаная;
- плоская замкнутая ломаная без самопересечений;
- часть плоскости, ограниченная замкнутой ломаной.

Воксель – элемент объёмного изображения, содержащий значение элемента растра в трёхмерном пространстве. Воксели являются аналогами пикселов для трехмёрного пространства. Воксельные модели часто используются для визуализации и анализа медицинской и научной информации. Как и в случае с пикселами, сами по себе воксели не содержат информации о своих координатах в пространстве. Их координаты вычисляются из их позиции в трёхмерной матрице — структуре, моделирующей объёмный объект или поле значений параметра в трёхмерном пространстве. Этим воксели отличаются от объектов векторной графики, для которых известны координаты их опорных точек (вершин) и прочие параметры. Воксельные модели имеют определенное разрешение. Каждый воксель имеет определенное значение, например, цвет. Одной из новейших перспективных технологий, позволяющей делать эффективную детализацию

воксельных объектов, является разреженное воксельное октодерево. В числе её преимуществ: значительная экономия памяти, естественная генерация уровней детализации и высокая скорость обработки в рейкастинге. Первый узел дерева — корень, является кубом, содержащим весь объект целиком. Каждый узел или имеет 8 кубов-потомков или не имеет никаких потомков. В результате всех подразбиений получается регулярная трёхмерная сетка вокселей.

Растр – представляет собой сетку пикселей или цветных точек (обычно прямоугольную) на компьютерном мониторе, бумаге и других отображающих устройствах и материалах. Важными характеристиками изображения являются:

- количество пикселей — размер. Может указываться отдельно количество пикселей по ширине и высоте (1024×768, 640×480, …) или же, редко, общее количество пикселей (часто измеряется в мегапикселях);
- количество используемых цветов или глубина цвета (эти характеристики имеют следующую зависимость: где f — количество цветов, a — глубина цвета);
- цветовое пространство (цветовая модель) RGB, CMYK, XYZ и др.
- разрешение — справочная величина, говорящая о рекомендуемом размере пикселя изображения.

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.

Графический процессор (GPU), иногда называемый визуальным процессором (VPU), представляет собой специализированную электронную схему, предназначенную для быстрого манипулирования и изменения памяти для ускорения создания изображений в буфере кадров, предназначенном для вывода на дисплей. Графические процессоры используются во встроенных системах, мобильных телефонах, персональных компьютерах, рабочих станциях и игровых консолях. Современные графические процессоры очень эффективны при обработке компьютерной графики и обработки изображений, а их высокопараллельная структура делает их более эффективными, чем универсальные процессоры для алгоритмов, где обработка больших блоков данных выполняется параллельно. В персональном компьютере графический процессор может присутствовать на видеокарте, или он может быть встроен на материнскую плату или, в некоторых CPU, на кристалле процессора.

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 представления воксельных данных в виде окто-дерева

Так как в основе структуры октодеревьев лежат принципы квадродеревьев, то сначала рассмотрим принципы работы с квадродеревьями.

Квадротомическое дерево (квадродерево) – структура данных, используемая для представления двумерных пространственных данных. Существует несколько типов квадродеревьев в зависимости от базового типа данных (точки, площади, кривые, поверхности или объемы). Наиболее общим типом квадродерева и примером использования является квадродерево растрового изображения (см. рис. 1.1, рис. 1.2 и рис. 1.3) – примеры изображения и его бинарного представления, блоков разбиения и квадродерева. Далее, для конкретности, под пространственными данными везде будем понимать растровое изображение.

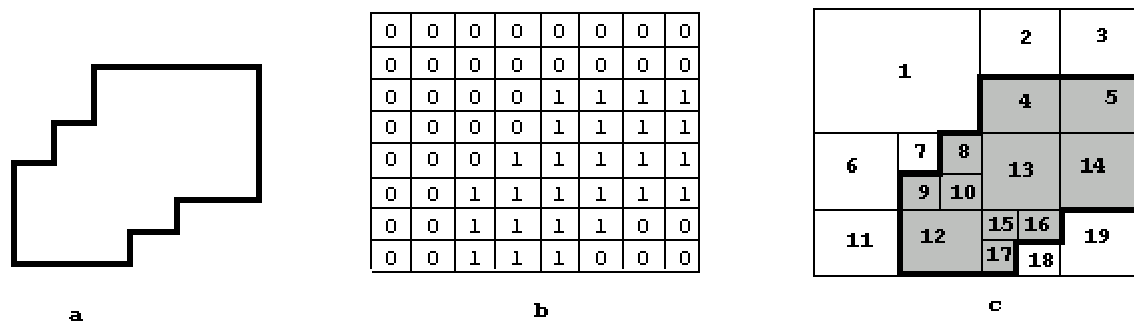

Рис. 1.1 (a) изображение, (b) его бинарный образ,
(c) его квадротомическое разбиение

Следуя ранним исследованиям, под двумерным изображением понимается массив элементов изображения (пикселов). Если каждый из пикселов имеет только два состояния – черный или белый (подсвечен или нет), то изображение называется бинарным. Если пикселы могут принимать более двух значений, то их значения могут трактоваться как оттенки серого, а изображение в этом случае называется изображением в градациях серого.

Цветные изображения организованы аналогичным образом. Квадродеревья наиболее широко используются для работы с двухцветными изображениями, поэтому в дальнейшем будем иметь дело преимущественно с бинарными изображениями. Будем также в дальнейшем подразумевать, что фоновым в бинарном изображении является белый цвет.

При построении квадродерева двумерное изображение рекурсивно подразделяется на квадранты. Каждый из четырех квадрантов становится узлом квадротомического дерева. Больший квадрант становится узлом более высокого иерархического уровня квадродерева, а меньшие квадранты появляются на более низких уровнях. Преимущества такой структуры в том, что регулярное разделение обеспечивает простое и эффективное накопление, восстановление и обработку данных. Простота проистекает из геометрической регулярности разбиения, а эффективность – за счет хранения только узлов с данными, которые представляют интерес. Основополагающая идея квадродерева – комбинирование одинаковых или сходных элементов

данных и кодирование больших однородных совокупностей данных малым количеством битов.

Корневой узел соответствует изображению в целом и имеет четыре дочерних узла, которые ассоциируются с четырьмя квадрантами исходного изображения (обозначаемыми NW – северо-западный, NE – северо-восточный, SW – юго-западный, SE – юго-восточный). В свою очередь каждый из дочерних узлов корня дерева также имеет по четыре дочерних узла, ассоциированных с шестнадцать субквадрантами исходного изображения. Дочерние узлы следующего уровня представляют собой шестьдесят четыре квадранта, составляющих исходное изображение и так далее.

В сформированном выше описанном способе передачи квадродерева дочерним узлам дерева (узлам, соответствующим каждому одиночному пикселу изображения) приписывают цвет связанного с ними пиксела (черный или белый). Если не дочерний узел имеет среди дочерних узлов, узлы как одного, так и другого цвета, ему приписывается серый цвет. Если же все дочерние узлы не дочернего узла дерева "окрашены" в один и тот же цвет, то такому узлу приписывается этот цвет, а его дочерние узлы исключаются из дерева. Таким образом, в квадродереве могут отсутствовать некоторые ветки, представляющие собой достаточно большие одноцветные области. Как показано в одном из исследований, квадродеревья и их варианты оказываются полезными в различных приложениях таких, как обработка изображений, машинная графика, распознавание образов, робототстроении и картографии.

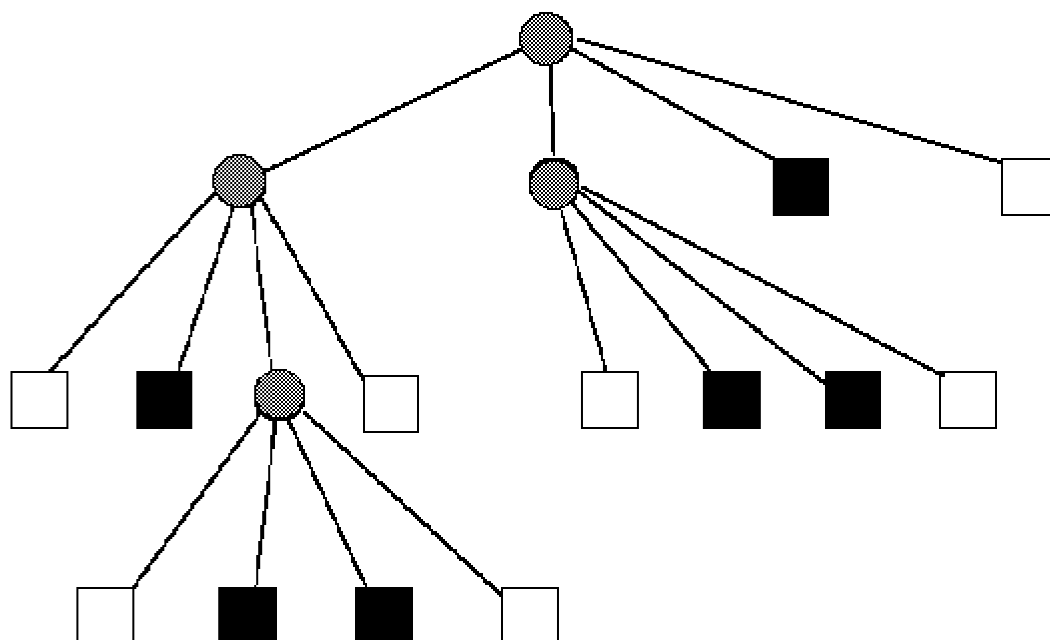

Рис. 1.2 Пример квадродерева

Построение квадродерева**.** На каждом этапе построения квадродерева изображение разбивается на четыре квадранта и каждому присваивается одно из следующих значений:

1. белый => квадрант полностью белый. Обозначается белым квадратом.

2. черный => квадрант полностью черный. Обозначается черным квадратом.

3. серый => квадрант - смесь черного и белого. Обозначается белым кругом.

На нулевом этапе полному изображению (рис. 1.1a) сопоставляется корневой узел дерева (рис. 1.2). Далее четырем равновеликим квадрантам первого этапа разбиения (рис. 1.3a) ставятся в соответствие дочерние узлы первого уровня. В показанном на рисунках частном случае северо-западный *NW*-квадрант обозначен белым квадратом, а остальные три - серыми кругами (рис. 1.4). На очередном этапе серые квадранты снова подвергаются разбиению (на рис. 1.3b для простоты показано лишь разбиение *SW*-квадранта).

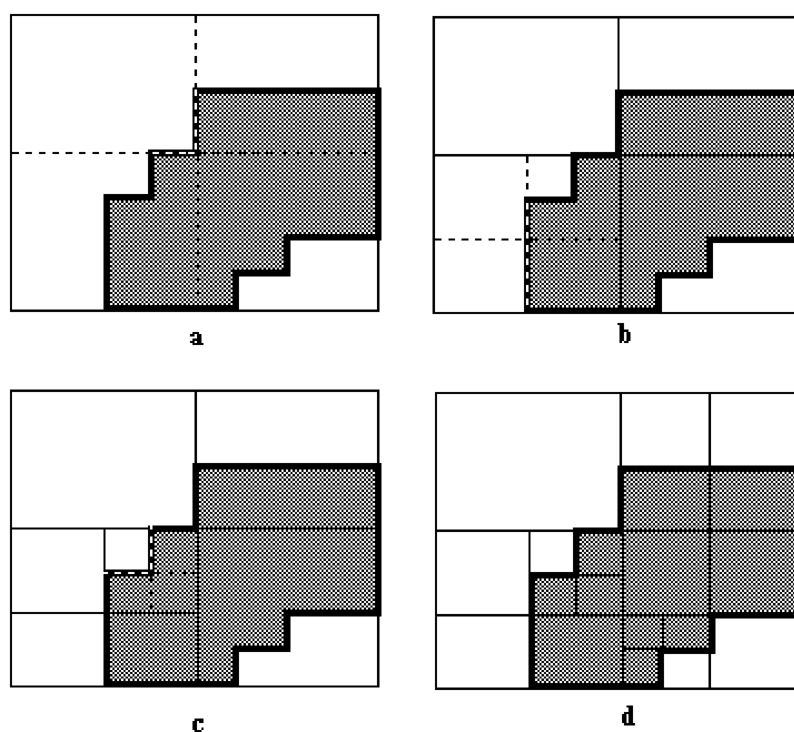

Рис. 1.3 (a) первый этап разбиения, (b) второй этап разбиения, (c) третий этап разбиения, (d) изображение полностью разбито

Как видно по рис. 1.3b *SW*-квадрант на этом этапе содержит два белых, один черный и один серый подквадранты. Они представлены в дереве на рис. 1.4 узлами второго уровня.

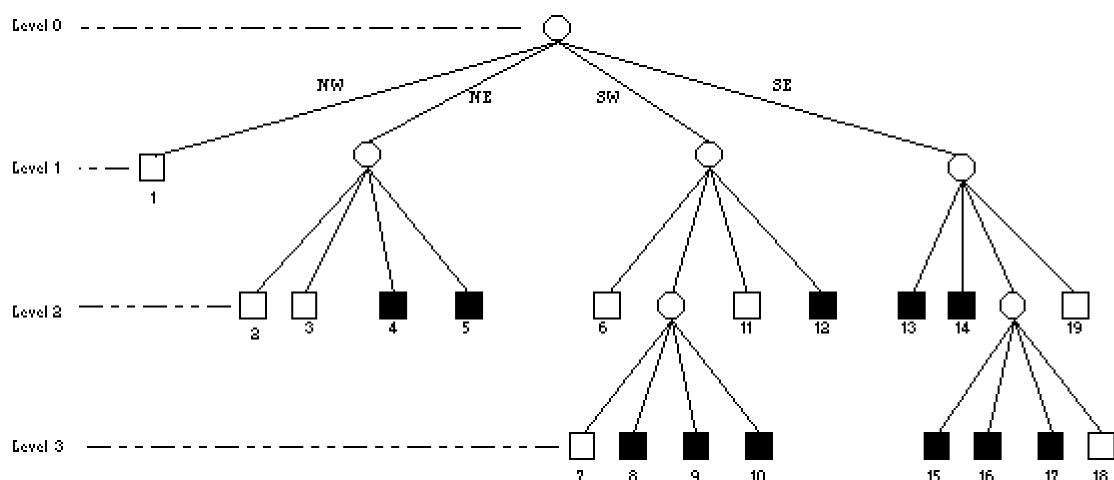

Рис. 1.4 Окончательный вид квадродерева.

Единственный серый квадрант снова разбивается. В данном случае это разбиение является последним, т.к. ни один из получившихся в результате подквадрантов не оказался серым (один белый и три черных). Подобным же образом обрабатываются и остальные квадранты изображения всех уровней.

Визуализация изображения, представленного квадродеревом, представляет собой простую рекурсивную процедуру. Начиная с корня дерева просматривается каждый узел. Если узел не является листовым, то он пропускается и просматривается его первый дочерний узел. Если этот дочерний узел не листовой, то *он* пропускается и происходит переход к *его* первому дочернему узлу и т.д. При достижении листового узла его цветовое значение отображается в соответствующей позиции. После этого происходит рекурсивный возврат к пропущенному родительскому узлу и аналогичным образом просматривается его второй дочерний узел. Этот процесс продолжается до тех пор, пока не будут посещены все листья дерева. Визуализация полного изображения происходит по мере продвижения по квадродереву.

Большинство приложений квадротомических деревьев к данным было сделано для изображений, но были проведены также современные алгоритмические разработки, которые дали результаты, сходные с теми, что используются при обработке географических данных.

Сюда входят расчеты площадей, центороидные определения, распознавание образов, классификация изображений, оверлейные операции над изображениями, выявление связанных компонент, определение соседства, преобразование расстояний, разделение изображений, сглаживание данных и усиление краевых эффектов.

Вследствие этих преимуществ отдельные исследователи предложили использовать квадротомические деревья для хранения географических данных. Основное достоинство квадродеревьев состоит в компактном представлении изображения. Компактность квадродерева целиком зависит от изображения. Изображение с большими областями, окрашенными в один

цвет представляются очень компактно, в то время как изображение, в котором все пикселы имеют разные цвета сводит на нет все преимущества квадродерева.

Как и для любой другой "древесной" структуры данных, иерархическую структура квадродерева позволяет обеспечивать высокоэффективный доступ к элементам дерева.

Существует также множество алгоритмов обхода и манипулирования деревьями в их различных формах и все эти наработки могут быть распространены на квадродеревья.

Поворот квадродерева на 90 градусов против часовой стрелки может быть осуществлен простым переприсваиванием узлов дерева в следующем порядке:

- Северо-западный (NW) => Юго-западный (SW)
- Северо-восточный (NE) => Северо-западный (NW)
- Юго-западный (SW) => Юго-восточный (SE)
- Юго-восточный (SE) => Северо-восточный (NE)

Поворот почасовой стрелке выполняется аналогично за исключением того, что пере присваивание производится в обратном порядке.

Ранние исследования также упоминают о том, что квадродерево полезно при изменении разрешения объекта. Рассмотрим, например, кваролерево на рисунке 1.4, которое представляет изображение, приведенное на рис. 1.1а. Если мы хотим изменить разрешение этого изображения, нам необходимо просто заменить все серые узлы второго уровня на черные узлы. Результатом будет новое изображение, показанное на рис. 1.5

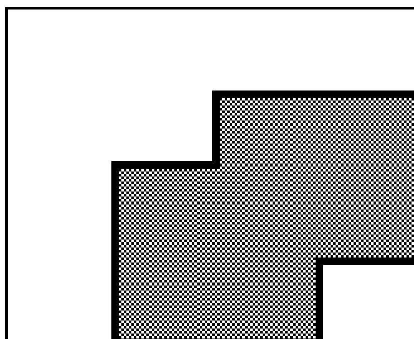

Рис. 1.5 Изображение с измененным при помощи квадродерева разрешением

Наконец, еще одним немаловажным достоинством квадродеревьев является также наличие доступных исходных текстов программ и алгоритмов для реализации этой структуры данных.

Октодеревья. Для представления трехмерных изображений и трехмерных графических сцен структурой данных, основанной на сходных принципах, является октотомическое дерево или октодерево. Подобно квадродереву октотомическое дерево является иерархической структурой, но представляет изображение на уровне объектов или вокселов, а не пикселов.

Октотомические деревья являются естественным распространением концепции квадродерева для представления трехмерного пространства. Подобно тому, как при построении квадродерева область разбивается на четыре части, при построении октодерева трехмерный объект подразделяется на восемь кубов (октантов). Если какой-либо из октантов является однородным, т.е. он располагается либо целиком внутри объекта, либо целиком снаружи, разбиение заканчивается. Иначе, если октант не является однородным, т.е. октант пересекается граничной поверхностью объекта, он разбивается далее на восемь подоктантов. Процесс разбиения завершается, когда все листовые узлы октодерева станут однородными, возможно с некоторой погрешностью.

В качестве простейшего варианта октотомической структуры данных называется октодерево регионов. Октодерево имеет те же недостатки, что и

квадродерево, в том смысле, что оно является лишь приближением изображения. Октодерево требует значительных вычислительных затрат на построение, поскольку требуется просмотреть большое количество элементов данных.

Октодеревья обладают той же структурой, что и квадродеревья за исключением того, что каждый родительский узел дерева имеет не четыре дочерних узла, а восемь. Сходство структуры квадро- и октодерева можно увидеть на рис. 1.6.

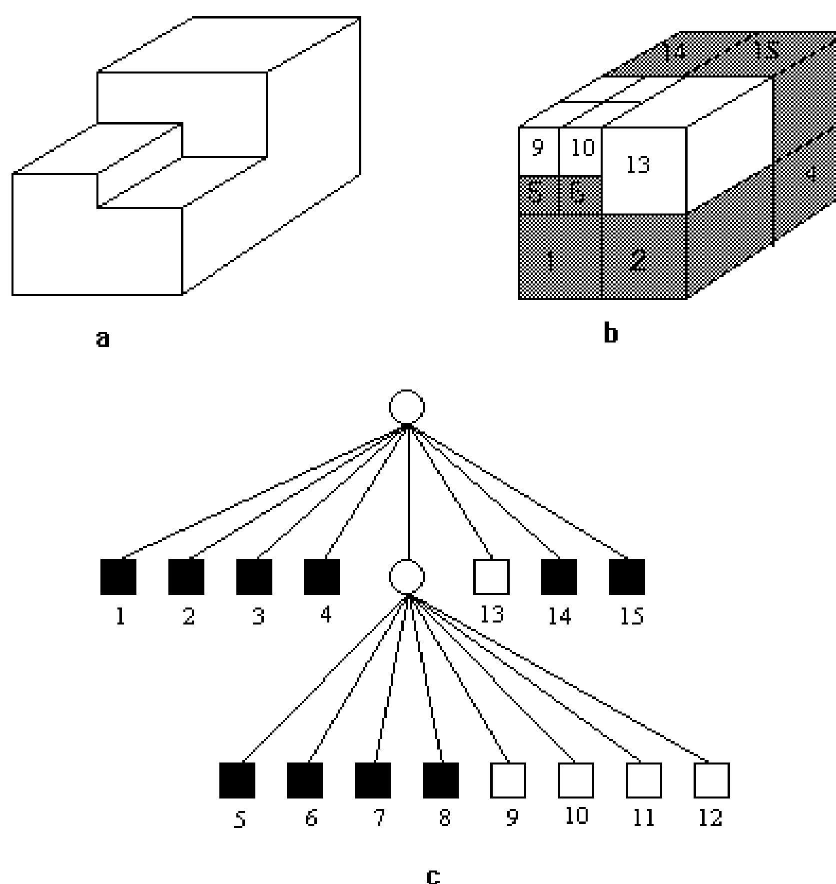

Рис. 1.6 Трехмерное изображение, октотомическое разбиение и октодерево

Принципы работы с октодеревьями почти ничем не отличаются от способов работы с квадродеревьями. Разница заключается в том, что исходными данными при построении октодеревьев являются трехмерные объекты, а не двумерные изображения. Объекты соотносятся с листовыми

узлами октодерева, которые формируются в том случае, когда объект либо целиком лежит внутри, либо целиком вне октанта, соответствующего листовому узлу. Корневой узел октодерева соответствует полной трехмерной сцене. Все узлы кроме листовых имеют по восемь дочерних узлов (октантов).

Одним из главных достоинств октодерева является возможность представления объектов почти произвольной формы вне зависимости от того выпуклый ли он, вогнутый или многосвязный. Точность представления определяется размерами наименьшего октанта разбиения. Это свойство октодеревьев находит, пожалуй, свое основное применение в приложениях построения реалистичных изображений путем трассировки лучей, при которой точность представления объектов является критической величиной для достижения качественных результатов.

С помощью октотомического дерева очень просто производятся расчёты таких величин как площадь поверхности, объем, центр масс. Октодеревья обладают всеми преимуществами и эффективностью, свойственными любой древесной структуре.

1.3 Постановка задачи

Целью данного дипломного проекта является представления воксельных данных в виде окто-дерева для ускорения визуализации блочных моделей. Разрабатываемая в рамках данного проекта система при помощи октодеревьев позволяет очень большие многополигональные модели представлять в компактном виде и работать с ними в реальном времени. Основные требования к системе:

1. Обеспечить пользователя удобными и настраиваемыми средствами для визуализации графических моделей с возможностью масштабирования и настройки их внешнего вида;
2. Обеспечить возможность сохранения и загрузки всех параметров в понятном универсальном формате;

3. Сократить время ресурсоёмких вычислений в системе и их обработки;

Требования к программному обеспечению системы:

1. Наличие операционной системы семейства Windows не ниже XP;
2. Наличие установленной платформы .NET Framework 4.0.
3. Наличие главного меню для выполнения наиболее актуальных задач;
4. Наглядный доступ к используемым функциям;
5. Удобный и продуманный графический интерфейс;
6. Наглядность и конфигурируемость представления информации.
7. Реализация собственных программных библиотек для ускоренной работы с OpenGL (распараллеливание на GPU и др.).

После постановки требований можно переходить к проектированию автоматизированной системы.

2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ, ТЕХНОЛОГИЙ И ВЫРАБОТКА ПОДХОДА К ВЫЧИСЛЕНИЯМ

В ходе постановки задачи было определено, что необходимо разработать метод представления воксельных данных в виде окто-дерева. После изучения предложенных на рынке инструментов, было решено отдать предпочтение следующему набору:

1. Платформа .NET Framework 4.0;
2. Операционная система не ниже WindowsXP (Windows Vista/Windows7);
3. Среда программирования Embarcadero RAD Studio 10.1 Berlin (язык C++);
4. Библиотека для работы с RAD Studio.

2.1. Проектирование модуля для системы недропользованя

Октодерево является древовидная структура, которая используется для разделения 3D-пространстве. Каждый узел октодерева имеет восемь граней. Наша система использует октодеревья для нескольких задач.

Октодерево – это дерево, где каждый внутренний (без дочерних элементов) узел имеет восемь граней. Каждый узел охватывает определенную область пространства, выраженную в выровненном по осям ограничивающем прямоугольнике (в виде коробки TOctreeNode). Каждый узел имеет также выбранную среднюю точку в этом поле (в качестве MiddlePoint собственности TOctreeNode класса). Этот пункт определяет три плоскости, параллельной основанию X, Y и Z осей и пересечение этой точке. Каждый дочерний элемент данного узла октодерево представляет собой одну

из восьми частей, которые создаются путем деления пространства с помощью этих трех плоскостей.

Каждый дочерний элемент, в свою очередь, может быть либо:

1. Другим внутренним узлом. Таким образом, он имеет свою среднюю точку и еще восемь дочерних элементов. Его средняя точка должна быть в пределах пространства, так как его "родители" дали ему узел.

2. Листом, который просто фактически содержит элементы, которые нужно сохранить в октодереве. Что такое "фактический пункт" зависит от того, хотят с элементов, которые необходимо вычислить столкновений с помощью этого октодерева.

Недропользоване – вид деятельности, науки направленный на изучения недр и дальнейшего их использования в качестве изучения, или добычи полезных ископаемых.

Данная работа при использовании октодеревьев позволит очень большие блочные месторождения представить в компактном виде и работать с ними в реальном времени. Тем самым заменив огромное количество вокселей

2.2. Использование среды разработки RAD Studio

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 я 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;

Большинство приложений, использующих GLScene, отрисовываются (рендерятся) в режиме реального времени. При этом время имеет большое значение. 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.

Сначала я объясню, что свет делает в GLScene. 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(Библиотéка визуáльных компонéнтов) — 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; То есть элементы пользовательского интерфейса на экране. 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, а не элементы управления на платформе, 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

Встроенная поддержка стилей

Поддержка визуальных эффектов (например, Glow, Inner Glow, Blur, например) и анимация визуальных компонентов

Из-за совместимости между платформами один и тот же исходный код может использоваться для развертывания на различных платформах, которые он поддерживает. Он изначально поддерживает 32-разрядные и 64-разрядные исполняемые файлы в Windows и 32-разрядные исполняемые файлы на MacOS и iOS, а также собственные исполняемые файлы на Android.

С момента его внедрения в XE2 во многих областях этой структуры произошло значительное улучшение, и оно активно развивается и совершенствуется. Например, разработка macOS тесно интегрирована в среду IDE, для чего требуется только Mac для развертывания. Были добавлены многочисленные компоненты, такие как датчики, сенсорный экран и GPS, что особенно полезно для тех, кто разрабатывает мобильные приложения. Значительная производительность и базовые технические усовершенствования тоже.

3. РЕАЛИЗАЦИЯ АВТОМАТИЗИРОВАННОЙ СИСТЕМЫ ВОКСЕЛИЗАЦИИ ПОЛИГОНАЛЬНЫХ МОДЕЛЕЙ

Для работы с заданными требованиями к вычислениям в автоматизированной системе необходимо реализовать следующие классы:

- Класс для определения и работы с геометрическими фигурами;
- Класс для определения и работы с координатными осями;
- Класс для определения принадлежности точек контурам;

3.1 Проектирование системы визуализации

Визуализация уровня как единой сетки очень проста, хотя и несколько неэффективна по времени. Но что делать в тех случаях, когда уровень требует большей детализации — зданий, деревьев, пещер и других, необходимых для игры объектов? А как насчет того, чтобы немного упростить разработку уровней?

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.

Как можно определить, какие полигоны видимы, не сканируя их все в каждом кадре? 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? Для дальнейших действий необходимо представить движок NodeTree. 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). Движок NodeTree достаточно универсален, поскольку может работать в двух различных режимах (отличающихся способом разделения на узлы): режиме квадродерева (quadtree) и режиме октодерева (octree). В режиме квадродерева мир (и последующие узлы) делится на четыре узла. Этот режим лучше подходит для сеток уровней, где нет значительных изменений по оси Y (высота точки просмотра меняется не сильно). Режим октодерева разделяет мир (и последующие узлы) на восемь узлов. Используйте этот режим для больших трехмерных сеток, в которых точка просмотра может располагаться в любом месте. Процесс деления продемонстрирован на рис. 3.1

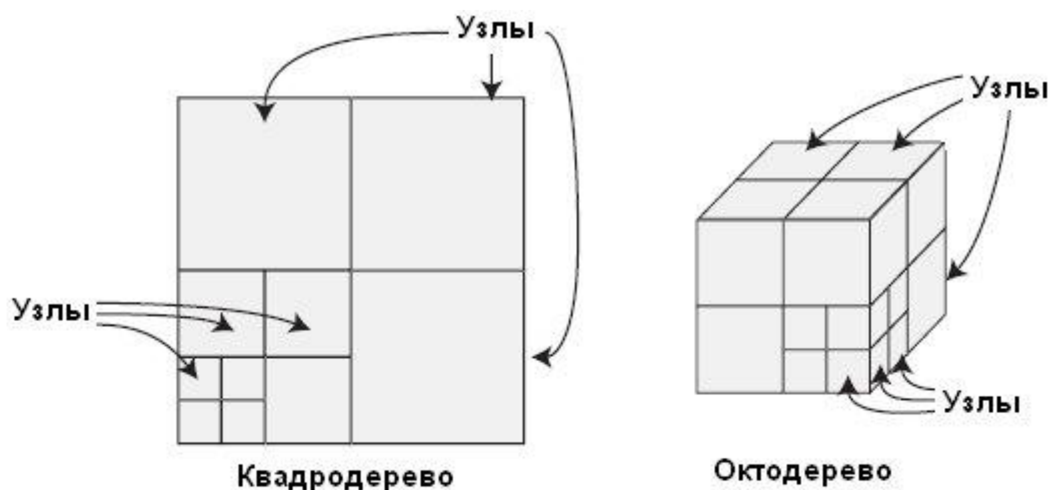

Рис. 3.1 Режим квадродерева разделяет мир (и последующие узлы) на четыре узла за раз, а режим октодерева разделяет мир (и последующие узлы) на восемь узлов за раз

Выбор используемого режима разделения зависит от разработчика. Необходимо принять во внимание сетку — там есть замок для всех необходимых действий. Если высоты вершин сеток не слишком различаются (как, например, в ландшафте), лучше подойдет режим квадродерева. С другой стороны, если сетка распространяется по всем осям (например, замок с несколькими уровнями), нужно использовать режим октодерева.

Мир (представляемый как куб, заключающий в себе все полигоны, описывающие уровень) разделяется на меньшие узлы равного размера. Квадродерево разделяет узлы в двухмерном пространстве (используя оси X и Z), а октодерево разделяет узлы в трехмерном пространстве (используя все оси).

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 связаны все другие узлы.

Вот трюк для узлов и деревьев: зная, какие полигоны содержатся в узле трехмерного пространства, вы можете 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.

Каждый полигон сетки заключается в прямоугольник (называемый ограничивающим прямоугольником, как показано на рис. 3.2.). 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 полигона может распространяться на несколько узлов.

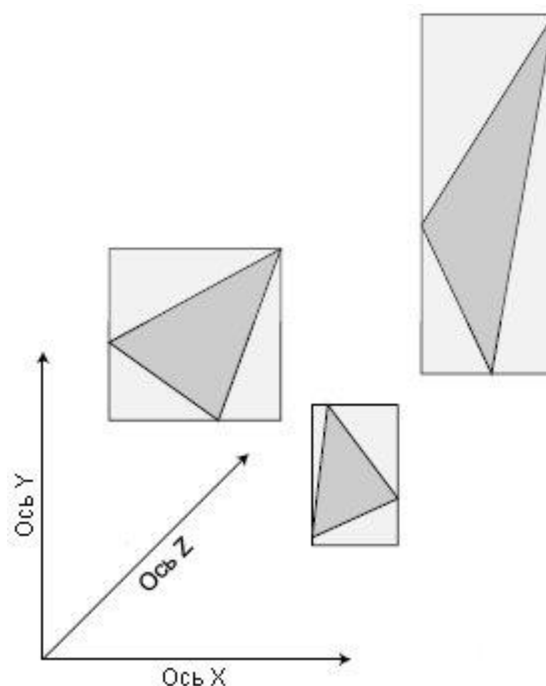

Рис. 3.2 У полигонов есть воображаемые прямоугольники окружающие их

Ограничивающие прямоугольники полезны для быстрого определения местоположения полигона в трехмерном пространстве.

В процессе группировки полигонов в узлы обращайте внимание, есть ли большие пространства между полигонами или много полигонов сосредоточено в одном месте. В последнем случае необходимо разделить узел на более мелкие подузлы и заново просканировать список полигонов, приняв во внимание новые узлы. Продолжайте этот процесс, пока все полигоны не будут разделены на достаточно маленькие группы и пока каждый узел, содержащий полигоны, не станет достаточно маленьким.

Для оптимизации древовидной структуры нужно отбрасывать все узлы не содержащие полигонов. Отбрасывание пустых узлов экономит память и позволяет быстро пересекать дерево. Экономия памяти и времени это залог успеха.

На рис. 3.3 показано несколько полигонов и мир (корневой узел), заключенный в квадрат. Квадрат разделен на меньшие узлы, а те, в свою очередь, также разделены на узлы. Каждый узел либо используется (содержит полигон или полигоны), либо не используется (не содержит полигонов).

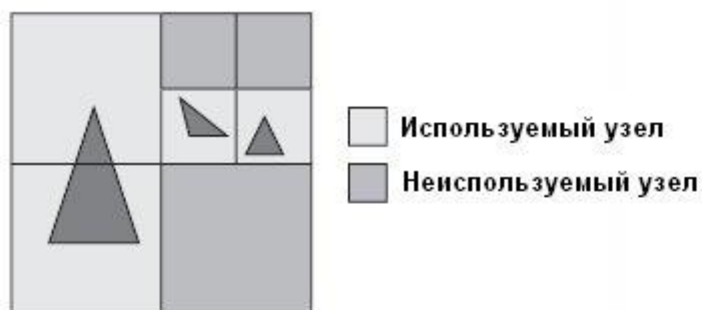

Рис. 3.3 Пример мира с несколькими полигонами, сгруппированными в узлы

Поскольку полигоны на рис. 3.3 расположены далеко друг от друга, вы можете разделить корневой узел на четыре меньших узла (создав квадродерево). Затем необходимо проверить каждый узел и продолжить

разделять большие. Для ускорения процесса можете пропустить пустые узлы. В конце концов будет получена древовидная структуру, готовая для сканирования.

На рис. 3.4 показаны обозначения элементов, которые используются в программной реализации.

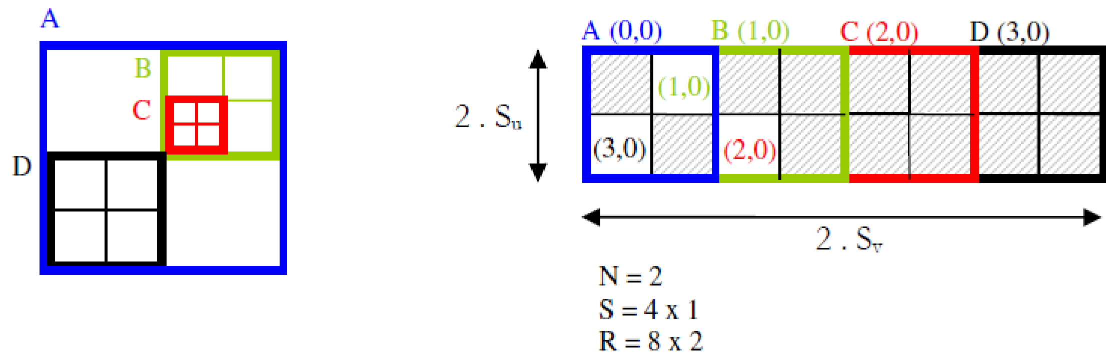

Рис. 3.4 Квадродерево и обозначения.

Заметим, что – количество косвенных сетей, хранящихся в ячейке и –

$$S = S_u \times S_v \times S_w$$
$$R = N \cdot S_u \times N \cdot S_v \times N \cdot S_w \qquad (3.1)$$

разрешение ячеек.

3.2.Реализация алгоритма визуализации с использованием структуры октодерева

Построив древовидную структуру, мы готовы визуализировать ее. Начав с корневого узла, мы выполняем проверку попадания узла в пирамиду видимого пространства. Узел считается видимым, если хотя бы одна из восьми точек (думайте о них, как об углах куба), формирующих его в трехмерном пространстве, находится внутри пирамиды видимого

пространства или сама пирамида видимого пространства находится внутри узла.

После того, как мы решили, что узел видим, выполните ту же самую проверку для его дочерних узлов (если они есть). Если у узла нет потомков, проверьте, есть ли в нем полигоны, которые надо рисовать (дочерние узлы в процессе сканирования могут рисовать одни и те же полигоны). После того, как полигоны в узле нарисованы, они отмечаются, как нарисованные, и сканирование возвращается к родительскому узлу (и любым другим оставшимся узлам, которые будут просканированы).

Разделим их на слои для обработки. Тем не менее, мы разделяем тома на разделы, содержащие несколько слоев, если они попадают в память и обрабатывают их параллельно. Первое ядро вычисляет коды куба, то есть восьмибитовые значения, кодирующие сдвиги, в противоположных направлениях. Это ядро также вычисляет пересечения между изоповерхностью и ребрами куба, если они есть. Предполагая, что четыре угла и четыре края дна куба уже обработаны, каждая нить обрабатывает только вершину v4 и ребра e4, e7 и e8 (нарисована на рисунке 3.5). Кроме того, ядро вычисляет градиент в вершине v4. Обратите внимание, что для повышения производительности мы используем конкретное ядро для первого фрагмента каждого раздела.

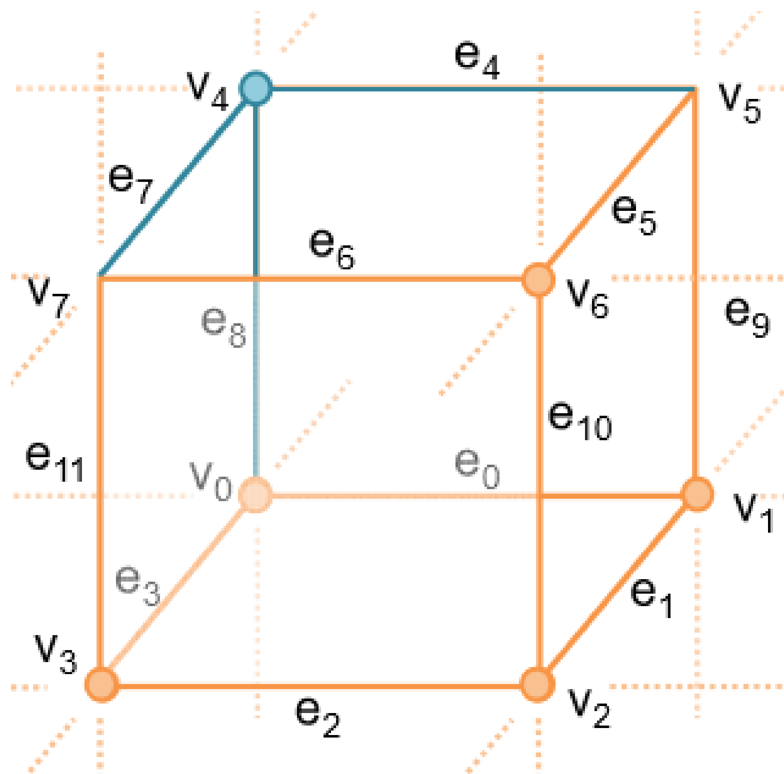

Рис.3.5 Граничные и вершинные индексы.

Второе ядро, созданное и созданное, было основано на знаниях. При этом классовые маршевые кубы смотрят таблицу, определяющую топологию поверхности. Это ядро также удаляет вырожденные треугольники и подает сетку во второй модуль, упрощая.

Разделим объем размерности dimX × dimY × dimZ на слои и срезы, K-й срез содержит все вершины с одинаковой y-координатой. K-й слой представляет собой набор всех вершин, ребер и патчей между или на k-м и (k + 1) -м срезе. На рисунке 3 показана взаимосвязь между срезами и слоями. Таким образом, объем содержит срезы от 0 до dimY -1 и слои от 0 до dimY -2. Затем мы обрабатываем слои в порядке возрастания.

Мы группируем срезы и слои в разделы при извлечении поверхности. Каждый раздел состоит из N slices и N-1 соответственно. Они обрабатываются в двух циклах и в основном из двух ядер Обратите внимание, что для вычисления нормалей поверхности из градиентов требуются два

дополнительных среза. Следовательно, перегородки перекрываются одним срезом в каждом направлении. Поскольку вершины первого среза уже были рассчитаны в предыдущем разделе, нам нужен только один дополнительный срез в каждом разделе. Это означает, что раздел с N срезами содержит только слои N-2, которые можно использовать для извлечения Кроме того, по этой причине мы используем отдельные ядра для первого и последнего срезов.

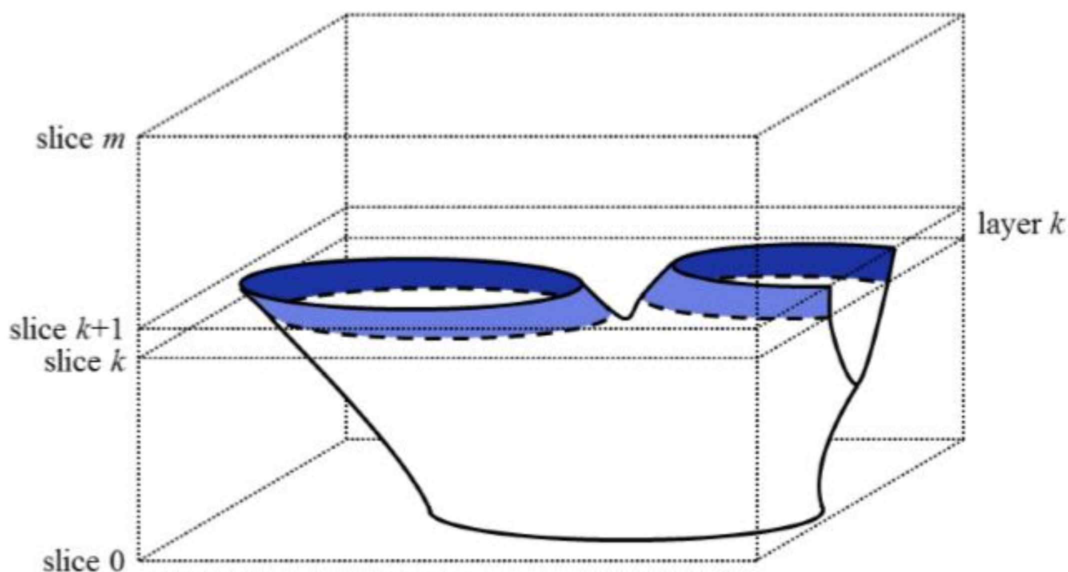

Рис.3.6 Связь между срезами и слоями.

Ядро кубического кода выполняется для каждого куба, используя размер блока потока 16 × 16 или 32 × 32 в зависимости от графического процессора. Предполагая, что четыре угла и четыре края дна куба уже обработаны, каждый поток только обрабатывает вершину v4 и ребра e4, e7 и e8 (см. Рис. 3.5). Исключением являются потоки на правой и / или передней фронтах сетки, которые должны обрабатывать дополнительные вершины. Каждый поток также вычисляет градиент вершины v4. Градиенты и кодовые коды всегда должны быть рассчитаны и для предыдущего слоя. Нить устанавливает первые четыре бита кода куба, а затем сдвигает код на четыре бита вправо в конце. Распараллеливание позволяет перечислять ребра куба не только локально, как показано на рис. 3.6, но и глобально. Это

предотвращает повторный расчет точек пересечения между краями изоповерхности и куба.

После того, как первое ядро вычислило пересечения, градиенты и коды куба, kernel_generate создает сетку, используя таблицу поиска классических маршевых кубов. Таблица поиска хранит топологию поверхности. Это означает, что и должны быть рассчитаны только пересечения ребер куба и изоповерхности, и соответствующая сетка извлекается из таблицы поиска. Результатом является набор уровней Iρ: = {v∈R3: f (v) = ρ}, уравнение которого можно упростить, вычитая изо-значение ρ из всего набора данных.

3.3 Geoblock система для недропользования

Программа Geoblock интегрирована программное обеспечение для 2D / 3D моделирования, вычислительной геометрии и визуализации пространственных данных. Программа может быть использована в области наук о Земле, особенно в таких областях, как моделирование геологии и добычи полезных ископаемых, рудных резервных оценок и прогноза минерального освобождения под измельчения и переработки полезных ископаемых операций(рис.3.7).

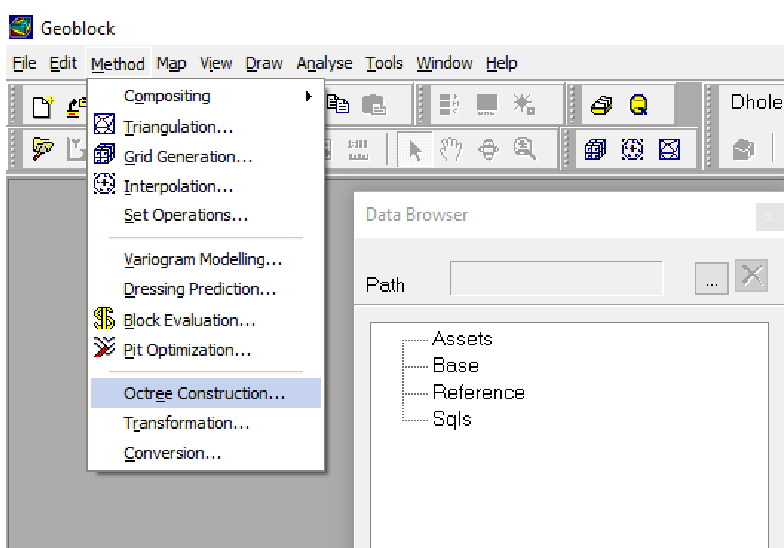

Рис 3.7 Geoblock

Программа поддерживает несколько типов пространственных наборов данных для научных вычислений и моделирования: Точки, высверливание отверстий или Скважины, многоугольники, TIN (Triangulated нерегулярной сети), Solids, Сетка и сетки.

Генерация сеток и сеток в 2D / 3D реализована. Подпрограммы для данных разведки и обработки скважинных включают в себя: анализ и статистические данные, данные преобразования; композитинг образцы буровых скважин внутри скамьи и интервалов классов; Расчет координаты проб по скважинам и разведочных линиям; предсказание минеральных освободительных явлений при шлифовании и параметров руд, моделирования горных процессов в шахтах и т.д.

Методы интерполяции: Inverse Distance, Линейная, ближайшая точка, Кригинг, Природные Соседей и полиномиальной регрессии.

Построенные сетки и блочные модели могут быть использованы для оптимизации открытого карьера и рудника планирования. Запасы месторождения для любого типа руды или рода можно рассчитать различными методами с использованием пространственных компьютерных моделей.

Комбинированные наборы данных могут быть организованы в коллекции модели и отображаются в окне карты программы в виде комбинации контуров, каркасные или блок моделей одновременно.

Некоторые пространственные модели могут быть визуализированы одновременно в GeoScene окне программы с помощью менеджера проекта, как показано на Рис. 3.8

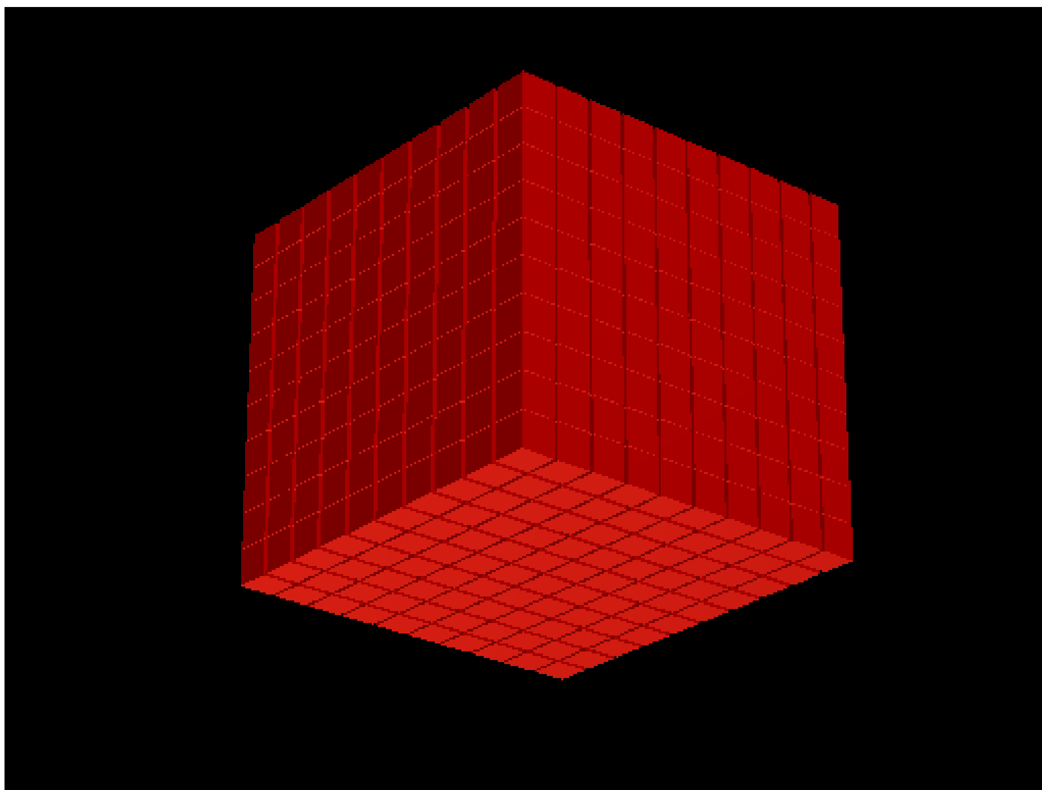

Рис. 3.8 Визуализация модели в Geoblock

Базы данных хранятся в PostgreSQL + PostGIS. Пространственные данные и графические объекты могут быть экспортированы / импортированы в DXF (AutoCAD), MIF / MID (MapInfo), GRD (Surfer, ArcInfo) и некоторые другие форматы.

4. АПРОБАЦИЯ МЕТОДА ПРЕДСТАВЛЕНИЯ ВОКСЕЛЬНЫХ ДАННЫХ В ВИДЕ ОКТО-ДЕРЕВА ДЛЯ УСКОРЕНИЯ ВИЗУАЛИЗАЦИИ БЛОЧНЫХ МОДЕЛЕЙ

После запуска приложения и выбора модели и расставления параметров визуализируется октодерево. Исходные объекты и воксели отображается следующим образом (рис. 4.1):

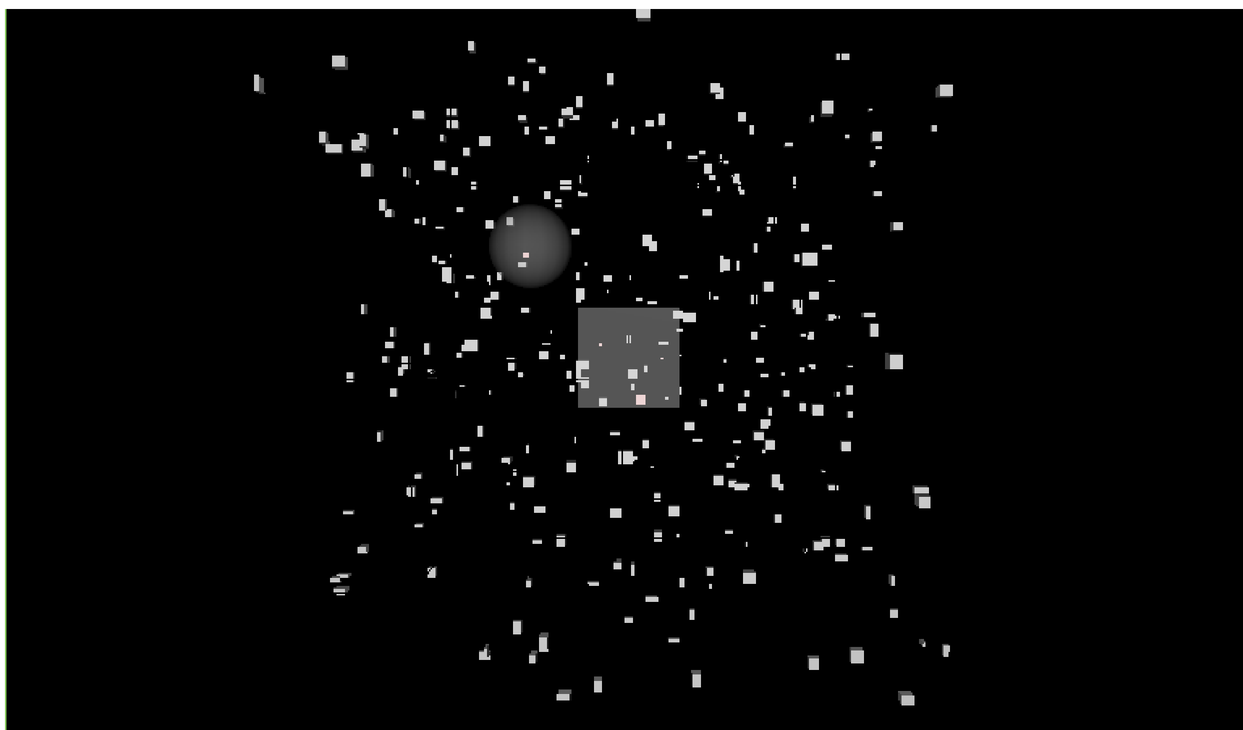

Рис. 4.1 Отображение вокселей и объектов.

На рис. 4.2 следующим образом отображена модель и примененные на нем октодеревья

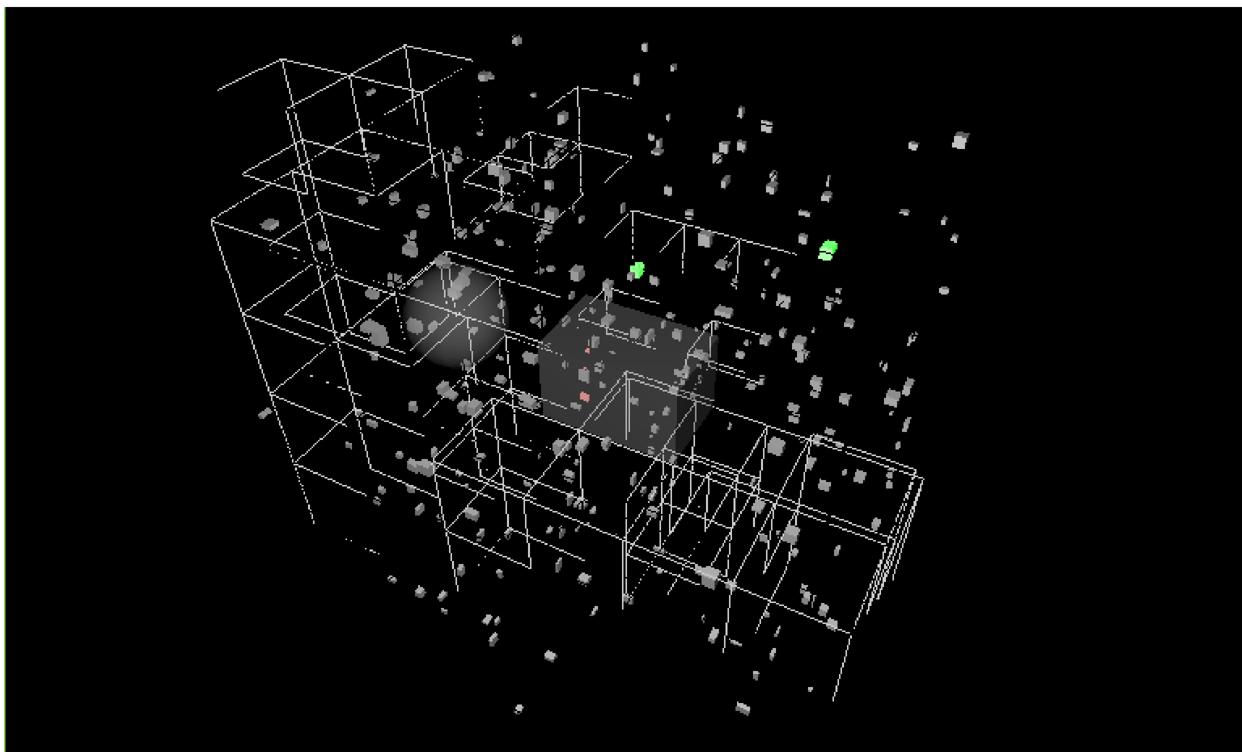

Рис. 4.2 Октодерево.

Также можно увеличить максимальное количество отображения октодеревьев рис4.3

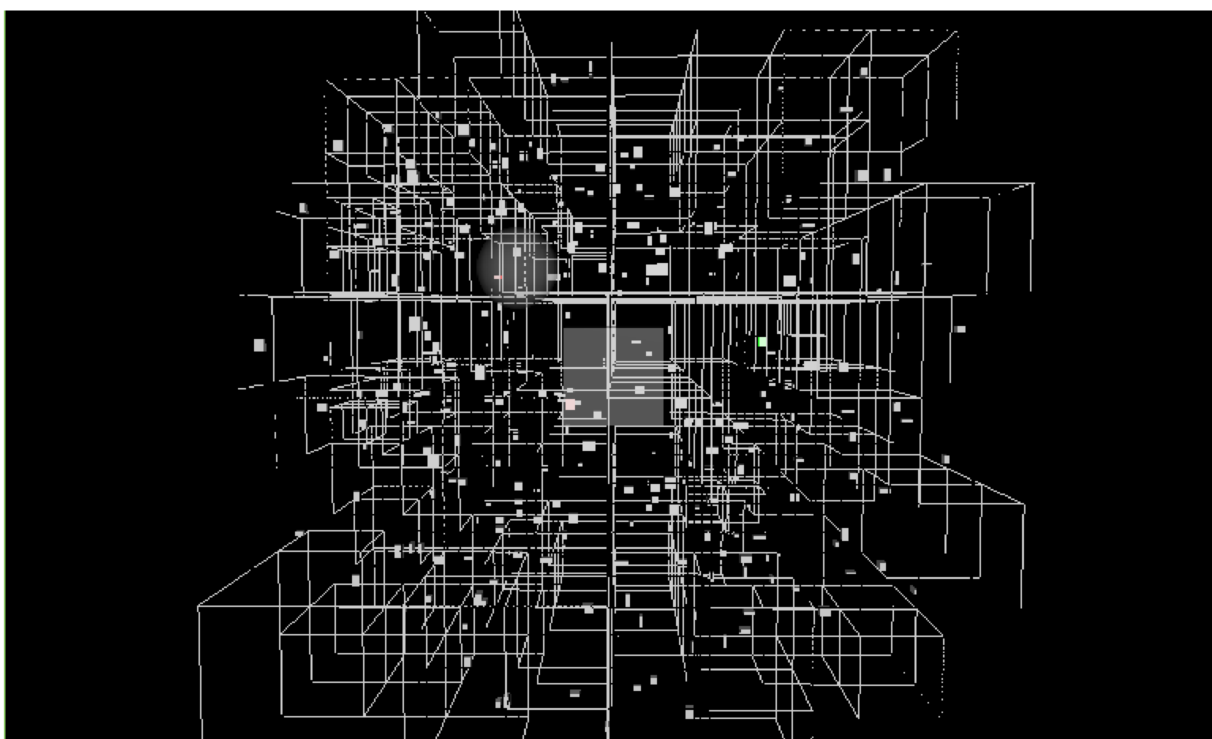

Рис 4.3Увеличение количества отображения октодеревьев

Пример отображения другой модели (рис. 4.4.) вместе с внешними гранями

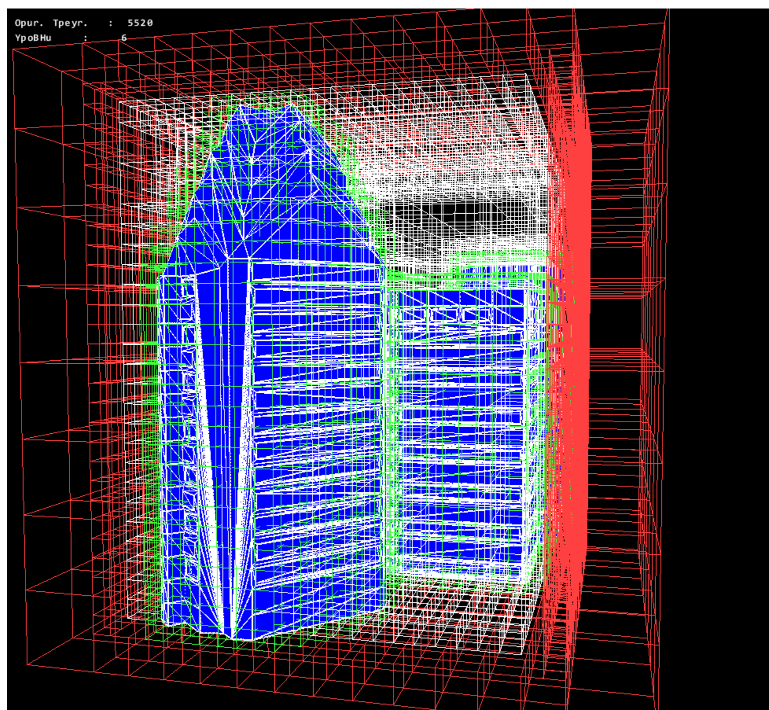

Рис. 4.4 Внешние грани модели здания.

Пример построения воксельной модели (рис. 4.5.):

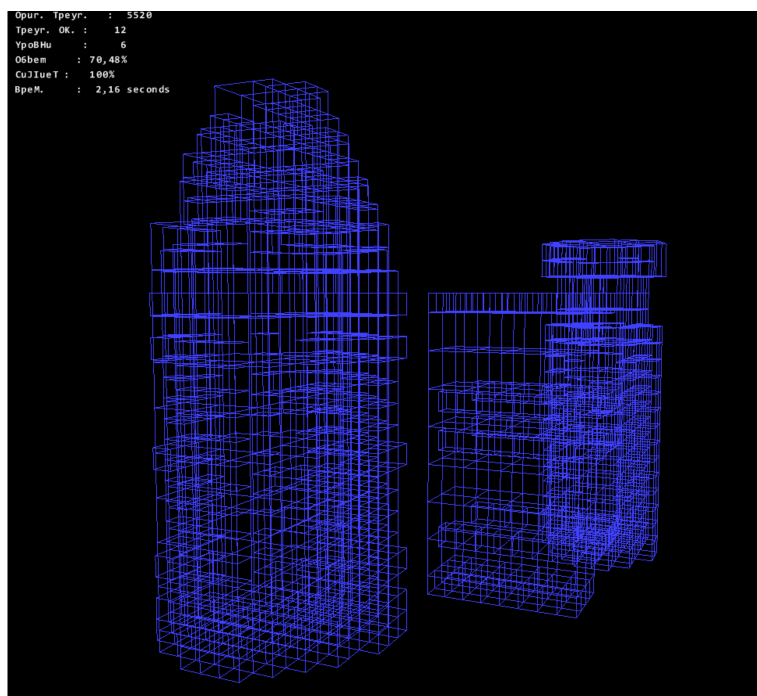

Рис. 4.5 Воксельная модель.

После просмотра результата можно сделать вывод, что приложение для визуализации октодерева выполняет все свои функции корректно.

ЗАКЛЮЧЕНИЕ

В данной проекте был разработан GPU-совместимый движок для визуализации октодеревьев и сопутствующих элементов. Этот движок 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 2D текстуры весьма предпочтительны в некоторых ситуациях и было показано, как текстура октодерева может быть динамически преобразована в 2D текстуру без артефактов.

Октодеревья – достаточно распространённые структуры данных, широко используемые в компьютерной графике. Они представляют собой удобный способ хранения информации о непараметризованной сетке, и вообще всего пространства, поэтому об их более эффективном применении в будущем сомневаться не приходится. Многие приложения, которые занимаются визуализацией данных, в том числе и игры, могут извлечь выгоду из аппаратной реализации данного подхода.

В будущей работе важным вопросом является улучшение качества распознавания коллизий (столкновений), т.е. достижение отделимости, надежности, точности и минимальности одновременно. Ожидания следующие: будет реализована операция смешивания текстур с плавающей точкой для улучшения эффективности алгоритма. В тестах будут загружены все сцены в полном разрешении на GPU памяти, в то время как лишь

небольшая часть будет необходима для показывания какого-либо одного изображения за счет блокирования, и разрешение будет падать вместе с расстоянием. Система уже поддерживает потоковое преобразование в зависимости от расстояния до камеры.

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 ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Н.Н.Калиткин, И.П.Пошивайло. О вычислении простых и кратных корнейнелинейного 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. Yaohong Dennis Jiang An Interactive 3-D Mine Modeling, Visualization and Information System Queen's University Kingston, Ont., Canada June, 1998

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.

26. Martz, Paul. 2007. OpenSceneGraph Quick Start Guide: A Quick Introduction to the Cross-Platform Open Source Scene Graph API. Skew Matrix Software.

27. Rost, Randi J. 2005. OpenGL(R) Shading Language (2nd Edition). Addison-Wesley Professional.

28. Segal, Mark, & Akeley, Kurt. 2006 (December). The OpenGL(R) Graphics System: A Specification (Version 2.1). Tech. rept. Khronos Group.

29. Shreiner, Dave, Woo, Mason, Neider, Jackie, & Davis, Tom. 2005. OpenGL(R) Programming Guide: The O-cial Guide to Learning OpenGL(R), Version 2 (5th Edition). Addison-Wesley Professional.

30. Zhang, Long, Chen, Wei, Ebert, David S., & Peng, Qunsheng. 2007. Conservative voxelization. The Visual Computer, 23(9), 783-792

ПРИЛОЖЕНИЕ

```
//---------------------------------------------------------------
//    Файл:     octreed.pas
//    Описание: debug для Octree
//---------------------------------------------------------------
unit octreed;

interface

uses
  Winapi.OpenGL,
  Octree;



//---------------------------------------------------------------
// прототипы функций
//---------------------------------------------------------------
procedure Octree_AddDebugRectangle( const center: vec3_t; width, height,
depth: single );
procedure Octree_RenderDebugLines;

//=============================================================
==============
implementation
//=============================================================
==============

uses
```

```
  littlemath;


//--------------------------------------------------------------
// типы, константы и глобальные переменные
//--------------------------------------------------------------
var
  g_lines: array[0..20000] of vec3_t;
  g_count: longword = 0;


//--------------------------------------------------------------
// добавляет точку в массив
//--------------------------------------------------------------
procedure AddPoint( const point: vec3_t );
begin
  g_lines[g_count] := point;
  inc( g_count );
end;


//--------------------------------------------------------------
// добавляет куб в массив
//--------------------------------------------------------------
procedure Octree_AddDebugRectangle( const center: vec3_t; width, height,
depth: single );
var
  points: array[0..7] of vec3_t;
  i:     integer;
begin
```

```
width  := width  / 2;
height := height / 2;
depth  := depth  / 2;

points[0] := Vector( center.x - width, center.y + height, center.z + depth );
points[1] := Vector( center.x - width, center.y + height, center.z - depth );
points[2] := Vector( center.x + width, center.y + height, center.z - depth );
points[3] := Vector( center.x + width, center.y + height, center.z + depth );

points[4] := Vector( center.x - width, center.y - height, center.z + depth );
points[5] := Vector( center.x - width, center.y - height, center.z - depth );
points[6] := Vector( center.x + width, center.y - height, center.z - depth );
points[7] := Vector( center.x + width, center.y - height, center.z + depth );

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

AddPoint( points[0] );
AddPoint( points[3] );
AddPoint( points[1] );
AddPoint( points[2] );

AddPoint( points[0] );
AddPoint( points[1] );
AddPoint( points[3] );
AddPoint( points[2] );

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

AddPoint( points[4] );
AddPoint( points[7] );
```

```
  AddPoint( points[5] );
  AddPoint( points[6] );

  AddPoint( points[4] );
  AddPoint( points[5] );
  AddPoint( points[7] );
  AddPoint( points[6] );

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

  AddPoint( points[0] );
  AddPoint( points[4] );
  AddPoint( points[1] );
  AddPoint( points[5] );

  AddPoint( points[2] );
  AddPoint( points[6] );
  AddPoint( points[3] );
  AddPoint( points[7] );
end;


//----------------------------------------------------------------
// рисует грани кубов октарного дерева
//----------------------------------------------------------------
procedure Octree_RenderDebugLines;
var
  i: integer;
begin
  glColor3f( 1.0, 1.0, 0.0 );
```

```
  glDisable( GL_TEXTURE_2D );

  glBegin( GL_LINES );
    for i := 0 to g_count - 1 do glVertex3fv( @g_lines[i] );
  glEnd();
end;

end.
//-----------------------------------------------------------------
//   Файл:     octree.pas
//   Описание: Реализация октарного дерева
//-----------------------------------------------------------------
unit Octree;

interface

uses
  Winapi.Windows,
  Winapi.OpenGL;

//-----------------------------------------------------------------
// типы, константы и глобальные переменные
//-----------------------------------------------------------------
type
  pvec3_t = ^vec3_t;
  vec3_t = record
    x: single;
    y: single;
    z: single;
  end;
```

```
vec3_t8 = array [0..7] of vec3_t;

uv_t = record
  u: single;
  v: single;
end;

pface_t = ^face_t;
face_t = record
  vai:    array[0..2] of longword; // vertex attribute indices
  tex_id: GLuint;
end;

renderportion_t = record
  tex_id: GLuint;           // номер устанавливаемой текстуры
  count:  longword;         // количество индексов вершин
  vai:    array of longword; // vertex attribute indices
end;

pnode_t = ^node_t;
node_t = record
  center:         vec3_t;                  // центр узла
  size:           single;                  // размер стороны узла
  cubeverts:      vec3_t8;                 // 8 вершин куба

  num_ports:      longword;                // количество порций геометрии
  ports:          array of renderportion_t; // массив порций
  num_intsects:   longword;                // количество дробных граней
  intsect_indices: array of longword;      // индексы дробных граней
```

```
    is_div:          boolean;                  // содержит ли подузлы?
    sub_nodes:       array [0..7] of pnode_t;  // 8 дочерних подузлов
  end;

var
  g_totalleafdraw: longword = 0; // кол-во выведенных листов




//-----------------------------------------------------------------
// прототипы функций
//-----------------------------------------------------------------
function Octree_Create(): boolean;
procedure Octree_Remove();
procedure Octree_Render();

//=====================================================================
implementation
//=====================================================================

uses
  littlemath,
  octreed,
  camera;
```

```
procedure CreateNode( node: pnode_t; var indices: array of longword;
num_infaces: longword ); forward;

//----------------------------------------------------------------
// типы, константы и глобальные переменные
//----------------------------------------------------------------
const
  MAX_FACES = 300; // пороговое количество граней, которые могут
                // находится в одном листе

  FILE_NAME = 'model\kitchen.mdl';
  NUM_VERTS = 4476;
  NUM_FACES = 3458;

var
  g_root:     pnode_t = nil;

  g_verts:     array of vec3_t = nil;
  g_texcoords: array of uv_t   = nil;
  g_faces:     array of face_t = nil;

  g_maxintsects:        longword = 0;             // максимально кол-во дробных
граней в листе
  g_intsectindices:     array of longword = nil; // индексы дробных граней
выводимого листа

  g_intsectbuffer:      array of longword = nil; // буфер уже отрисованных в
кадре дробных граней
  g_intsectbuffersize: longword = 0;            // размер буфера
```

```
  g_numrenderintsects: longword = 0;           // кол-во отрисованных в кадре дробных граней



//-----------------------------------------------------------------
// загружает данные из модели
//-----------------------------------------------------------------
function LoadModel(): boolean;
var
  i: integer;
  f: file;
begin
  result := false;

  // массивы для вершинных атрибутов
  setlength( g_verts, NUM_VERTS );
  setlength( g_texcoords, NUM_VERTS );

  setlength( g_faces, NUM_FACES );

  assignfile( f, FILE_NAME );
  reset( f, 1 );

  // считываем вершинные атрибуты
  for i := 0 to NUM_VERTS - 1 do
  begin
    blockread( f, g_verts[i], sizeof( vec3_t ) );
    blockread( f, g_texcoords[i], sizeof( uv_t ) );
  end;
```

```
  // считываем грани
  for i := 0 to NUM_FACES - 1 do
  begin
    blockread( f, g_faces[i], sizeof( face_t ) );
  end;

  close( f );

  result := true;
end;




//------------------------------------------------------------
// выделяет память для структуры node_t
//------------------------------------------------------------
procedure NewNode( var node: pnode_t );
begin
  new( node );
  ZeroMemory( node, sizeof( node_t ) );
end;

//------------------------------------------------------------
// удаляет из памяти структуру node_t
//------------------------------------------------------------
procedure DeleteNode( node: pnode_t );
var
  i, j: integer;
begin
```

```
  if node^.is_div then
  for i := 0 to 7 do
  begin
    if node^.sub_nodes[i] <> nil then
      DeleteNode( node^.sub_nodes[i] ); // рекурсивный вызов
  end;

  for j := 0 to node^.num_ports - 1 do node^.ports[j].vai := nil;
  node^.ports           := nil;
  node^.intsect_indices := nil;

  dispose( node );
end;

//-----------------------------------------------------------------
// рассчитывает центр и размер куба, описывающего всю сцену
//-----------------------------------------------------------------
procedure GetRootBBox( node: pnode_t; var verts: array of vec3_t; num_verts:
longword );
var
  max: vec3_t;
  min: vec3_t;
  v:   ^vec3_t;

  i: integer;
begin
  max := verts[0];
  min := verts[0];

  // получаем максимальное и минимальное
```

```
// значения x, y, z
for i := 0 to num_verts - 1 do
begin
  v := @verts[i]; // сократим код

  if v^.x > max.x then
    max.x := v^.x;
  if v^.y > max.y then
    max.y := v^.y;
  if v^.z > max.z then
    max.z := v^.z;

  if v^.x < min.x then
    min.x := v^.x;
  if v^.y < min.y then
    min.y := v^.y;
  if v^.z < min.z then
    min.z := v^.z;
end;

// получаем центр сцены
node^.center.x := (max.x + min.x) / 2;
node^.center.y := (max.y + min.y) / 2;
node^.center.z := (max.z + min.z) / 2;

// нам нужен куб, а не параллелепипед
node^.size := max.x - min.x;
if max.y - min.y > node^.size then
   node^.size := max.y - min.y;
if max.z - min.z > node^.size then
```

```
    node^.size := max.z - min.z;
end;

//------------------------------------------------------------------
// возвращает центр нового подузла
// center  - центр родительского узла
// size    - размер родительского узла
// node_id - идентификатор подузла
//------------------------------------------------------------------
function GetNewNodeCenter( const center: vec3_t; size: single; node_id: byte ): vec3_t;
begin
  size := size / 4;

  case node_id of
    // top_left_front
    0: result := Vector( center.x - size, center.y + size, center.z + size );
    // top_left_back
    1: result := Vector( center.x - size, center.y + size, center.z - size );
    // top_right_back
    2: result := Vector( center.x + size, center.y + size, center.z - size );
    // top_right_front
    3: result := Vector( center.x + size, center.y + size, center.z + size );
    // bottom_left_front
    4: result := Vector( center.x - size, center.y - size, center.z + size );
    // bottom_left_back
    5: result := Vector( center.x - size, center.y - size, center.z - size );
    // bottom_right_back
    6: result := Vector( center.x + size, center.y - size, center.z - size );
    // bottom_right_front
```

```
    7: result := Vector( center.x + size, center.y - size, center.z + size );
  end;
end;

//----------------------------------------------------------------
// вычисляем координаты вершин куба узла, они требуются при работе с
// viewing frustum
//----------------------------------------------------------------
procedure GetNewNodeCubeVerts( node: pnode_t );
var
  x, y, z, size: single;
begin
  // опираемся на координаты центра и размер узла
  x    := node^.center.x;
  y    := node^.center.y;
  z    := node^.center.z;
  size := node^.size / 2;

  node^.cubeverts[0].x := x - size;
  node^.cubeverts[0].y := y - size;
  node^.cubeverts[0].z := z - size;

  node^.cubeverts[1].x := x + size;
  node^.cubeverts[1].y := y - size;
  node^.cubeverts[1].z := z - size;

  node^.cubeverts[2].x := x - size;
  node^.cubeverts[2].y := y + size;
  node^.cubeverts[2].z := z - size;
```

```
  node^.cubeverts[3].x := x + size;
  node^.cubeverts[3].y := y + size;
  node^.cubeverts[3].z := z - size;

  node^.cubeverts[4].x := x - size;
  node^.cubeverts[4].y := y - size;
  node^.cubeverts[4].z := z + size;

  node^.cubeverts[5].x := x + size;
  node^.cubeverts[5].y := y - size;
  node^.cubeverts[5].z := z + size;

  node^.cubeverts[6].x := x - size;
  node^.cubeverts[6].y := y + size;
  node^.cubeverts[6].z := z + size;

  node^.cubeverts[7].x := x + size;
  node^.cubeverts[7].y := y + size;
  node^.cubeverts[7].z := z + size;
end;

//---------------------------------------------------------------
// записывает в min и max крайние вершины куба
//---------------------------------------------------------------
procedure GetLeafMinMaxVertex( leaf: pnode_t; min, max: pvec3_t );
var
  i: integer;
begin
  min^ := leaf^.cubeverts[0];
  max^ := leaf^.cubeverts[0];
```

```
  for i := 0 to 7 do  // 1 to 7
  begin
    // ближняя к началу системы координат вершина
    if leaf^.cubeverts[i].x < min^.x then
      min^.x := leaf^.cubeverts[i].x;

    if leaf^.cubeverts[i].y < min^.y then
      min^.y := leaf^.cubeverts[i].y;

    if leaf^.cubeverts[i].z < min^.z then
      min^.z := leaf^.cubeverts[i].z;

    // дальняя от начала системы координат вершина
    if leaf^.cubeverts[i].x > max^.x then
      max^.x := leaf^.cubeverts[i].x;

    if leaf^.cubeverts[i].y > max^.y then
      max^.y := leaf^.cubeverts[i].y;

    if leaf^.cubeverts[i].z > max^.z then
      max^.z := leaf^.cubeverts[i].z;
  end;
end;

//---------------------------------------------------------------
// взвращает true, если три вершины v1, v2, v3 принадлежат листу
//---------------------------------------------------------------
function WholeFaceInLeaf( const min, max, v1, v2, v3: vec3_t ): boolean;
begin
```

```
  result := false;

  if (v1.x < min.x) or
     (v1.y < min.y) or
     (v1.z < min.z) or
     (v1.x > max.x) or
     (v1.y > max.y) or
     (v1.z > max.z) then exit;

  if (v2.x < min.x) or
     (v2.y < min.y) or
     (v2.z < min.z) or
     (v2.x > max.x) or
     (v2.y > max.y) or
     (v2.z > max.z) then exit;

  if (v3.x < min.x) or
     (v3.y < min.y) or
     (v3.z < min.z) or
     (v3.x > max.x) or
     (v3.y > max.y) or
     (v3.z > max.z) then exit;

  result := true;
end;

//-------------------------------------------------------------
// сортирует треугольники по текстурам, используя алгоритм Хоара
//-------------------------------------------------------------
procedure QSortFaces( var indices: array of longword; left, right: longword );
```

```
var
  i, j: integer;
  c, temp: longword;
begin
  i := left; j := right;
  c := indices[(i + j) div 2]; // выбор компаранда - здесь "в лоб", но
                              // можно легко оптимизировать
  repeat
    while (g_faces[indices[i]].tex_id < g_faces[c].tex_id) and (i < right) do
inc( i );
    while (g_faces[c].tex_id < g_faces[indices[j]].tex_id) and (j > left) do
dec( j );

    if i <= j then
    begin
      // переставляем индексы граней
      if g_faces[indices[i]].tex_id <> g_faces[indices[j]].tex_id then
      begin
        temp       := indices[i];
        indices[i] := indices[j];
        indices[j] := temp;
      end;

      inc( i ); dec( j );
    end;
  until i > j;

  if left < j then
     QSortFaces( indices, left, j );
```

```
  if i < right then
    QSortFaces( indices, i, right );
end;

//--------------------------------------------------------------
// возвращает количество порций для рендеринга
//--------------------------------------------------------------
function GetPortsCount( var indices: array of longword; num_faces: longword ): longword;
var
  i: integer;
begin
  result := 1;

  for i := 0 to num_faces - 2 do // - 2 - потому что i + 1
  begin
    if g_faces[indices[i]].tex_id <> g_faces[indices[i + 1]].tex_id then inc( result );
  end;
end;



//--------------------------------------------------------------
// формируем куски геометрии, indices - отсортированный массив граней
//--------------------------------------------------------------
procedure InitPorts( var ports: array of renderportion_t; num_ports: longword;
                  var indices: array of longword; num_faces: longword );
var
  i, j, k, basei: integer;
  index:     integer;
```

```
  face_index: longword;
begin
  i := 0; basei := 0;

  if num_ports = 1 then
  begin
    ports[0].tex_id := g_faces[indices[0]].tex_id;
    ports[0].count  := num_faces * 3;
    setlength( ports[0].vai, ports[0].count );

    j := 0;
    for k := 0 to num_faces - 1 do // все грани
    begin
      face_index := indices[k];

      ports[0].vai[j    ] := g_faces[face_index].vai[0];
      ports[0].vai[j + 1] := g_faces[face_index].vai[1];
      ports[0].vai[j + 2] := g_faces[face_index].vai[2];

      inc( j, 3 );
    end;
  end
  else
  for index := 0 to num_ports - 1 do
  begin
    // до границы, i не переступает за границу
    while (i < num_faces - 1) and
        (g_faces[indices[i]].tex_id = g_faces[indices[i + 1]].tex_id) do inc( i );
```

```
    ports[index].tex_id := g_faces[indices[i]].tex_id; // значение последнего
одинакового элемента
    ports[index].count  := (i - basei + 1) * 3;
    setlength( ports[index].vai, ports[index].count );

    j := 0;
    for k := basei to i do
    begin
      face_index := indices[k];

      ports[index].vai[j    ] := g_faces[face_index].vai[0];
      ports[index].vai[j + 1] := g_faces[face_index].vai[1];
      ports[index].vai[j + 2] := g_faces[face_index].vai[2];

      inc( j, 3 );
    end;

    inc( i ); basei := i;
  end;
end;

//---------------------------------------------------------------
// присваивает вершины конечному листу
//---------------------------------------------------------------
procedure AssignVerticesToLeaf( node: pnode_t; var indices: array of
longword;
                                num_faces: longword );
var
  whole_indices: array of longword;   // индексы целиком входящих граней
  intsect_indices: array of longword; // индексы дробных граней
```

```
  num_wholes:    longword;
  num_intsects:  longword;

  min, max:      vec3_t;
  v1, v2, v3:    vec3_t;

  i, j: integer;
begin
  node^.is_div := false;
  setlength( whole_indices, num_faces );
  setlength( intsect_indices, num_faces );

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

  GetLeafMinMaxVertex( node, @min, @max );

  // разносим грани по двум массивам
  num_wholes := 0;
  num_intsects := 0;
  for i := 0 to num_faces - 1 do
  begin
    // получаем три вершины грани
    v1 := g_verts[g_faces[indices[i]].vai[0]];
    v2 := g_verts[g_faces[indices[i]].vai[1]];
    v3 := g_verts[g_faces[indices[i]].vai[2]];

    if WholeFaceInLeaf( min, max, v1, v2, v3 ) then
    begin
      whole_indices[num_wholes] := indices[i];
      inc( num_wholes );
```

```
    end
    else
    begin
      intsect_indices[num_intsects] := indices[i];
      inc( num_intsects );
    end;
  end;

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

  if num_wholes = 0 then node^.num_ports := 0
  else
  begin
    setlength( whole_indices, num_wholes );        // урезаем до нужной длины
    QSortFaces( whole_indices, 0, num_wholes - 1 ); // сортируем по
текстурам

    node^.num_ports := GetPortsCount( whole_indices, num_wholes );
    setlength( node^.ports, node^.num_ports );
    InitPorts( node^.ports, node^.num_ports, whole_indices, num_wholes );
  end;

  if num_intsects = 0 then node^.num_intsects := 0
  else
  begin
    setlength( intsect_indices, num_intsects );          // урезаем до нужной
длины
    QSortFaces( intsect_indices, 0, num_intsects - 1 ); // сортируем по
текстурам
```

```
    setlength( node^.intsect_indices, num_intsects );
    CopyMemory( node^.intsect_indices, intsect_indices, num_intsects * sizeof( longword ) );
    node^.num_intsects := num_intsects;

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

    inc( g_intsectbuffersize, num_intsects );

    if g_maxintsects < num_intsects then
      g_maxintsects := num_intsects;
  end;

  whole_indices := nil;
  intsect_indices := nil;
end;

//-------------------------------------------------------------
// создаёт новый подузел в дереве
//-------------------------------------------------------------
procedure CreateNewNode( node: pnode_t;
                    var indices: array of longword; num_parentnodefaces: longword;
                    var faces: array of boolean; num_faces: longword;
                    center: vec3_t; size: single; node_id: byte );
var
  node_indices: array of longword; // индексы граней создаваемого подузла
  i, j: integer;
begin
  setlength( node_indices, num_faces );
```

```
  j := 0;
  for i := 0 to num_parentnodefaces - 1 do
  begin
    if faces[i] = true then
    begin
      node_indices[j] := indices[i];
      inc( j );
    end;
  end;

  // j = num_faces

  NewNode( node^.sub_nodes[node_id] );

  // получаем центр, размер и 8 вершин куба
  node^.sub_nodes[node_id].center := GetNewNodeCenter( center, size,
node_id );
  node^.sub_nodes[node_id].size   := size / 2;
  GetNewNodeCubeVerts( node^.sub_nodes[node_id] );

  CreateNode( node^.sub_nodes[node_id], node_indices, num_faces );
  node_indices := nil;
end;

//---------------------------------------------------------------
// создаёт новый узел в дереве
//---------------------------------------------------------------
procedure CreateNode( node: pnode_t; var indices: array of longword;
num_infaces: longword );
```

```
var
  center:    vec3_t;
  size:      single;

  faces:     array [0..7] of array of boolean;
  num_faces: array [0..7] of longword;
  vertex:    ^vec3_t;

  i, j: integer;
  ID:   byte;
begin
  // сократим код
  center := node^.center;
  size   := node^.size;

  Octree_AddDebugRectangle( center, size, size, size );

  if num_infaces <= MAX_FACES then
    AssignVerticesToLeaf( node, indices, num_infaces )
  else
  begin
    node^.is_div := true;

    for ID := 0 to 7 do
    begin
      setlength( faces[ID], num_infaces );
      num_faces[ID] := 0;
    end;

    // Перебираем весь массив вершин и выясняем, в какой из 8 дочерних
```

```
// узлов попадает каждая вершина. Грань, содержащая эту вершину, целиком
// относится к узлу. Иногда часть грани может лежать в одном узле - а другая
// часть - в соседнем. В этом случае одну и ту же грань придётся отнести к
// разным узлам, это - дробная грань.
for i := 0 to num_infaces - 1 do
for j := 0 to 2 do
begin
  vertex := @g_verts[g_faces[indices[i]].vai[j]]; // сократим код

  // top
  if vertex^.y >= center.y then
  begin
    if vertex^.x <= center.x then
      if vertex^.z >= center.z then
        faces[0][i] := true    // left_front
      else
        faces[1][i] := true    // left_back

    else
    if vertex^.x >= center.x then
      if vertex^.z <= center.z then
        faces[2][i] := true    // right_back
      else
        faces[3][i] := true;   // right_front
  end
  else
  // bottom
```

```
  if vertex^.y < center.y then
  begin
   if vertex^.x <= center.x then
     if vertex^.z >= center.z then
       faces[4][i] := true    // left_front
     else
       faces[5][i] := true    // left_back
   else
   if vertex^.x >= center.x then
     if vertex^.z <= center.z then
       faces[6][i] := true    // right_back
     else
       faces[7][i] := true;   // right_front
  end;
 end;

 for ID := 0 to 7 do              // перебираем все 8 узлов
 for i := 0 to num_infaces - 1 do  // для каждого узла перебираем все грани
 begin                          // если
  if faces[ID][i] then           // j-тый узел содержит i-тую грань,
    inc( num_faces[ID] );        // увеличиваем счётчик граней j-того узла
 end;

 // делим этот узел на подузлы
 for ID := 0 to 7 do
 begin
  if num_faces[ID] > 0 then
    CreateNewNode( node, indices, num_infaces, faces[ID], num_faces[ID],
center, size, ID );
```

```
    faces[ID] := nil;
   end;
  end;
end;




//-----------------------------------------------------------------
// возвращает true, если грань уже была отрисована
//-----------------------------------------------------------------
function FaceAlreadyRender( face_index: longword ): boolean;
var
  i: integer;
begin
  result := false;
  if g_numrenderintsects = 0 then
     exit;

  for i := 0 to g_numrenderintsects - 1 do
  begin
    if g_intsectbuffer[i] = face_index then
    begin
      result := true;
      exit;
    end;
  end;

  // заранее заносим в список отрисованных
  g_intsectbuffer[g_numrenderintsects] := face_index;
```

```
  inc( g_numrenderintsects );
end;

//-----------------------------------------------------------------
// выводит геометрию листа
//-----------------------------------------------------------------
procedure RenderNode( node: pnode_t );
var
  num_intsects:     longword;
  num_intsectports: longword;
  intsect_ports:    array of renderportion_t;

  i, j: integer;
begin
  if not CubeInFrustum( node^.cubeverts ) then
     exit;

  // если у узла есть потомки - рендерим их, иначе рендерим себя
  if node^.is_div then
  begin
    for i := 0 to 7 do if node^.sub_nodes[i] <> nil then
        RenderNode( node^.sub_nodes[i] );
  end;


  // дошли до листа


  inc( g_totalleafdraw );
```

```
// выводим целые грани
if node^.num_ports > 0 then
begin

  for j := 0 to node^.num_ports - 1 do
  begin
    if node^.ports[j].tex_id <> 0 then
    begin
      glEnable( GL_TEXTURE_2D );
      glBindTexture( GL_TEXTURE_2D, node^.ports[j].tex_id );
    end
    else glDisable( GL_TEXTURE_2D );

    glDrawElements( GL_TRIANGLES, node^.ports[j].count,
GL_UNSIGNED_INT, node^.ports[j].vai );
  end;

end;


// выводим дробные грани
if node^.num_intsects > 0 then
begin
  num_intsects := 0;
  for j := 0 to node^.num_intsects - 1 do
  begin
    if FaceAlreadyRender( node^.intsect_indices[j] ) then
       continue;
```

```
      // заносим в массив
      g_intsectindices[num_intsects] := node^.intsect_indices[j];
      inc( num_intsects );
    end;

    if num_intsects = 0 then // все дробные грани этого листа уже выведены ранее
       exit;

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

    // "на ходу" составляем из ещё неотрисованных граней массив индексов вершин
    num_intsectports := GetPortsCount( g_intsectindices, num_intsects );
    setlength( intsect_ports, num_intsectports );
    InitPorts( intsect_ports, num_intsectports, g_intsectindices, num_intsects );

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

    for j := 0 to num_intsectports - 1 do
    begin
      if intsect_ports[j].tex_id <> 0 then
      begin
        glEnable( GL_TEXTURE_2D );
        glBindTexture( GL_TEXTURE_2D, intsect_ports[j].tex_id );
      end
      else glDisable( GL_TEXTURE_2D );

      glDrawElements( GL_TRIANGLES, intsect_ports[j].count, GL_UNSIGNED_INT, intsect_ports[j].vai );
```

```
    end;

    intsect_ports := nil;
  end;
end;

//------------------------------------------------------------------
// строит октарное дерево для сцены
//------------------------------------------------------------------
function Octree_Create(): boolean;
var
  indices: array of longword; // индексы граней
  i: integer;
begin
  result := false;

  // загружаем данные
  if not LoadModel() then
     exit;

  // подготавливаем массив индексов ГРАНЕЙ - здесь, в начале, мы передадим в
  // корень индексы всех граней сцены
  setlength( indices, NUM_FACES );
  for i := 0 to NUM_FACES - 1 do indices[i] := i;


  NewNode( g_root );                          // выделяем память для родительского узла
```

```
  GetRootBBox( g_root, g_verts, NUM_VERTS ); // получаем центр и
размер корня
  GetNewNodeCubeVerts( g_root );             // получаем координаты восьми
вершин куба-корня
  CreateNode( g_root, indices, NUM_FACES );  // создаём узел-корень,
далее процесс идёт рекурсивно
  indices := nil;



  // буфер отрисованных дробных граней
  setlength( g_intsectbuffer, g_intsectbuffersize );
  setlength( g_intsectindices, g_maxintsects );

  // текущие массивы, откуда будут браться вершины для рендеринга
  glVertexPointer( 3, GL_FLOAT, 0, g_verts );
  glTexCoordPointer( 2, GL_FLOAT, 0, g_texcoords );

  result := true;
end;

//-----------------------------------------------------------------
// удаляет октарное дерево
//-----------------------------------------------------------------
procedure Octree_Remove();
begin
  g_verts         := nil;
  g_texcoords     := nil;
  g_faces         := nil;

  g_intsectindices := nil;
```

```
  g_intsectbuffer  := nil;

  if g_root <> nil then
    DeleteNode( g_root );
end;

//-----------------------------------------------------------------
// выводит геометрию октарного дерева по узлам
//-----------------------------------------------------------------
procedure Octree_Render();
begin
  g_numrenderintsects := 0;
  g_totalleafdraw := 0;

  RenderNode( g_root );
end;

end.
unit fMethodOctree;

interface

uses
  Winapi.Windows,
  Winapi.Messages,
  System.SysUtils,
  System.Variants,
  System.Classes,
  System.ImageList,
  Vcl.Graphics,
```

```
  Vcl.Controls,
  Vcl.Forms,
  Vcl.Dialogs,
  Vcl.ImgList,
  Vcl.StdCtrls,
  Vcl.ComCtrls,
  Vcl.Buttons,
  Vcl.ToolWin,
  Vcl.ExtCtrls,
  Vcl.Samples.Spin,
  fMethodDialog;

type
  TfmMethodOctree = class(TfmMethodDialog)
    lbLevels: TLabel;
    seNumberOfLevels: TSpinEdit;
    procedure ButtonOKClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmMethodOctree: TfmMethodOctree;

implementation

{$R *.dfm}
```

```
procedure TfmMethodOctree.ButtonOKClick(Sender: TObject);
begin
  OutModelName := OutModelName + '_octree';
  // Construct Octree;
  // Write Octree in 3D Grid type
end;end.
```