# Lab 4

## Overview

## This lab work is dedicated to the implementation of the Mesh Editor based on the MVO pattern. After implementing this Lab work, Mesh Editor will allow to load, save, delete face and display mesh using various settings of camera and viewport.

## Objective

1. To implement *saveModel/loadModel* functions that imports/exports .DAE to/from **Model** class
2. To implement **View** that uses **Viewport** and **Camera** from the Lab work 2:
   1. Implement **Application** class that manages **View** and **IRenderSystem** implemented in the Lab work 3
   2. Implement **Operator** class and add operators for the functions in the **Camera** and **Viewport** classes. The list of functions is listed below:

|  |  |
| --- | --- |
| Command | Key |
| A user presses the V key and a separate window displaying a model appears | V |
| Pressing the S key a user saves the result 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 |

1. Implement **DeleteFaceOperator**. *This problem is put in a separate item because it describes the problem of Collision Detection*

## Infrastructure

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **.** | **MeshEditor** | **GLRenderSystem** | **HalfEdge** | **Interfaces** | **ThirdParty** |
| MeshEditor  GLRenderSystem  HalfEdge  Interfaces  ThirdParty  MeshEditor.sln | **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 | **tinyxml2**  glm  glfw |

## Dependencies

## Add **tinyxml2** library which is needed for parsing COLLADA \*.dae files.

## Task 1

Start lab work by writing *loadModel* and *saveModel* methods. Note, that *loadModel* returns **unique\_ptr**. Thus, it is not necessary to manage lifetime of the result **Model** explicitly. In case of an error loading return **nullptr**.

Note that each COLLADA file may contain multiple mesh objects; more generally, a COLLADA file describes a scene graph that is a hierarchical representation of all objects in the scene (meshes, cameras, lights, etc.), as well as their coordinate transformations.

|  |  |
| --- | --- |
| COLLADAParser.h | MeshEditor |
| #include <string>  #include "Model.h"  std::unique\_ptr<Model> loadModel(const std::string& filename);  void saveModel(const Model& model, const std::string& filename); | |

**Model** 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 other **Node**s.

|  |  |
| --- | --- |
| Model.h | MeshEditor |
| #include "Node.h"  class Model  {  public:  Model();  void attachNode(std::unique\_ptr<Node> node);  const std::vector<std::unique\_ptr<Node>>& getNodes() const;  private:  std::vector<std::unique\_ptr<Node>> nodes;  }; | |

A **Node** is a *tree node* 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.

*calcAbsoluteTransform* calculates absolute transformation matrix of the **Node** by applying all parents’ transformations recursively up by hierarchy.

*attachNode* adds *newNode* to the children list by moving input variable. It requires that *newNode* doesn’t have a parent. If this requirement is violated, then throw the *logic\_error* exception.

*deleteFromParent* pop back element from parent’s child list and deletes it. Don’t use pointer after calling this function.

*applyRelativeTransform* multiplies current relative transform by the input matrix. It is useful for some incremental changes to the transformation matrix.

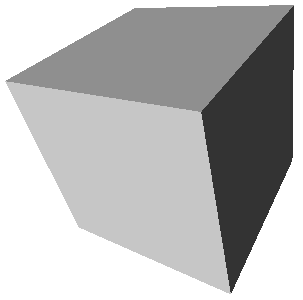
|  |  |
| --- | --- |
| Node.h | MeshEditor |
| #include <string>  #include <memory>  #include <glm/glm.hpp>  #include "Mesh.h"  class Node  {  public:  Node();  virtual ~Node();  void setName(const std::string& inName);  const std::string& getName() const;  void attachMesh(std::unique\_ptr<Mesh> inMesh);  Mesh\* getMesh() const;  void setParent(Node\* inParent);  Node\* getParent() const;  void setRelativeTransform(const glm::mat4& trf);  const glm::mat4& getRelativeTransform() const;  void applyRelativeTransform(const glm::mat4& trf);  const std::vector<std::unique\_ptr<Node>>& getChildren() const;    glm::mat4 calcAbsoluteTransform() const;  void attachNode(std::unique\_ptr<Node> newNode);  void deleteFromParent();  private:  // TODO  }; | |

Note that for the relations **point to, BUT does not own**, a pointer (\*) is used. Hence, in this case a parent points to the **Node**, but does not own it.

For relations **owns and point to** **std::unique\_ptr<T>** is used.

### COLLADA Format

COLLADA is a XML specification (schema) that stores information about models, materials, scenes, cameras, lights, etc. This information is then transferable between different 3D applications. The specification is designed in such a way to accommodate most of the mainstream 3D development tools. As a starting point to support COLLADA, it is necessary to implement only a subset of the [COLLADA Release 1.4.1](#COLLADASpec) specification during this Lab work. The portion of the specification that will be implemented is concerned with the geometric information of models.



The 3D cube model above is described by the COLLADA XML given below.

|  |
| --- |
| <collada>    <library\_geometries>      <geometry id="Cube-mesh">        <mesh>          <source id="Cube-mesh-positions">            <float\_array id="Cube-mesh-positions-array" count="24">              1 1 -1 1 -1 -1 -1 -1 -1 -1 1 -1 1 1 1 1 -1 1 -1 -1 1 -1 1 1            </float\_array>          </source>          <vertices id="Cube-mesh-vertices">            <input semantic="POSITION" source="#Cube-mesh-positions" />          </vertices>          <polylist material="Material1" count="6">            <input semantic="VERTEX" source="#Cube-mesh-vertices" offset="0"/>            <vcount>4 4 4 4 4 4 </vcount>            <p>0 1 2 3 4 7 6 5 0 4 5 1 1 5 6 2 2 6 7 3 4 0 3 7</p>          </polylist>        </mesh>      </geometry>    </library\_geometries>    <library\_visual\_scenes>      <visual\_scene id="Scene" name="Scene">        <node id="Cube" type="NODE">          <instance\_geometry url="#Cube-mesh"></instance\_geometry>  <translate>32 42 16</translate>        </node>      </visual\_scene>    </library\_visual\_scenes>  </collada> |

All the COLLADA information of the cube model is wrapped in the <collada> root node of the XML tree. The two children of the root node, <library\_geometries> and <library\_visual\_scenes>, describe the geometric information and instantiation of the cube respectively. When we look at the <library\_geometries> node it is easy to see that a geometry named ‘Cube-mesh’ is defined. This mesh contains 8 vertices (the 24 float values are the x, y and z dimensions of these vertices), and 6 four-sided polygons created from these vertices. The integer values in the <p> tag are indexes into the vertex array. The values in the <vcount> node indicates that the <p> tag should be interpreted as defining 6 four-sided polygons. In the <library\_visual\_scenes> section of the COLLADA, we use the <node> tag, to instantiate a "Cube-mesh" geometry and place it to the (32;42;16) location in the **WCS**. The above example should result into **Model** object that contains one **Node** with relative transformation matrix that corresponds to translation({32;42;16}) and **Mesh** with encapsulates HalfEdgeTable that consist of 12 faces, 8 vertices and 38 half-edges.

## Task 2

**Application** is the main class that manages 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** shouldn’t outlive the **Model**. *run* function starts an infinite loop that lasts until the user presses **ESC**.

|  |  |
| --- | --- |
| Application.h | MeshEditor |
| #include <memory>  #include <string>  #include <vector>  #include "View.h"  #include "Model.h"  class Application  {  public:  Application(const std::string& filename);  View\* createView(const std::string& title, uint32\_t width, uint32\_t height);  void run();  private:  std::unique\_ptr<IRenderSystem> renderSystem;  std::vector<std::unique\_ptr<View>> views;  // TODO  }; | |

## View

The *update* function does rendering of the **Model**. In order to implement rendering procedure it is necessary recursively traverse Nodes in the Model and get transformation of the node using *Node::calcAbsoluteMatrix* in the scene and pass it to *IRenderSystem::setWorldMatrix*, after that call *Mesh::render*. Inside, **View** owns pointer to the **IWindow** interface and window should be created inside **View** constructor. Method *raycast* will be explained while implementing **DeleteFaceOperator**.

|  |  |
| --- | --- |
| View.h | MeshEditor |
| #include <memory>  #include <vector>  #include <string>  #include "Viewport.h"  #include "Model.h"  #include "OperatorDispatcher.h"  #include "../Interfaces/RenderSystem.h"  class View  {  public:  View(IRenderSystem& rs, const std::string& title, uint32\_t width, uint32\_t height);  void update();  void setModel(Model\* model);  Model\* getModel() const;  void addOperator(KeyCode enterKey, KeyCode exitKey, std::unique\_ptr<Operator> op);  void addOperator(ButtonCode button, std::unique\_ptr<Operator> op);  void addOperator(KeyCode key, std::unique\_ptr<Operator> op);  template<class Lambda>  void addOperator(KeyCode key, Lambda lambda)  {  //TODO  }    Viewport& getViewport();  const Viewport& getViewport() const;  std::vector<Contact> raycast(double x, double y, FilterValue filterValues) const;  private:  OperatorDispatcher operatorDispatcher;  // TODO  }; | |

**View** contains four different *AddOperator* functions which can be used in the following way:

view->addOperator(ButtonCode::Button\_Left, std::make\_unique<PanOperator>());

Activates pan operator when user presses left key. Deactivates it when user releases left key.

view->addOperator(KeyCode::E, KeyCode::ESCAPE, std::make\_unique<EditMeshOperator>());

Activates **EditMeshOperator** when user presses E key and deactivates it when user presses ESC.

view->addOperator(KeyCode::F1, [](View& view, Action, Modifier) {view.getViewport().getCamera().setFrontView(); });

Sets Front View operator when user presses F1 key and deactivates it immediately.

All needed operators are set inside the *Application::createView* function.

**OperatorDispatcher** is a class that listens for keyboard and mouse events and fires operator that matches input. **OperatorDispatcher** contains three private functions that only visible to View class and are called in the View constructor:

|  |
| --- |
| window->setKeyCallback([&](KeyCode key, Action action, Modifier mods)  {  operatorDispatcher.processKeyboardInput(\*this, key, action, mods);  });  window->setCursorPosCallback([&](double x, double y)  {  operatorDispatcher.processMouseMove(\*this, x, y);  });  window->setMouseCallback([&](ButtonCode button, Action action, Modifier mods, double x, double y)  {  operatorDispatcher.processMouseInput(\*this, button, action, mods, x, y);  }); |

Implement class **OperatorDispatcher**. Start with the one-click operators as they are easiest to implement. Then think how to implement complex operators that starts and ends with the different keys. Note, that if operator is bind with the key that already in use then throw *logic\_error* exception. For example, the chain of the following calls: addOperator(KeyCode::F1, myOperator1); addOperator(KeyCode::F1, KeyCode::F2, myOperator2); should lead to the exception.

|  |  |
| --- | --- |
| OperatorDispatcher.h | MeshEditor |
| #include "Operator.h"  #include <vector>  #include <stack>  #include <map>  class OperatorDispatcher  {  public:  friend class View;  void addOperator(KeyCode enterKey, KeyCode exitKey, std::unique\_ptr<Operator> op);  void addOperator(ButtonCode button, std::unique\_ptr<Operator> op);  void addOperator(KeyCode key, std::unique\_ptr<Operator> op);  template<class Lambda>  void addOperator(KeyCode key, Lambda lambda)  {  //TODO  }  private:  void processMouseInput(View& view, ButtonCode button, Action action, Modifier mods, double x, double y);  void processMouseMove(View& view, double x, double y);  void processKeyboardInput(View& view, KeyCode key, Action action, Modifier mods);  //TODO  }; | |

In order to perform some kind of an action on the **Model** and the **View**, it is needed to specify the **Operator**:

|  |  |
| --- | --- |
| Operator.h | MeshEditor |
| #include "View.h"  #include "../Interfaces/Window.h"  class Operator  {  public:  virtual ~Operator() {}  virtual void onEnter(View&) {}  virtual void onExit(View&) {}  virtual void onMouseMove(View& view, double x, double y) {}  virtual void onMouseInput(View& view, ButtonCode button, Action action, Modifier mods, double x, double y) {}  virtual void onKeyboardInput(View& view, KeyCode key, Action action, Modifier mods) {}  }; | |

The *onEnter/onExit* functions are called respectively when the **Operator** is started/finished.

## Task 3

Implement **DeleteFaceOperator**. The logic goes as following: find the closest contact to the camera (intersected face and pointer to the node) between user ray and the tree nodes. If there is an intersection exist than take intersected node and intersected face and then call deleteFace in order to delete intersected face.

|  |
| --- |
| class DeleteFaceOperator : public Operator  {  void onMouseInput(View& view, ButtonCode button, Action action, Modifier mods, double x, double y) override  {  if (action == Action::Release)  {  std::vector<Contact> contacts = view.raycast(x, y, FilterValue::Node);  if (contacts.empty())  return;  Contact& contact = contacts.front();  Node\* node = contact.node;  if (node)  {  Mesh\* mesh = node->getMesh();  if (mesh)  mesh->deleteFace(contact.face);  }  }  } }; |

### Contact Detection

It is necessary to implement *View::raycast* function. This function takes as input mouse position and filter value. Filter value can be *Node* or *Manipulator*:

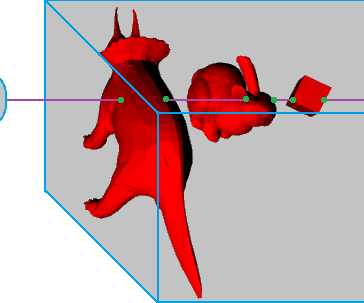
|  |  |
| --- | --- |
| FilterValue.h | MeshEditor |
| enum class FilterValue : uint32\_t  {  Node = 1,  Manipulator = 2,  NM = (uint32\_t)(Node | Manipulator), //Node and Manipulator  }; | |

It returns a sorted list of *Contact*s between ray and a **Model** (tree node). Contacts are sorted by distance to the observer along the ray.

Contact contains the following fields: face – interested face and corresponding intersected node.

|  |  |
| --- | --- |
| Contact.h | MeshEditor |
| struct Contact  {  HalfEdgeLib::FaceHandle face;  Node\* node;  }; | |

Green dotes are contacts in the figure below. Purple line is the cursor ray. Blue box is a clip space.



### Brute force algorithm

Simple brute force algorithm requires checking all faces in all Node’ Meshes that result in big complexity and low performance:

*For each i-th object in the Scene:*

*Calculate intersection between ray and i-th object:*

*If there is intersection then add intersected face and node to the contacts list*

*Sort contacts by distance to the observer along the ray*

Let’s divide raycast algorithm into two phases:

1. [The Broad phase](#BroadphaseVSNarrowphase) – all Node’ geometries are approximated with simple convex geometries like Bounding Boxes or Spheres for example. Now checking the ray for a simple geometry is cheaper than checking each face in the **Mesh**. The result of Broad phase is a list of Nodes whose bounding volumes intersect with the ray.

### The Narrow phase takes as an input the result of the Broad phase, and then for each node in the list it checks the ray against the Node’s Mesh.

### Broad phase

To speed up a process of finding potential candidates for the intersecting ray and tree node consider the following trick:

1. Build bonding boxes for each node. Each **Mesh** has *getBoundingBox* function that returns bounding box
2. Calculate intersection between ray and bounding boxes of the nodes. Note it is necessary to consider transformation matrix in the process of bounding box calculation

Now, instead of checking all faces in all meshes, it is necessary to check their bounding boxes only. As the outcome of the Broad phase there are the list of potential candidates (Nodes).

### Narrow phase

Now it is necessary to check each candidate’s (from the previous step) **Mesh** against the ray and [find the closest intersected](#RayTriangleIntersection) triangle to the camera. As the output a list of intersected triangles is created that are sorted by distance to the camera. This is called Narrow phase.

Now do filtering by the *filterValue*. If user asked for checking intersection only with Manipulators then remove all nodes from the list that are not Manipulators. The inverse logic works for Nodes: if **Node** is not a **Manipulator** than it should be in the result list.

Such filtering is needed in order to check if user selects **Node** (geometry) or **Manipulator** and based on the type of the selection provides needed manipulations.

## Exercise

1. While implementing *Node::calcAbsoluteTransform* think how it can be optimized for the large hierarchies
2. Implement **Pan** operator based on the code from lab work 2 (see Exercises section)
3. Implement **TrackBall** operator based on the code from lab work 2 (see Exercises section)
4. Implement **Broad phase** using [Octree](#Octree). Measure the performance gain after adding **Octree**. Think how you can add **Octree** without removing brute force algorithm for checking list of bounding boxes

## Resources and Notes

1. <https://www.khronos.org/files/collada_spec_1_4.pdf> - DAE format specification
2. <https://en.wikipedia.org/wiki/Octree> - Octree

1. <http://planning.cs.uiuc.edu/node214.html> - Broad phase vs Narrow phase

1. <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates> – Barycentric coordinates and ray-triangle intersection