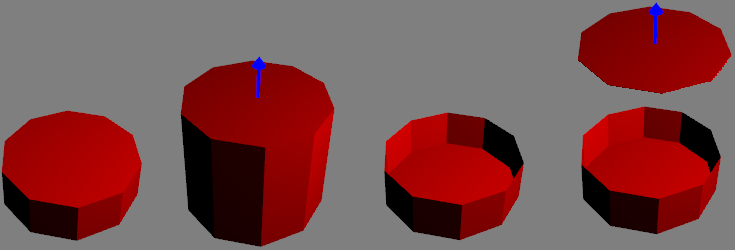
# Lab 3

## Overview

# This lab work is dedicated to the *half-edge data structure* implementation. As it was found out in Lab work 1, *triangle soup* is used to render a mesh. Such a data structure is very effective and optimized for all video cards. But this data structure has one important drawback - the absence of information regarding the topology. For example, a programmer cannot query all triangles that are adjacent to a given triangle. The topology information is important during mesh manipulations such as moving or deleting triangles.



If a mesh is presented using *triangle soup*, then when moving the given triangle, the adjoining triangles wouldn’t move and it would result a hole in the mesh (see picture above). Also, the analysis of the adjacent triangles will require additional work in most cases, which will lead to a huge performance hit. To avoid that half-edge data structure can be used.

## Objective

1. To reorganize previous code by separating solution into two modules – **MeshEditor** (exe) and **GLRenderSystem** (dll)
2. To implement the **HalfEdge** data structure. Add another module **HalfEdge** (static lib)
3. To write a **Mesh** class that wraps the half-edge data structure and implements the *render* method to render the mesh. In addition, **Mesh** should provide functions for deleting and transforming triangles.

## Infrastructure

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **.** | **MeshEditor** | **GLRenderSystem** | **HalfEdge** | **Interfaces** | **ThirdParty** |
| MeshEditor  **GLRenderSystem**  **HalfEdge**  Interfaces  ThirdParty  MeshEditor.sln | 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 |

## Task 1

Add to your solution a new project **GLRenderSystem** with a dynamic library type and move **GLWindow** and **GLRenderSystem** with the needed dependencies (glfw, glad) there. Move glad.h, glad.c, khrplatform.h files into **GLRenderSystem** folder and add them to **GLRenderSystem.vcxproj** project.

It is necessary to define the appropriate **IWindow** and **IRenderSystem** interfaces that will be implemented in **GLRenderSystem.dll**. To do this, create **Window.h** and **RenderSystem.h** in the Interfaces folder in the project's root directory. Add the **Exports.h**, **Exports.cpp** file to the **GLRenderSystem** project with the following functions:

|  |  |
| --- | --- |
| Exports.h | GLRenderSystem |
| #include <string>  #ifdef OGL\_RENDER\_SYSTEM\_EXPORT  #define OGL\_RENDER\_SYSTEM\_API \_\_declspec(dllexport)  #else  #define OGL\_RENDER\_SYSTEM\_API \_\_declspec(dllimport)  #endif  class IRenderSystem;  class IWindow;  extern "C" OGL\_RENDER\_SYSTEM\_API IRenderSystem\* createRenderSystem();  extern "C" OGL\_RENDER\_SYSTEM\_API IWindow\* createWindow(const std::string& title, uint32\_t width, uint32\_t height);  extern "C" OGL\_RENDER\_SYSTEM\_API void waitEvents();  extern "C" OGL\_RENDER\_SYSTEM\_API void swapDisplayBuffers(IWindow\* window);  extern "C" OGL\_RENDER\_SYSTEM\_API bool windowShouldClose(IWindow\* window); | |

and be sure to add OGL\_RENDER\_SYSTEM\_EXPORT; in the preprocessor definitions in the **GLRenderSystem** project settings. These free functions are needed to create **IRenderSystem** and **IWindow** in the main **MeshEditor** module. Later, **View** would be implemented with the help of them.

|  |  |
| --- | --- |
| IWindow.h | GLRenderSystem |
| #include <functional>  enum class Modifier  {  NoModifier = 0,  Shift = 1,  Control = 2,  Alt = 4,  Super = 8,  };  enum class Action  {  Release = 0,  Press = 1,  Repeat = 2,  };  enum class ButtonCode  {  Button\_0 = 0,  // repeats all buttons codes from the glfw header  };  enum class KeyCode  {  UNKNOWN = -1,  Space = 32,  // repeats all key codes from the glfw header  };  class IWindow  {  public:  using KeyCallback = std::function<void(KeyCode, Action, Modifier)>;  using CursorPosCallback = std::function<void(double, double)>;  using MouseCallback = std::function<void(ButtonCode, Action, Modifier, double, double)>;  using ScrollCallback = std::function<void(double, double)>;  virtual ~IWindow() {}  virtual uint32\_t getWidth() const = 0;  virtual uint32\_t getHeight() const = 0;  virtual void setKeyCallback(const KeyCallback& callback) = 0;  virtual void setCursorPosCallback(const CursorPosCallback& callback) = 0;  virtual void setMouseCallback(const MouseCallback& callback) = 0;  virtual void setScrollCallback(const ScrollCallback& callback) = 0;  }; | |

|  |  |
| --- | --- |
| IRenderSystem.h | GLRenderSystem |
| #include <vector>  #include <glm/glm.hpp>  struct Vertex  {  glm::vec3 position;  glm::vec3 normal;  };  class IRenderSystem  {  public:  virtual ~IRenderSystem() {}  virtual void init();  virtual void clearDisplay(float r, float g, float b) = 0;  virtual void setViewport(double x, double y, double width, double height) = 0;  virtual void renderTriangleSoup(const std::vector<Vertex>& vertices) = 0;  virtual void setupLight(uint32\_t index, glm::vec3 position, glm::vec3 Ia, glm::vec3 Id, glm::vec3 Is) = 0;  virtual void turnLight(uint32\_t index, bool enable) = 0;  virtual void setWorldMatrix(const glm::mat4& matrix) = 0;  virtual const glm::mat4& getWorldMatrix() const = 0;  virtual void setViewMatrix(const glm::mat4& matrix) = 0;  virtual const glm::mat4& getViewMatrix() const = 0;  virtual void setProjMatrix(const glm::mat4& matrix) = 0;  virtual const glm::mat4& getProjMatrix() const = 0;  }; | |

The corresponding **GLWindow /** **GLRenderSystem** classes are now inherited from **IWindow / IRenderSystem** interfaces. **GLRenderSystem** is loaded dynamically when MeshEditor starts. To implement dynamic loading of **GLRenderSystem.dll** add class **DynamicLibrary** to **MeshEditor** project:

|  |  |
| --- | --- |
| DynamicLibrary.h | MeshEditor |
| #include <string>  class DynamicLibrary  {  public:  DynamicLibrary(const std::string& name);  ~DynamicLibrary();  void\* getSymbol(const std::string& symbolName) const;  template<class T>  T getSymbol(const std::string& symbolName) const  {  return (T)getSymbol(symbolName);  }  private:  void\* instance;  }; | |
| DynamicLibrary.cpp | MeshEditor |
| #include "DynamicLibrary.h"  #if PLATFORM == PLATFORM\_WIN32  #define WIN32\_LEAN\_AND\_MEAN  #if !defined(NOMINMAX) && defined(\_MSC\_VER)  #define NOMINMAX  #endif  #include <windows.h>  #endif  DynamicLibrary::DynamicLibrary(const std::string& name)  : instance(nullptr)  {  #if PLATFORM == PLATFORM\_WIN32  instance = (void\*)LoadLibrary(name.c\_str(), NULL, 0);  #endif  }  DynamicLibrary::~DynamicLibrary()  {  #if PLATFORM == PLATFORM\_WIN32  FreeLibrary((HMODULE)instance);  #endif  }  void\* DynamicLibrary::getSymbol(const std::string& symbolName) const  {  #if PLATFORM == PLATFORM\_WIN32  return (void\*)GetProcAddress((HMODULE)instance, symbolName.c\_str());  #endif  return nullptr;  } | |

and be sure to add PLATFORM=PLATFORM\_WIN32; in the preprocessor definitions in the MeshEditor project settings. *Note that code listing above works only for Windows platform. In order to implement dynamic loading, it is necessary to change PLATFORM macro and provide implementation using ifdef/elseif directives in DynamicLibrary.cpp.*

As result, MeshEditor may use different **RenderSystem**, which can be implemented under DirectX, for example, replacing just one line at program startup.

|  |  |
| --- | --- |
| main.cpp | MeshEditor |
| #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtx/transform.hpp>  #include "../Interfaces/Window.h"  #include "../Interfaces/RenderSystem.h"  #include "DynamicLibrary.h"  void renderScene(RenderSystem& rs)  {  // TODO  }  int main()  {  DynamicLibrary dll("GLRenderSystem32.dll");  auto createRenderSystem = dll.getSymbol<IRenderSystem\* (\*)()>("createRenderSystem");  auto createWindow = dll.getSymbol<IWindow\* (\*)(const std::string& title, uint32\_t width, uint32\_t height)>("createWindow");  auto waitEvents = dll.getSymbol<void(\*)()>("waitEvents");  auto swapDisplayBuffers = dll.getSymbol<void(\*)(IWindow\* window)>("swapDisplayBuffers");  auto windowShouldClose = dll.getSymbol<bool(\*)(IWindow\* window)>("windowShouldClose");  IRenderSystem\* rs = createRenderSystem();  IWindow\* window = createWindow("myWindow", 640, 480);  rs->init();  rs->setupLight(0, glm::vec3{0,5,0}, glm::vec3{1,0,0}, glm::vec3{0,1,0}, glm::vec3{0,0,1});  rs->turnLight(0, true);  glm::mat4 viewMatrix = glm::translate(glm::mat4(), glm::vec3(0.0f, 0.0f, -10.0f));  rs->setViewMatrix(viewMatrix);  glm::mat4 projMatrix = glm::perspective(glm::radians(60.0f), 640.0f / 480.0f, 0.1f, 500.f);  rs->setProjectionMatrix(projMatrix);  while (windowShouldClose(window)  {  rs->setViewport(0, 0, window->getWidth(), window->getHeight());  rs->clearDisplay(0.5f, 0.5f, 0.5f);  renderScene(\*rs);  swapDisplayBuffers(window);  waitEvents();  }  delete window;  delete rs;  return 0;  } | |

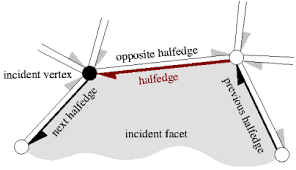
## Task 2

### The Half-edge Data Structure

For simple mesh operations (e.g., loading a mesh from disk and drawing it on screen), one can use a mesh data structure like triangle soup. For many geometry processing tasks this representation is no longer sufficient. In this lab work, half-edge data structure is used, which provides a good tradeoff between simplicity and sophistication.

### HalfEdge

The basic idea behind the HalfEdge data structure is that in addition to the vertices, edges, and faces that make up a polygon mesh, there is an entity called a **half-edge**. Each edge consists of two half-edges: directed edges between the same two vertices, pointing in opposite directions. Each half-edge holds pointer to its *twin* half-edge, the face the half-edge is part of, the *next* half-edge of that face in CCW order, and the vertex the half-edge it points to.



If face equals -1, then HalfEdge is a boundary.

struct HalfEdgeHandle { int64\_t index = -1; };

struct VertexHandle { int64\_t index = -1; };

struct FaceHandle { int64\_t index = -1; };

struct HalfEdge

{

//the face it belongs to. invalid (== -1) in case of boundary edge

FaceHandle fh;

//the vertex it points to. always valid

VertexHandle dst;

HalfEdgeHandle twin;

//next HalfEdge in CCW order. always valid

HalfEdgeHandle next;

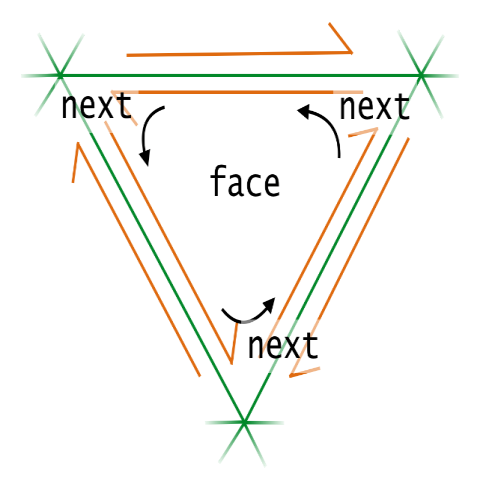
//can be stored for the optimization purposes. For the triangle meshes prev = next->next->next

HalfEdgeHandle prev;

};

### Face

While doing *heh=next(heh)* operation step by step the chain of edges would be described. This chain of edges forms the **Face**. The **Face** contains a reference to arbitrarily **HalfEdge** which defines the face.



struct Face

{

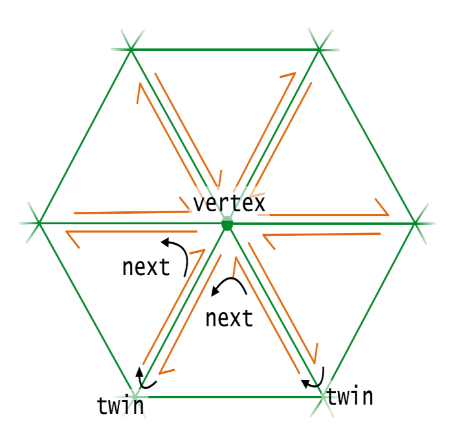
//one of the HalfEdges bounding it. always valid

HalfEdgeHandle heh;

};

### Vertex

It is assumed that a vertex has a reference to HalfEdge(heh). While doing heh = next(twin(heh)) operation step by step the chain of faces will describe the vertex. The vertex object contains a reference to just one of the outgoing HalfEdges which defines it, selected arbitrarily.



struct Vertex

{

//outgoing HalfEdge that starts at this vertex. It is == -1 in case of isolated vertex

HalfEdgeHandle heh;

};

Let’s define main helper operations (*deref, handle, next, twin, sourceVertex, destVertex)* that used in half-edge data structure. Note that **HalfEdgeHandle**, **VertexHandle** and **FaceHandle** are simple wrappers around **int** which is index to *halfEdges* array. This fact can be used in order implement *deref/handle* functions efficiently. Think about **Handle** as C pointer. *deref* function returns value by handle, and *handle* does the opposite thing – returns ‘pointer’ to the **Vertex**, **Edge** or **Face** entity. Consider handle is null if it equals to -1.

|  |
| --- |
| std::vector<Vertex> vertices;  std::vector<HalfEdge> halfEdges;  std::vector<Face> faces;  HalfEdge& deref(HalfEdgeHandle heh)  {  return halfEdges[heh.index];  }  HalfEdgeHandle handle(const HalfEdge& he) const  {  return { static\_cast<int>(&he - &halfEdges[0]) };  }  HalfEdgeHandle next(HalfEdgeHandle heh) const  {  return deref(heh).next;  }  HalfEdgeHandle twin(HalfEdgeHandle heh) const  {  return deref(heh).twin;  }  VertexHandle destVertex(HalfEdgeHandle heh) const  {  return deref(heh).dst;  }  VertexHandle sourceVertex(HalfEdgeHandle heh) const  {  return destVertex(twin(heh));  } |

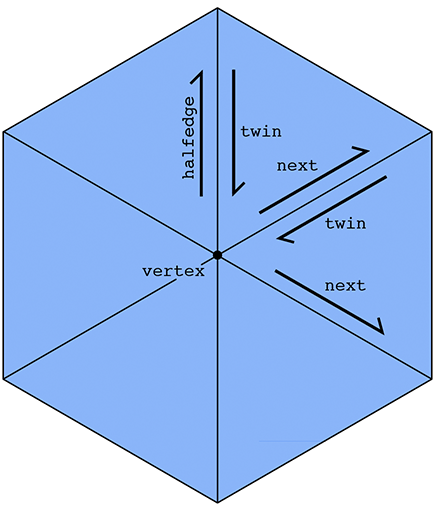
The similar functions (except *sourceVertex* and *destVertex*) are needed for **Face** and **Vertex** data structures.

### Traversal operations

As an example - visit all the vertices of a given **Face**. The face's **HalfEdge**(heh) is taken, and then *next* pointer is moved until the initial **HalfEdge** is met.

|  |
| --- |
| HalfEdgeHandle heh = deref(fh).heh;  HalfEdgeHandle start\_heh = heh;  HalfEdgeHandle next\_heh = heh;  do  {  visit(dst(next\_heh)); //do something with the vertex  next\_heh = next(next\_heh);  } while (next\_heh != start\_heh); |

Another example is visiting all the vertices adjacent to a given **Vertex**. Start by getting its outgoing HalfEdge, then its *twin*, then its *next* **HalfEdge**; this final **HalfEdge** will also point out of **Vertex**, but it will point toward a different vertex than the first HalfEdge. By repeating this process, all the neighboring vertices are visited:



|  |
| --- |
| HalfEdgeHandle heh = deref(vh).heh;  HalfEdgeHandle start\_heh = heh;  HalfEdgeHandle next\_heh = heh;  do  {  visit(dst(next\_heh)); //do something with the vertex  next\_heh = next(twin(next\_heh));  } while (next\_heh != start\_heh); |

An interesting consequence of the **HalfEdge** representation is that any valid **HalfEdge** mesh must be manifold and orientable.

|  |  |
| --- | --- |
| HalfEdgeTable.h | HalfEdge |
| #include <glm/glm.hpp>  struct HalfEdgeHandle { int64\_t index = -1; };  struct VertexHandle { int64\_t index = -1; };  struct FaceHandle { int64\_t index = -1; };  struct HalfEdge  {  //the face it belongs to. invalid (== -1) in case of boundary edge  FaceHandle fh;  //the vertex it points to. always valid  VertexHandle dst;  HalfEdgeHandle twin;  //next HalfEdge in CCW order. always valid  HalfEdgeHandle next;  //can be stored for the optimization purposes. For the triangle meshes prev = next->next->next  HalfEdgeHandle prev;  };  struct Face  {  //one of the HalfEdges bounding it. always valid  HalfEdgeHandle heh;  };  struct Vertex  {  //outgoing HalfEdge that starts at this vertex. It is == -1 in case of isolated vertex  HalfEdgeHandle heh;  };  class HalfEdgeTable  {  public:  //Main functions that needed for building HalfEdge data structure  VertexHandle addVertex(glm::vec3 position); // returns index of the added vertex  FaceHandle addFace(VertexHandle vh0, VertexHandle vh1, VertexHandle vh2);  FaceHandle addFace(VertexHandle vh0, VertexHandle vh1, VertexHandle vh2, VertexHandle vh3);  void deleteFace(FaceHandle fh);  HalfEdgeHandle prev(HalfEdgeHandle heh) const;  HalfEdgeHandle next(HalfEdgeHandle heh) const;  HalfEdgeHandle twin(HalfEdgeHandle heh) const;  VertexHandle destVertex(HalfEdgeHandle heh) const;  VertexHandle sourceVertex(HalfEdgeHandle heh) const;  HalfEdge& deref(HalfEdgeHandle vh);  const HalfEdge& deref(HalfEdgeHandle vh) const;  HalfEdgeHandle handle(const HalfEdge& v) const;  Vertex& deref(VertexHandle vh);  const Vertex& deref(VertexHandle vh) const;  VertexHandle handle(const Vertex& v) const;  Face& deref(FaceHandle fh) const;  const Face& deref(FaceHandle fh) const;  FaceHandle handle(const Face& f) const;  const glm::vec3& getPoint(VertexHandle handle) const;  void setPoint(VertexHandle handle, glm::vec3 data);  const glm::vec3& getStartPoint(HalfEdgeHandle handle) const;  void setStartPoint(HalfEdgeHandle handle, glm::vec3 data);  const glm::vec3& getEndPoint(HalfEdgeHandle handle) const;  void setEndPoint(HalfEdgeHandle handle, glm::vec3 data);  private:  // TODO  }; | |

Example usage:

|  |
| --- |
| HalfEdgeTable createTriangle()  {  HalfEdgeTable halfEdgeTable;  VertexHandle vh0 = halfEdgeTable.addVertex({ 0.0, 0.0, 1.0 });  VertexHandle vh1 = halfEdgeTable.addVertex({ 1.0, 0.0, 1.0 });  VertexHandle vh2 = halfEdgeTable.addVertex({ 1.0, 1.0, 1.0 });  halfEdgeTable.addFace(vh0, vh1, vh2);  return halfEdgeTable;  } |

## Task 3

Finally, implement the **Mesh** class that will be rendered in the **IWindow** window.

|  |  |
| --- | --- |
| Mesh.h | MeshEditor |
| #include "../HalfEdge/HalfEdge.h"  struct bbox  {  glm::vec3 min;  glm::vec3 max;  };  class Mesh  {  public:  Mesh(const HalfEdgeTable& halfEdgeTable);  void render(IRenderSystem& rs);  const bbox& getBoundingBox() const;  void applyTransformation(FaceHandle fh, const glm::mat4& trf);  void deleteFace(FaceHandle fh);  const HalfEdgeTable& getHalfEdgeTable() const;  private:  // TODO  }; | |

Implement all functions except *getBoundingBox* (it will be explained in Lab 4). Note, that *applyTransformation* function applies transformation matrix to all vertices of the given face (fh).

Then the main rendering code will be rewritten as follows:

|  |  |
| --- | --- |
| main.cpp | MeshEditor |
| #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtx/transform.hpp>  #include "../Interfaces/Window.h"  #include "../Interfaces/RenderSystem.h"  #include "../HalfEdge/HalfEdgeTable.h"  #include "DynamicLibrary.h"  #include "Mesh.h"  HalfEdgeTable createCube()  {  // TODO  }  int main()  {  DynamicLibrary dll("GLRenderSystem.dll");  auto createRenderSystem = dll.getSymbol<IRenderSystem\* (\*)()>("createRenderSystem");  auto createWindow = dll.getSymbol<IWindow\* (\*)(const std::string& title, uint32\_t width, uint32\_t height)>("createWindow");  auto waitEvents = dll.getSymbol<void(\*)()>("waitEvents");  auto swapDisplayBuffers = dll.getSymbol<void (\*)(IWindow\* window)>("swapDisplayBuffers");  auto windowShouldClose = dll.getSymbol<bool (\*)(IWindow\* window)>("windowShouldClose");  IRenderSystem\* rs = createRenderSystem();  IWindow\* window = createWindow("myWindow", 640, 480);  rs->init();  rs->setupLight(0, glm::vec3{0,5,0}, glm::vec3{1,0,0}, glm::vec3{0,1,0}, glm::vec3{0,0,1});  rs->turnLight(0, true);  glm::mat4 viewMatrix = glm::translate(glm::mat4(), glm::vec3(0.0f, 0.0f, -10.0f));  rs->setViewMatrix(viewMatrix);  glm::mat4 projMatrix = glm::perspective(glm::radians(60.0f), 640.0f / 480.0f, 0.1f, 500.f);  rs->setProjectionMatrix(projMatrix);  HalfEdgeTable halfEdgeTable = createCube();  Mesh mesh(halfEdgeTable);  while (windowShouldClose(window))  {  rs->setViewport(0, 0, window->getWidth(), window->getHeight());  rs->clearDisplay(0.5f, 0.5f, 0.5f);  mesh.render(\*rs);  swapDisplayBuffers();  waitEvents();  }  delete rs;  delete window;  return 0;  } | |

## Exercises

1. Implement HalfEdgeTable meshCylinder(double R, double h, uint32\_t numSubdivisions);
2. Implement HalfEdgeTable meshCone(double R, double h, uint32\_t numSubdivisions);
3. Implement HalfEdgeTable meshTorus(double minorRadius, double majorRadius, uint32\_t majorSegments);
4. Implement HalfEdgeTable meshArrow(); using meshCylinder and meshCone functions. meshArrow should ‘look’ in the (0,0,1) direction.
5. Make sure that above implemented functions return correct built half-edge data structure. Try to change vertex position and see if there is hole.

## Resources and Notes

1. <http://kaba.hilvi.org/homepage/blog/halfedge/halfedge.htm> - half-edge structure
2. [The Half-Edge Data Structure](http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml) - another tutorial on the half-edge data structure
3. [OpenMesh](https://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf) - a different implementation than the one used in our lab work, but discusses some of the software design challenges associated with building a half-edge structure