1. Введение

2. Обзор предметной области

3. Реализация алгоритма упрощения

4. Архитектура приложения

В данной главе представлено описание архитектуры и примеры реализации разработанного приложения. Также дано описания стека технологий для разработки и визуализации.

4.1. Используемые технологии.

Так как разрабатываемое приложение должно обрабатывать большие объемы данных за короткое время, было принято решения взять C++ в качестве основного языка разработки. Также с использованием данного языка появляется возможность написания графического интерфейса на довольно низком уровне за короткое время с использованием готовых графических библиотек, а также GUI библиотек, что значительно ускоряет работу программы при визуализации значительного количества данных, в нашем случае полигонов.

В качестве основной графической библиотеки был выбран **OpenGL**, так как данная библиотека является наилучшим решением при разработке графики на платформе Linux при использовании языка разработки C++. Также для ускорения процесса разработки была выбрана библиотека **GLEW**, упрощающая запросы и загрузку расширений OpenGL.

Для создания интерфейса программы была использована библиотека **ImGui**. Данная библиотеки представляет удобные инструменты при создании окон и виджетов для удобного взаимодействия с приложением.

При разработке было принято решения отойти от использования кроссплатформенного IDE Qt из-за большого объема дополнительных библиотек, необходимых для создания приложения, также использование ImGui дает соразмерный результат при создании окон, использует меньшее количество сторонних библиотек, а также повышает скорость написания кода.

4.2. Основной программный цикл.

Для реализации основного программного цикла был разработан класс singleton класс Application с определенным методом run, внутри которого обрабатываются команды пользователя при взаимодействии с интерфейсом программы.

Данный классный является singleton классом, то есть объект данного класса может существовать в единственном экземпляре, который хранится как статическое поле класса и инициализируется после первой попытки создания объекта. Данное решение обусловлено тем, что для разрабатываемого приложения достаточно наличия единственного контекста работы, в котором обрабатываются события по обработке данных и визуализации полученных результатов.

Основные методы класса Application:

- Application() – Используется для инициализации хранимых данных, а также для создания контекста работы gui приложения и инициализации callback методов по обработки действий пользователя.
- run() – Запуск основного цикла работы программы, а также инициализация компонент для визуализации обрабатываемых данных, таких как шейдеры. В основном цикле происходит обработка событий от пользователя, таких как нажатие клавиш устройств ввода или изменения размеров рабочего окна.
- getInstance() – Используется для получения экземпляра объекта класса. Данный метод является публичный и используется для получения доступа к объекту от callback функций или gui составляющей приложения.
- resetInstance() – Освобождает экземпляр класса.

Остальные методы являются контекстно зависимыми и будут рассмотрены в следующих разделах.

После запуска приложения происходит инициализация и вызов соответствующего метода run, который запускает рабочий цикл программы.

4.3. Хранение данных.

В качестве обрабатываемых данных выступает полигональная сетка, для которой был разработан шаблонный класс ***template <typename TVertexComponents> class Mesh***, который хранит информацию о вершинах и гранях модели. Так как для данного алгоритма упрощения достаточно исключительно информации о вершинах и гранях, данные о ребрах или полу-ребрах(half-edge) было решено не хранить для экономии ram. Для использования функционала данного класса и его наследников необходимо, чтобы тип TVertexComponents был наследником класса VertexComponentsColored, в котором уже заложены основные компоненты вершин, которые будут описаны ниже.

Основные поля:

- m_vertices – список вершин сетки.
- m_faces – список граней сетки.

Данные поля являются приватным и предоставляются при помощи соответствующих методов vertices() и faces(), которые предоставляют доступ только на чтение для предотвращения их изменения вне основного функционала.

4.3.1. Данные о вершинах.

Для хранения информации о вершине используется шаблонный класс **template <typename TVertexComponents> class Vertex**, который содержит единственное публичное поле TVertexComponents components. Данный

шаблон является структурой наследником абстрактного класса VertexComponents который хранит информацию о компонентах вершины, таких как позиция, цвет, нормаль и т.д. Благодаря наследованию появляется возможность расширять имеющийся набор данных, хранимый вершиной, тем самым ограничивая расходы памяти на хранения ненужных данных. Также появляется возможность определения поведения компоненты вершины при изменении её позиции при помощи виртуального метода virtual void interpolate(…), который будет описан ниже.

Методы обработки вершин:

- interpolate(…) - Является линейной интерполяцией компонент вершины, однако может переопределяется разработчиком для каждого наследника при необходимости, например, для отключения интерполяции для uv координат. Данный метод используется для пересчета компонент вершин при схлопывании.

Основными компонентами вершины, которые используются приложением являются:

- Позиция.
- Нормаль. Необходима для определения «направления» грани и используется для вычисления объема модели и освещения на сцене.
- Цвет. Используется как замена текстурам.
- Координаты текстур. Используется для определения текселя(пикселя текстуры).

4.3.2. Данные о гранях

Для хранения информации о гранях используются классы Face или UVFace, в каждом из которых присутствуют три целочисленных поля для хранения индексов соответствующих вершин в вершинном списке.

Класс UVFace является наследником класса Face и хранит дополнительную информацию о текстурных координатах соответствующих

вершин. Данная информация используется для инициализации вершинного буфера во время рендера, так как каждая вершина, являющаяся частью нескольких полигонов, может содержать разные текстурные координаты.

Таким образом список вершин и список граней с индексами соответствующих вершин позволяют хранить любую полигональную сетку.

4.4. Обработка данных.

В данном подразделе описаны методы по упрощению полигональной сетки, а также методы для оценки точности упрощения, такие как площадь, объем, а также значение квадратичной ошибки упрощенной модели.

4.4.1. Упрощение модели.

Для упрощения 3д моделей был написан блок функций в пространстве имен Simplify. Основой является шаблонная фукция simplify_mesh, которая принимает указатель на объект класса Mesh и финальное количество граней модели. Полученных списков вершин и ребер из меша достаточно для запуска алгоритма, однако для работы алгоритма необходимо наличие дополнительных параметров для каждой вершины или ребра, таких как ошибка, список соседних граней и т.д. Для этого были добавлены отдельные классы для хранения вершины и ребра, которые представлены ниже.

- struct Vertex { vec3f p; int tstart, tcount; SymetricMatrix q; int border; };
- struct Triangle { int v[3]; double err[4]; int deleted, dirty; vec3f n; };

В начале алгоритма происходит копирование данных из меша в соответствующие списки вершин и граней с последующей инициализацией дополнительных параметров.

Также для хранения связей между вершинами и гранями добавляется список структур типа: struct Ref { int tid,tvertex; };

После инициализации всех данных алгоритм начинает работу, итеративно проходя по вершинам и удаляя те грани, значение ошибки которых

меньше определенного значения(threshold). Как только удаляется очередное ребро, появляется необходимость определения позиции новой точки, а также изменение или установка компонент новой вершины. Позиция вершины находится исходя из значения с квадратичной ошибки в функции **calculate_error**, описание которой дано ниже. Для расчета значений дополнительных компонент вершины достаточно получить значение весов(weights) для формулы линейной интерполяции, которое получается из отношения

$$t = \frac{|v - v0|}{|v1 - v0|}$$

Далее для новой вершины достаточно вызвать метод v.**interpolate**(v0, v1, t), получив тем самым корректные значения компонент вершины.

После достижения указанного количества полигонов алгоритм прекращает работу и компонует списки вершин и граней. Стоит отметить, что никакого копирования значений из временного списка, необходимого для работы алгоритма, в исходный не происходит, так как все значения позиции вершин были получены из интерполяции, а индексы граней перезаписывались во время работы алгоритма одновременно с временными данными.

Для запуска алгоритма были реализованы две виртуальные функции, описание которых дано ниже:

- void simplify(float p = 0.5f) – Упрощает модель, оставляя долю граней, указанную в параметре p.
- void simplify(uint finalFaceCount) – Упрощает модель, оставляя указанное количество граней.

4.4.2. Расчет нормалей.

Нормаль можно определить по формуле:

$N = \left|\left|crossProduct(v1 - v0, v2 - v0)\right|\right|$, где v0, v1,v2 – координаты конкретной вершины.

Для придания модели более сглаженного вида было принято решение хранить для каждой вершины только одну нормаль, которая является средним между всеми нормалями соединяющих её граней, что также позволяет избавиться от копий существующих вершин. Таким образом нормаль вершины определяется по следующей формуле:

$N = \left|\sum_i^k Ni\right|$, где Ni – нормаль связанной грани, которая вычисляется по формуле выше.

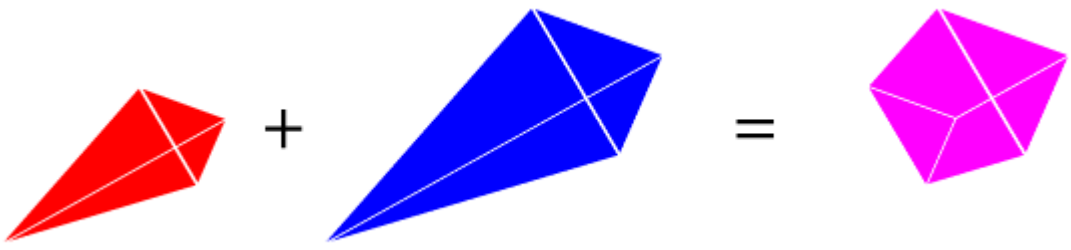
Данная формула реализуется в методе **calculate_normals** и используется после каждого загрузки или упрощения модели.

4.4.3. Площадь и объём модели.

Площадь модели оценивается исходя из суммы площадей всех её граней. Площадь отдельного же треугольника можно получить из формулы:

$S = |crossProduct(v1 - v0, v2 - v0)|$, где v0, v1, v2 – вершины треугольника.

Объем модели можно оценить исходя из суммы знаковым объемов тетраэдров, образованных точками v0, v1, v2, O, где v0, v1, v2 – координаты вершин конкретного треугольника, а O – начало(origin) координат. Стоит отметить, что нас интересует именного знаковый объем, так как в нашем случае у каждого треугольника имеется конкретное направление.



Значение знакового объема можно получить из формулы:

$$\frac{dotProduct\big(v0, crossProduct(v1, v2)\big)}{6}$$

Данные формулы реализованы в качестве публичных методов класса **Mesh**.

5. Визуализация.

В данном подразделе описаны методы для визуализации полигональной сетки. Дается общее представление об используемых технологиях, а также информация о классах и методах, реализующих хранение, доступ и взаимодействие с данными для рендера.

5.1. Хранение данных.

После разработки структур для хранения данных о полигональной сетке необходимо позаботиться о хранении данных, которые будут использоваться для ее визуализации, так как простого списка для вершин и граней не хватит для корректного представления данных. Для этих целей был разработан шаблонный класс template **<typename TVertexComponents> RenderMesh**, который является публичным наследником класса **Mesh**.

Создание данного класса понадобилось для разделения данных меша, которые можно использовать без графической составляющей, и данных, которые необходимы **OpenGL** для рисования полигонов. В качестве таких данных выступают вершинные и индексные буферы, материалы и текстуры.

Так как визуальная составляющая меша не сильно важна в рамках данной статьи, в качестве материалов выступают исключительно текстуры или же цвет, которых достаточно для передачи пользователю образа модели.

5.1.1. Материалы и текстуры.

Инициализация данных для рендера осуществляется в приватном методе initMaterials, который вызывается после каждой загрузки нового меша. В данном методе происходит привязка буферов OpenGL к текущим данным о вершинах и гранях. Стоит отметить, что списки вершин укомплектованы

достаточно хорошо, чтобы привязка происходила без необходимости копирования информации. Это осуществляется при помощи параметров OpenGL, которые позволяют указывать смещение и отступ от буфера с данными для каждой компоненты, тем самым позволяя использовать один и тот же список вершин для разных буферов. Однако из-за наличия у вершины разных текстурных координат перед рендером необходимо создавать копии уже имеющихся вершин, что немного увеличивает общий объем, о чем будет написано в разделе 5.1.3.

Также в методе initMaterials происходит загрузка в память используемых моделью текстур, пути к которым уже хранятся в специальной структуре внутри классов для загрузки модели из файла, о которых будет написано в разделе **IO**.

5.1.2. Рендер-группы.

После загрузки текстур происходит инициализации индексного буфера, который используется для указания порядка рендера вершин в созданном заранее вершинном буфере, а также создание рендер-групп.

Render-groups – разработанная структура группировки вершин по типу используемого материала. Данная структура необходима для рендера вершин, которые разделяют одну и ту же текстуру, что дополнительно позволяет уменьшить количество drawCall-ов, а как следствие, увеличить производительность.

5.1.3. Создание копий вершин.

Несколько связанных граней могут использовать различные участки текстуры, или же совершенно разные. По этой причине у конкретной вершины, разделяющей подобные грани, могу быть разные текстурные координаты, которые невозможно хранить в самой вершине из-за нарушения «однородности данных». Чтобы обойти данную проблему было реализовано создание копий существующих вершин для каждой координаты текстуры, а

также привязка граней с созданным копиям. Данное решение незначительно увеличивает общий объем хранимых данных, порядка нескольких десятков килобайт, что не очень существенно по сравнению с хранением текстур, которые в лучшем случае занимают несколько мегабайт.

Однако такое решение вызывает проблемы при упрощении расширенной модели, так как копии вершин не являются реальной частью модели и не могут обрабатываться алгоритмом. Появляется необходимость в хранении для граней, указывающих на копии вершин, данных об их реальных вершинах и восстановлении этих данных перед каждым упрощением. Для этого была реализована структура **FaceNativeData**, хранящая информацию о гране и ее родных вершинах, а также соответствующий список.

5.2. Обработка данных

Так как класс RenderMesh добавляет дополнительные данные в текущий список вершин необходимо переопределить методы по обработке данных и выводе информации о модели. Для этого вводится новая приватная переменная класса m_vertices_count, хранящая реальное количество вершин.

Также переопределяются методы для упрощения исходной модели simplify(…), в которых перед упрощением происходит восстановление оригинальных данных модели, а также повторная инициализации буферов OpenGL, так как предыдущие данные были искажены удалением вершин.

5.3. Шейдеры

6. Тестирование приложения

7. Безопасность жизнедеятельности