# Introduction

## Overview ­­­­

The course aims to teach a student the basics of CAD application creation from a scratch and working with the main design patterns. During the course, a student will learn how to work with technologies such as *OpenGL*, *glfw*, *COLLADA* and will get acquainted with some basic terms like *scene tree*, *manipulators*, *half-edge data structure*, *MVO pattern*. As an application, a student will write a simple mesh editor tool.

Цель курса - научить студента основам создания САПР с нуля и работать с основными шаблонами дизайна. Во время курса ученик узнает, как работать с такими технологиями, как OpenGL, glfw, COLLADA, и познакомится с некоторыми базовыми терминами, такими как дерево сцен, манипуляторы, структура данных с половинным краем, шаблон MVO. В качестве приложения ученик напишет простой инструмент редактора сетки.

## Objective

During the course, a student will implement Mesh Editor that allows manipulating with the triangle meshes. Resulting application will allow loading and editing COLLADA mesh files that are now used by many major modeling packages and real-time graphics engines.

Во время курса студент будет реализовывать редактор Mesh, который позволяет манипулировать с помощью треугольных сеток. Результирующее приложение позволит загружать и редактировать файлы сетки COLLADA, которые теперь используются многими основными пакетами моделирования и графическими движками в реальном времени.

## Structure of the Course

The course contains 5 lab works. Each lab work contains theoretical info or the external link to additional materials that are recommended to read. Also, there is a practical part in every lab work and some extra exercises to polish the skills.

Each lab work contains the header file, which you will need to implement. Your header file should be exactly the same as in Lab work. Note that all code listings do not contain header guards (#pragma once or ifdef X\_H/endif), and you need to manually place them. Your task is to complete the code in those places where there is a comment **TODO** and write corresponding *.cpp* file. Sometimes part of the *.cpp* file will be shown as a demonstration or an explanation of the task. The resulting code does not have to be exactly the same in your implementation.

Курс содержит 5 лабораторных работ. Каждая лабораторная работа содержит теоретическую информацию или внешнюю ссылку на дополнительные материалы, которые рекомендуется читать. Кроме того, есть практическая часть в каждой лабораторной работе и некоторые дополнительные упражнения, чтобы отполировать навыки.

Каждая лабораторная работа содержит файл заголовка, который вам нужно реализовать. Ваш заголовочный файл должен быть точно таким же, как в работе лаборатории. Обратите внимание, что все списки кодов не содержат защиты заголовков (#pragma once или ifdef X\_H / endif), и вам нужно вручную их разместить. Ваша задача - завершить код в тех местах, где есть комментарий TODO и записать соответствующий файл .cpp. Иногда часть файла .cpp будет показана как демонстрация или объяснение задачи. Полученный код не должен быть точно таким же в вашей реализации.

### Lab Work 1

Creating a window, rendering geometry and processing of keyboard input/mouse input.

### Lab Work 2

Creating **Camera** and **Viewport** classes.

### Lab Work 3

*Half-edge data structure* implementation. Based on class **HalfEdgeTable** the class **Mesh** will be implemented.

Реализация структуры данных с половинной гранью. На основе класса HalfEdgeTable будет реализован класс Mesh.

### Lab Work 4

Implementation of the Mesh Editor basis using the MVO pattern. Class **Operator** is introduced. Implementation of **DeleteFaceOperator**. On this stage Mesh Editor allows to load/save \*.dae model, delete face and display a mesh using various settings of camera and viewport.

Реализация основы Mesh Editor с использованием шаблона MVO. Оператор класса введен. Реализация DeleteFaceOperator. На этом этапе Mesh Editor позволяет загрузить / сохранить модель \* .dae, удалить лицо и отобразить сетку, используя различные настройки камеры и видового экрана.

### Lab Work 5

Implementation of the Translation and Rotation manipulators. Based on these manipulators a **Triad** manipulator will be assembled. **EditMeshOperator** will be implemented using **TranslationManipulator**. Then a **Triad** will be used for **TransformMeshOperator** implementation. On this final stage Mesh Editor allows to edit faces, select a mesh and change its transformation.

## Requirements

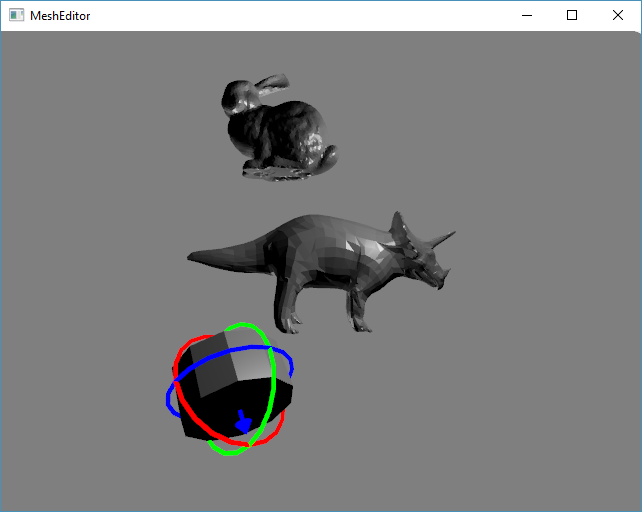
Algebra, С++, OOP, algorithms, computer graphics - 1, 2 courses.

## Mesh Editor

### Functional Requirements

Mesh Editor is an application that takes COLLADA (\*.dae) file at the input and provides operations for changing a mesh - moving a triangle in the given direction or deleting a triangle. A modified mesh appears at the output of an application and this mesh can be saved in COLLADA (\*.dae).

Mesh Editor - это приложение, которое принимает файл COLLADA (\* .dae) на входе и предоставляет операции для изменения сетки - перемещение треугольника в заданном направлении или удаление треугольника. Модифицированная сетка появляется на выходе приложения, и эта сетка может быть сохранена в COLLADA (\* .dae).



### Commands

All commands are divided into three types: a *simple* keyboard command, a *complex* keyboard command, and a mouse command.

A simple keyboard command is a command that is executed when the user presses and releases the button immediately. For example, the user can change the perspective projection to parallel by pressing **F8**.

A complex keyboard command starts when the user presses a key and ends the command by pressing **ESC**. For example, to start a mode when a user can delete faces, press the **D** key, and then delete the needed faces with the mouse. To exit this mode, press the **ESC** button.

A table of all the keyboard controls in the mesh editor application is provided below:

|  |  |  |
| --- | --- | --- |
| Command | Activated by pressing the key: | Deactivated by pressing the key: |
| A user selects a face and removes it | D | ESC |
| A user selects a triangle and moves it in the given direction. | E | ESC |
| User selects a node and changes its transformation with the help of the triad | T | ESC |
| A user presses the V key and a separate window displaying a model appears | V |  |
| A user presses the S key and the result is saved in DAE file (initial name + modified) | S |  |
| Front View | F1 |  |
| Rear View | F2 |  |
| Right View | F3 |  |
| Left View | F4 |  |
| Top View | F5 |  |
| Bottom View | F6 |  |
| Isometric View | F7 |  |
| Parallel Projection | F8 |  |
| Zoom to Fit | F9 |  |

The mouse command starts when the user clicks the mouse button and ends when the user releases it. For example, to perform a pan operation, press the left mouse button, make the necessary manipulations with the camera and release it.

A table of all the mouse controls in the Mesh Editor application is provided below:

|  |  |  |
| --- | --- | --- |
| Command | Activated by pressing the mouse button: | Deactivated by pressing the mouse button: |
| Pan | Press LMB | Release LMB |
| Zoom | Press MMB | Release MMB |
| Trackball | Press RMB | Release RMB |

*LMB – left mouse button, MMB – middle mouse button, RMB – right mouse button.*

### Example Workflow

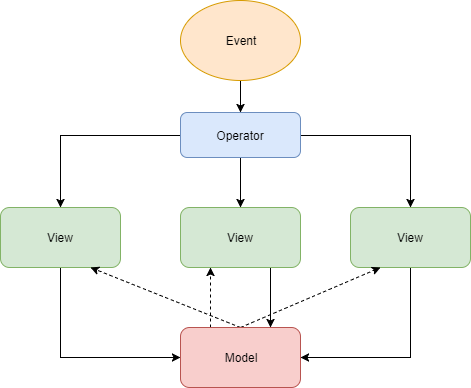
1. User runs Mesh Editor with command argument ‘MeshEditor rhino.dae’
2. User presses the **F9** key in order to do *Zoom to Fit*
3. User adjusts a camera in the required way, using mouse buttons and the **F1**-**F9** keys
4. User presses **E** and enters to **EditMeshOperator**
5. User selects the triangle on the model and moves it in the given direction
6. Item 5 should be repeated until the desired result is achieved
7. User presses **ESC** in order to finish **EditMeshOperator**
8. User presses the **S** key and the file is saved to ‘rhino-modified.dae’

### Environment

MSVC 2015, OpenGL 2.1, glfw, glm, tinyxml

## Theory

Mesh Editor will be implemented using **MVO** pattern (M – model, V – view, O – operator). This is a basic pattern used in one form or another in many CAD engines. It is assumed that you have a **Model** in the scene and the **View**, which displays this **Model**. In turn, a **Model** or the **View** respond to some external *irritants*, that can change the model or the **View**. Such *irritants* are called **Operators**. Mouse buttons and a keyboard are the source of *irritation*. Overriding an **Operator** class, a user creates his own *reactions* for the **Model**.



A **Model** is a list of triangles, the **View** is an OpenGL window, which can draw this list of triangles, and **Operator** is an abstract class, which contains three methods: *onKeyPressed*, *onMousePressed* and *onMouseMove*.

During the course it is necessary to implement two operators for the **Model** - editing and transforming a mesh and operators for changing the camera view.

In the code application can be expressed as follows:

|  |
| --- |
| #include "Application.h"  #include "DeleteFaceOperator.h"  #include "COLLADAParser.h"  int main()  {  Application app("modified-myModel.dae");  std::unique\_ptr<Model> model = loadModel("myModel.dae");  View\* view = app.createView("Hello World", 640, 480);  view->setModel(model.get());  view->addOperator(KeyCode::D, KeyCode::ESCAPE, std::make\_unique<DeleteFaceOperator>());  app.run();  return 0;  } |

Further, we’ll discuss the fundamental classes, that would be implemented in the next lab works that will provide an overall picture of Mesh Editor application architecture. At this stage, there is no need to focus on each class and deal with it in detail. It is enough to understand the general idea of MVO pattern. *Note that MVO is the fully equivalent to* [MVC](#MVC) *(M – model, V – view, C – controller) pattern.* Consider reading technical part of [HOOPS](#HOOPS) MVO architecture that describes MVO pattern usage in HOOPS engine.

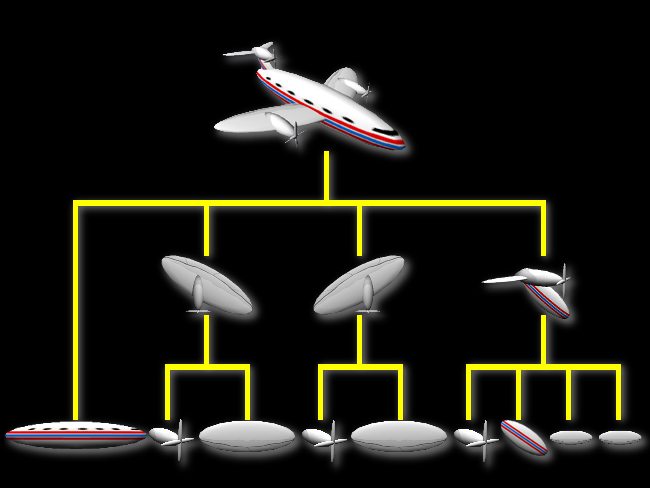
## MVO classes

In Mesh Editor, **Model** is something more complex than just a list of triangles, it is a tree node, which is built according to the loaded \*.dae file. Each **Node** stores a **Mesh**, a relative matrix transformation and the list of pointers to child **Nodes**.

The **View** displays a **Model**. For one **Model** can be more than one **View**. In our particular case the **View** is an OpenGL window, which can render (but can’t change) a **Model** (a node tree).

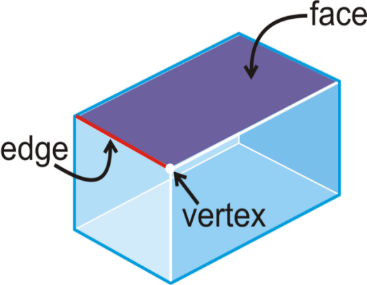
**Operator** is application logic. All the operations on the **Model** and **Camera** are performed in the **Operator**. At the input the **Operator** receive the events of keyboard and mouse pressing. Depending on the required key, a certain action is performed (for example, changing the camera position or mesh modification).

A **Node** is a [tree node](#SceneGraph) which contains a **Mesh** and its transformation into the scene, as well as the list of child elements and the pointer to the parent node. The tree node data structure is very useful for presenting hierarchical data structures like 3D Scene. It allows you to describe the relations between different 3D objects. See the image below:



Also for **Node** it will be useful to have a function, which can calculate the absolute matrix transformation.

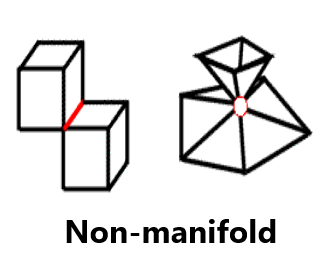
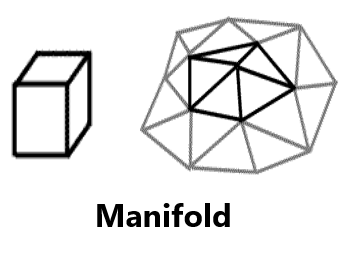
*Mesh data structure* is a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling. The faces usually consist of triangles (only triangle meshes are used in labs), since this simplifies rendering, but may also be composed of more general concave polygons, or polygons with holes.



*Half-edge data structure* is necessary to build a connection between the faces of the mesh. For example, it is important to have topology information for the operation of moving the face. If a user moves the face (triangle), then it is necessary that the neighboring faces also catch up and also moves with no holes appearing. Any valid half-edge mesh must be *manifold* and *orientable*. In this course, therefore only manifold and oriented meshes are provided as an input.

A mesh is a *manifold* if:

1. Every edge is adjacent to one (boundary) or two faces
2. For every vertex, its adjacent polygons form a disk (internal vertex) or a half-disk (boundary vertex)



**HalfEdgeTable** class that implements *half-edge data structure* will be explained in Lab 3.

**Mesh** is the class that encapsulates **HalfEdgeTable** data structure and its operations. It provides *render* function which converts **HalfEdgeTable** data into the data (*triangle soup data structure*) that **IRenderSystem** understands. It is obvious that doing such a conversion in each frame is quite an expensive operation, so most likely this conversion will be done in two cases: first case happens when **HalfEdgeTable** is passed to the constructor, and the second case - in all operations, that require changes in the Mesh topology (like *deleteFace*, for example).

**View** is an OpenGL window that can render a **Model** using **Render System** and the specified **Camera** settings. In addition, **View** provides a *raycast* function that finds intersection between the cursor ray and the model and returns the list of contacts. **Contact** is a structure that contains a pointer to the intersected face and its corresponding **Node**. The *update* function does rendering of the **Model**. The *raycast* function will be explained in Lab 4. The **IRenderSystem** will be explained in Lab 1. **Camera** and **Viewport** will be explained in the Lab 2.

**Application** is the main class that manages the objects of application. Using it, a user can create a **View**. The lifetime of **Application** is limited by the scope of the *main* function. The lifetime of **View** is limited by the lifetime of **Application**. **View** should not outlive the **Model**.

Assembling everything together the application that would allow user to delete a face in the mesh would look like this:

|  |
| --- |
| #include "Application.h"  #include "DeleteFaceOperator.h"  #include "COLLADAParser.h"  int main()  {  Application app("modified-myModel.dae");  std::unique\_ptr<Model> model = loadModel("myModel.dae");  View\* view = app.createView("Hello World", 640, 480);  view->setModel(model.get());  view->addOperator(KeyCode::D, KeyCode::ESCAPE, std::make\_unique<DeleteFaceOperator>());  app.run();  return 0;  } |

This chapter did not show the other classes (i.e. **IRenderSystem**or**Camera**) that are used to implement the Mesh Editor. They are also important, but they are the parts of the implementation of the interiors of the MVC and therefore they are omitted for the sake of simplicity. These classes will be explained later in the lab works.

In the end the following solution structure should be created:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **.** | **MeshEditor** | **GLRenderSystem** | **HalfEdge** | **Interfaces** | **ThirdParty** |
| MeshEditor  GLRenderSystem  HalfEdge  Interfaces  ThirdParty  MeshEditor.sln | TransformNode.h  TransformNode.cpp  EditMesh.h  EditMesh.cpp  Manipulator.h  Manipulator.cpp  Triad.h  Triad.cpp  RotationManipulator.h  RotationManipulator.cpp  TranslationManipulator.h  TranslationManipulator.cpp  Application.h  Application.cpp  View.h  View.cpp  FilterValue.h  Contact.h  ColladaParser.h  ColladaParser.cpp  TrackBall.h  TrackBall.cpp  Pan.h  Pan.cpp  Node.h  Node.cpp  Model.h  Model.cpp  Camera.h  Camera.cpp  Viewport.h  Viewport.cpp  Mesh.h  Mesh.cpp  DynamicLibrary.h  DynamicLibrary.cpp  main.cpp  MeshEditor.vcxproj | GLRenderSystem.h  GLRenderSystem.cpp  GLWindow.h  GLWindow.cpp  Exports.h  Exports.cpp  glad.h  glad.c  khrplatform.h  GLRenderSystem.vcxproj | HalfEdge.h  HalfEdge.cpp  HalfEdge.vcxproj | IWindow.h  IRenderSystem.h | glm  glfw  tinyxml2 |

## Resources and Notes

1. <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> - MVC pattern

1. <http://docs.techsoft3d.com/visualize/3df/21/2113/HoopsMVO/tech_overview/TechnicalOverview_C.html> - HOOPS/MVO Technical Overview

1. <https://en.wikipedia.org/wiki/Scene_graph> - explanation of s*cene graph* term. In the lab works *Tree node* will be implemented instead for a simplicity

Введение

Обзор ¬¬¬¬

Цель курса - научить студента основам создания САПР с нуля и работать с основными шаблонами дизайна. Во время курса ученик узнает, как работать с такими технологиями, как OpenGL, glfw, COLLADA, и познакомится с некоторыми базовыми терминами, такими как дерево сцен, манипуляторы, структура данных с половинным краем, шаблон MVO. В качестве приложения ученик напишет простой инструмент редактора сетки.

Задача

Во время курса студент будет реализовывать редактор Mesh, который позволяет манипулировать с помощью треугольных сеток. Результирующее приложение позволит загружать и редактировать файлы сетки COLLADA, которые теперь используются многими основными пакетами моделирования и графическими движками в реальном времени.

Структура курса

Курс содержит 5 лабораторных работ. Каждая лабораторная работа содержит теоретическую информацию или внешнюю ссылку на дополнительные материалы, которые рекомендуется читать. Кроме того, есть практическая часть в каждой лабораторной работе и некоторые дополнительные упражнения, чтобы отполировать навыки.

Каждая лабораторная работа содержит файл заголовка, который вам нужно реализовать. Ваш заголовочный файл должен быть точно таким же, как в работе лаборатории. Обратите внимание, что все списки кодов не содержат защиты заголовков (#pragma once или ifdef X\_H / endif), и вам нужно вручную их разместить. Ваша задача - завершить код в тех местах, где есть комментарий TODO и записать соответствующий файл .cpp. Иногда часть файла .cpp будет показана как демонстрация или объяснение задачи. Полученный код не должен быть точно таким же в вашей реализации.

Лабораторная работа 1

Создание окна, визуализация геометрии и обработка ввода / ввода клавиатуры.

Лабораторная работа 2

Создание классов Camera и Viewport.

Лабораторная работа 3

Реализация структуры данных с половинной границей. На основе класса HalfEdgeTable будет реализован класс Mesh.

Лабораторная работа 4

Внедрение основы редактора Mesh с использованием шаблона MVO. Вводится оператор класса. Реализация DeleteFaceOperator. На этом этапе Mesh Editor позволяет загружать / сохранять \* .dae модель, удалять лицо и отображать сетку, используя различные настройки камеры и видового экрана.

Лабораторная работа 5

Внедрение манипуляторов перевода и поворота. На основе этих манипуляторов будет собран манипулятор Triad. EditMeshOperator будет реализован с помощью TranslationManipulator. Затем для реализации TransformMeshOperator будет использоваться Triad. На этом заключительном этапе Mesh Editor позволяет редактировать лица, выбирать сетку и изменять ее преобразование.

Функциональные требования

Mesh Editor - это приложение, которое принимает файл COLLADA (\* .dae) на входе и предоставляет операции для изменения сетки - перемещение треугольника в заданном направлении или удаление треугольника. Модифицированная сетка появляется на выходе приложения, и эта сетка может быть сохранена в COLLADA (\* .dae).

Команды

Все команды делятся на три типа: простая команда клавиатуры, сложная команда клавиатуры и команда мыши.

Простая команда клавиатуры - это команда, которая выполняется, когда пользователь нажимает и отпускает кнопку немедленно. Например, пользователь может изменить проекцию перспективы параллельно, нажав F8.

Сложная команда клавиатуры начинается, когда пользователь нажимает клавишу и завершает команду, нажимая ESC. Например, чтобы запустить режим, когда пользователь может удалить лица, нажмите клавишу D, а затем удалите нужные лица с помощью мыши. Чтобы выйти из этого режима, нажмите кнопку ESC.

Теория

Редактор Mesh будет реализован с использованием шаблона MVO (M - модель, V - view, O - оператор). Это базовая модель, используемая в той или иной форме во многих CAD-системах. Предполагается, что у вас есть модель в сцене и вид, который отображает эту модель. В свою очередь, модель или вид реагируют на некоторые внешние раздражители, которые могут изменить модель или вид. Такие раздражители называются операторами. Кнопки мыши и клавиатура являются источником раздражения. Переопределяя класс Operator, пользователь создает свои собственные реакции для Модели.

Модель представляет собой список треугольников, представление - это окно OpenGL, которое может нарисовать этот список треугольников, а Operator - абстрактный класс, который содержит три метода: onKeyPressed, onMousePressed и onMouseMove.

Во время курса необходимо реализовать два оператора для модели - редактирование и преобразование сетки и операторов для изменения вида камеры.

В приложении кода можно выразить следующим образом:

Далее мы обсудим фундаментальные классы, которые будут реализованы в следующих лабораторных работах, которые предоставят общую картину архитектуры приложения Mesh Editor. На этом этапе нет необходимости сосредотачиваться на каждом классе и подробно разбираться с ним. Достаточно понять общую идею шаблона MVO. Обратите внимание, что MVO полностью эквивалентен шаблону MVC (модель M, модель V, view, C-controller). Рассмотрим возможность чтения технической части архитектуры HOOPS MVO, описывающей использование шаблонов MVO в движке HOOPS.

Классы MVO

В Mesh Editor модель является чем-то более сложным, чем просто список треугольников, это узел дерева, который построен в соответствии с загруженным файлом \* .dae. Каждый узел хранит Mesh, относительное преобразование матрицы и список указателей на дочерние узлы.

В представлении отображается модель. Для одной модели может быть несколько видов. В нашем конкретном случае View представляет собой окно OpenGL, которое может отображать (но не изменять) модель (дерево узлов).

Оператор - прикладная логика. Все операции над моделью и камерой выполняются в Операторе. На входе оператор получает события нажатия клавиш и мыши. В зависимости от требуемого ключа выполняется определенное действие (например, изменение положения камеры или модификации сетки).

Узел - это узел дерева, который содержит Mesh и его преобразование в сцену, а также список дочерних элементов и указатель на родительский узел. Структура данных узла дерева очень полезна для представления иерархических структур данных, таких как 3D-сцена. Это позволяет описать отношения между различными 3D-объектами. См. Изображение ниже:

Также для узла будет полезно иметь функцию, которая может вычислять абсолютное матричное преобразование.

Структура данных сетки представляет собой набор вершин, ребер и граней, который определяет форму многогранного объекта в 3D компьютерной графике и твердотельном моделировании. Лица обычно состоят из треугольников (в лабораториях используются только треугольные сетки), поскольку это упрощает рендеринг, но может также состоять из более общих вогнутых многоугольников или многоугольников с отверстиями.

Для построения связи между гранями сетки необходима структура данных с половинным фронтом. Например, важно иметь топологическую информацию для операции перемещения лица. Если пользователь перемещает лицо (треугольник), то необходимо, чтобы соседние грани также догоняли, а также перемещались без появления отверстий. Любая допустимая сетка с половинной кромкой должна быть многообразной и ориентируемой. Поэтому в этом курсе в качестве входных данных предусмотрены только многомерные и ориентированные сетки.

Сетка представляет собой многообразие, если:

1. Каждое ребро смежно к одной (границе) или двум граням

2. Для каждой вершины ее смежные многоугольники образуют диск (внутренняя вершина) или полупик (граничная вершина)

Класс HalfEdgeTable, который реализует структуру данных с половинным фронтом, будет объяснен в Лаборатории 3.

Mesh - это класс, который инкапсулирует структуру данных HalfEdgeTable и ее операции. Он предоставляет функцию рендеринга, которая преобразует данные HalfEdgeTable в данные (структуру данных суппорта треугольника), которые понимает IRenderSystem. Очевидно, что такое преобразование в каждом кадре является довольно дорогостоящей операцией, поэтому, скорее всего, это преобразование будет выполнено в двух случаях: первый случай случается, когда HalfEdgeTable передается конструктору, а второй случай - во всех операциях, что требуют изменений в топологии Mesh (например, deleteFace, например).

View - это окно OpenGL, которое может отображать модель с использованием системы Render и заданных настроек камеры. Кроме того, View предоставляет функцию raycast, которая находит пересечение между лучом курсора и моделью и возвращает список контактов. Contact - это структура, содержащая указатель на пересекаемую грань и соответствующий ей узел. Функция обновления выполняет рендеринг Модели. Функция raycast будет объяснена в Лаборатории 4. IRenderSystem будет объяснена в Lab 1. Camera and Viewport будет объяснена в Лаборатории 2.

Приложение является основным классом, который управляет объектами приложения. Используя его, пользователь может создать представление. Срок действия приложения ограничен объемом основной функции. Время жизни View ограничено временем жизни приложения. Просмотр не должен переживать Модель.

Сборка всего вместе приложения, которое позволит пользователю удалить лицо в сетке, будет выглядеть так:

В этой главе не показаны другие классы (например, IRenderSystem или Camera), которые используются для реализации редактора Mesh. Они также важны, но они являются частью реализации интерьеров MVC, и поэтому они опущены для простоты. Эти классы будут объяснены позже в лабораторных работах.

В итоге должна быть создана следующая структура решения: