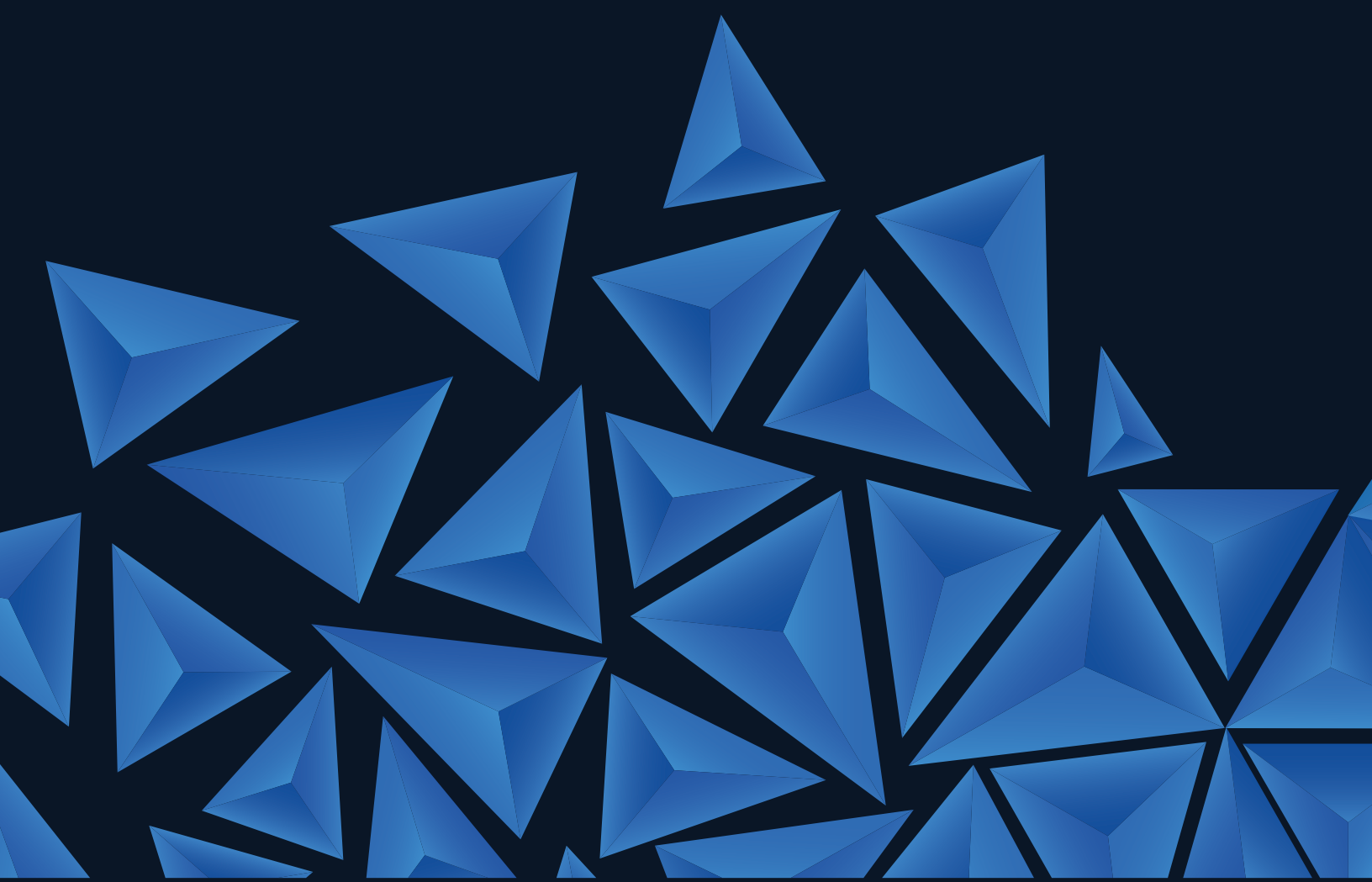
Computer Graphics Course

Developed by AMC Bridge and National University of Lviv Polytechnic, Institute of Computer Science and Information Technology, Department of Artificial Intelligence Systems



Introduction

Computer Graphics Course

Overview

This course aims to teach students the basics of CAD application development and main design patterns, including mastering such technologies as OpenGL, GLFW, COLLADA and basic terms like scene tree, manipulators, half-edge data structure, MVO pattern. The course application is a simple Mesh Editor tool.

Objective

The course objective is to develop from scratch Mesh Editor to manipulate the triangle meshes. The finished application loads and edits COLLADA mesh files that are now used by many major modeling packages and real-time graphics engines.

Conventions

This course uses the following typographical conventions to distinguish between different kinds of information:

- Names of the technologies and technical terms are used according to their owners, for instance: TinyXML.
- New terms and important words are in *italic*, for instance: follows the *model-view-operator (MVO)* pattern.
- Source code in the text is marked by monospace font `Courier New`, for instance: All the code listings do not have header guards (`#pragma once`, or `ifdef X_H/endif`).
- Names of keyboard keys are Capitalized, for instance: press D to run.
- The text to type at the command prompt is **bold**, for instance: At the command prompt type **MeshEditor rhino.dae**.
- Names of the files are in *italic*, for instance: to save the file to *rhino-modified.dae*.
- File types follow the file extensions rule of lowercase preceded with a dot, for instance: a part of the .cpp file.
- Links to external materials are underlined blue, for instance: MVO architecture used in HOOPS engine.

Structure of the Course

This course consists of 5 lab classes, which have theoretical and practical parts. Some labs have additional materials and extra exercises to get a deeper understanding of the subject.

There is a header file in each lab for you to implement. Your header should be the same as in the lab. **Note**: All the code listings do not have header guards (`#pragma once`, or `ifdef X_H/endif`), and you need to place them manually. Your job is to complete the code under the `TODO` comments and to write the corresponding .cpp file. Sometimes a part of the .cpp file is demonstrated to explain the task. Your implementation may differ from that demonstration.

Lab Work 0

Implementation of basic tessellation algorithms. Implementation of STL parser.

Lab Work 1

Creation of a window, geometry rendering, and processing of the inputs from a keyboard and a mouse.

Lab Work 2

Creation of `Camera` and `Viewport` classes.

Lab Work 3

Implementation of the half-edge data structure with the `Mesh` class based on `HalfEdgeTable` class.

Lab Work 4

Implementation of the Mesh Editor basis using the MVO pattern. Introduction to `Operator` class. Implementation of `DeleteFaceOperator`. On this stage, Mesh Editor can load/save COLLADA model, delete face, and display a mesh using various settings of camera and viewport.

Lab Work 5

Implementation of the `EditMeshOperator` using `TranslationManipulator`, and the `Translation` and the `Rotation` manipulators as a basis to assemble the `Triad` manipulator, used to implement `TransformMeshOperator`. At the final stage, Mesh Editor can edit faces, select a mesh, and change its transformation.

Requirements

Algebra, C++, OOP, algorithms, the first and second courses of computer graphics.

Mesh Editor

Functional Requirements

Mesh Editor loads a COLLADA file, its extension is .dae and then provides the tools to change a mesh: to move a triangle in the given direction or to delete a triangle. The editor outputs a modified mesh and can save it to a COLLADA file.

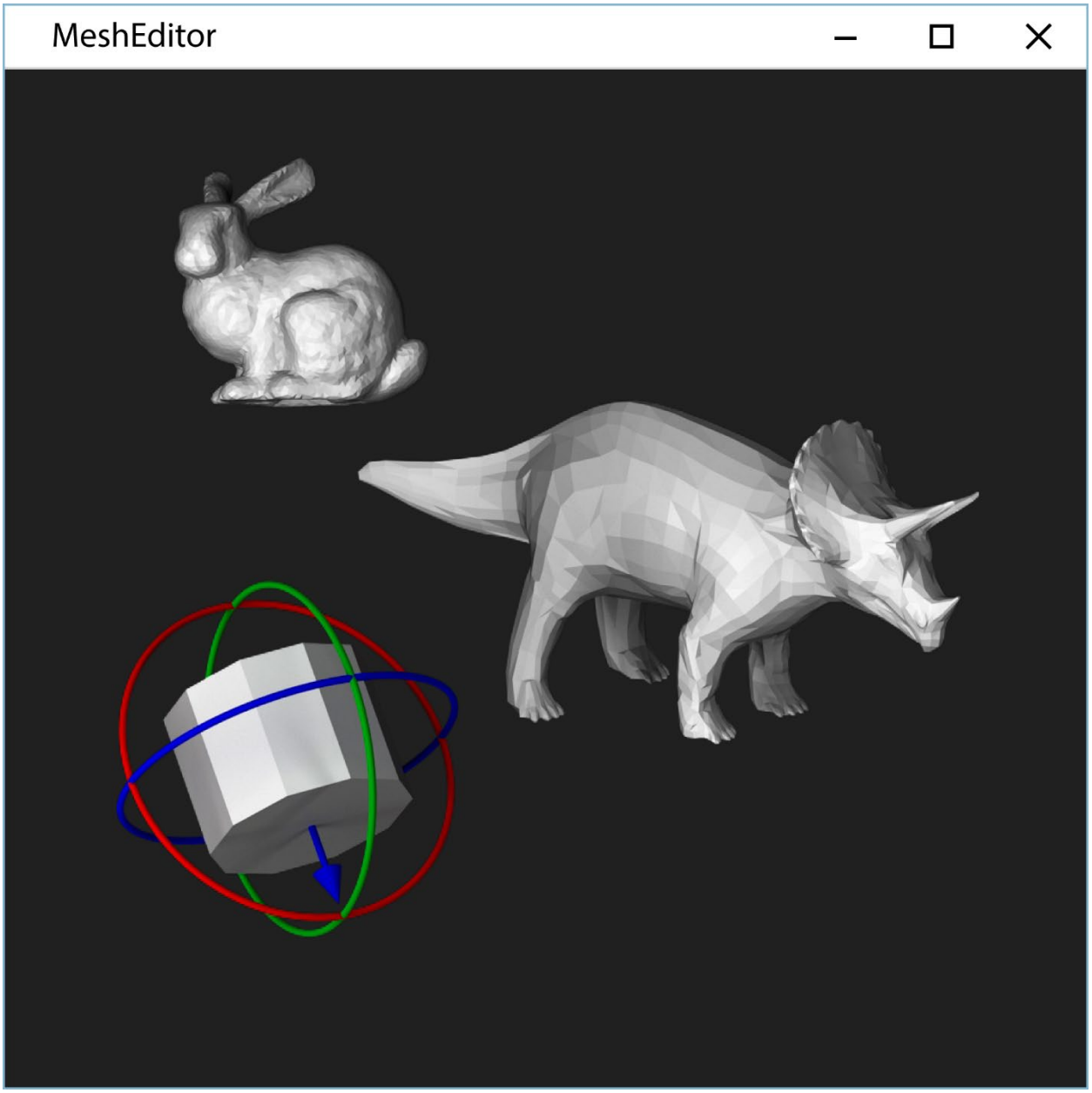


Figure 1. Mesh Editor

Commands

There are three types of commands:

1. Simple keyboard command of one action. To run it, just press a single key, and it stops when completed. For instance, to change the perspective projection to parallel, press F8 (see Table 1).
2. Complex keyboard command of two actions. To run it, press a single key, and to stop it when completed, press Esc. For instance, to delete faces, press D to run the Delete faces mode, then delete with the mouse the unnecessary faces, and finally press Esc to exit this mode (see Table 1).

Table 1. Keyboard Commands

Command	Run Key	Stop Key
Select and remove a face	D	ESC
Select and move a triangle in the given direction	E	ESC
Select a node and change its transformation with the triad	T	ESC
Display a model in a separate window	V	
Save the result to a COLLADA file, named *initial name + modified.dae*	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	

3. Mouse command of one continuous action. To run it, hold a click. For instance, to pan a camera, manipulate it while holding a left-click (see Table 2).

Table 2. Mouse Commands

Command	Hold to Run
Pan	Left-click
Zoom	Wheel button
Trackball	Right-click

A Workflow Example

1. At the command prompt, type **MeshEditor rhino.dae** to run Mesh Editor.
2. Press F9 to zoom to fit.
3. To adjust a camera in the required way, use a mouse and the F1-F9 keys.
4. Press E to run `EditMeshOperator`.
5. Select the triangle on the model and move it in the given direction.
6. Repeat step 5 until achieving the desired result.
7. Press Esc to stop `EditMeshOperator`.
8. Finally, to save the file to *rhino-modified.dae*, press S key.

Environment

MSVC 2015, OpenGL 2.1, GLFW, GLM, TinyXML

Theory

Implementation of Mesh Editor follows the *model-view-operator (MVO)* pattern. Many CAD engines use this pattern. According to it, the *View* displays the *Model* in a scene. *Operators* are external irritants that can change the *Model* and the *View* (see Fig. 2). Mouse and keyboard are examples of irritants. In other

words, the *Model* and the *View* react to irritations. You create your own reactions of the *Model* overriding the `Operator` class.

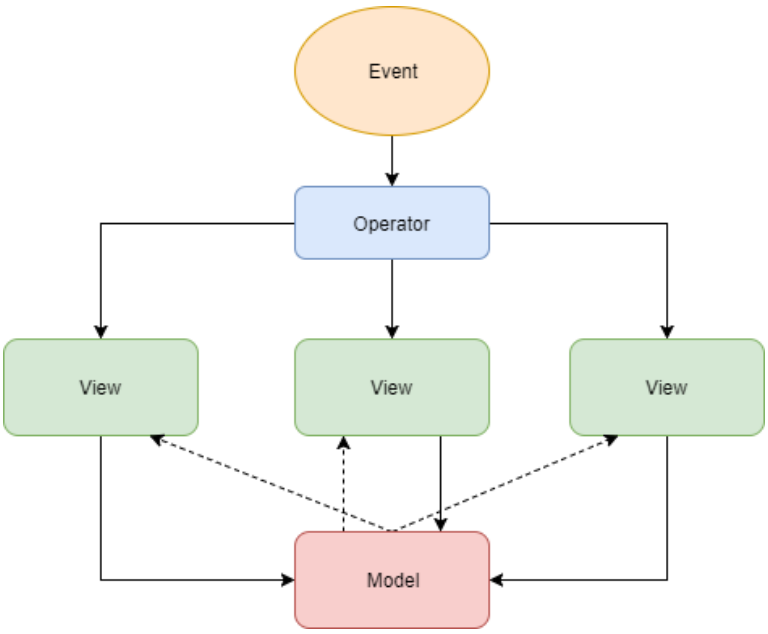


Figure 2. The MVO Pattern

This is an abstract description of an MVO implementation:

- Model is an array of triangles.
- The View is an OpenGL window drawing the array of triangles.
- The Operator is an abstract class with three methods: `onKeyPressed`, `onMousePressed` and `onMouseMove`.

This course requires to implement the operators to:

- Edit a mesh.
- Transform a mesh.
- Change the camera view.

Code Listing 1 shows an example of the application expression.

```
#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;
}
```

Code Listing 1. Application Expression

In the next lab, we will discuss and implement the fundamental classes that provide an overall picture of Mesh Editor architecture. Right now, it is important to understand the general idea of MVO pattern, not to focus on these classes in details. **Note**: MVO is similar to the *model-view-controller (MVC)* pattern. Consider reading the description of MVO architecture used in HOOPS engine.

MVO Classes

In Mesh Editor, *Model* is much more than an array of triangles. It is a *tree node* built according to the loaded .dae file. Each *Node* stores a *Mesh*, a relative matrix transformation, and an array of pointers to child *Nodes*.

The *View* displays a *Model*. For one *Model* there can be several *Views*. In our case, the *View* is an OpenGL window that can *only* render a *Model*.

The *Operator* encapsulates the application logic, thus runs all the operations on the *Model* and the *Camera*. It performs certain actions in response to the keyboard and the mouse events, for example, changes a camera position when F1-F9 pressed.

A *Node* is a tree node that has:

- Mesh
- Mesh transformation into the scene
- The array of child elements
- Pointer to the parent node

The tree node data structure is very useful for presenting hierarchical data structures, like the 3D Scene, because you can describe the relations between different 3D objects (see Fig. 2).

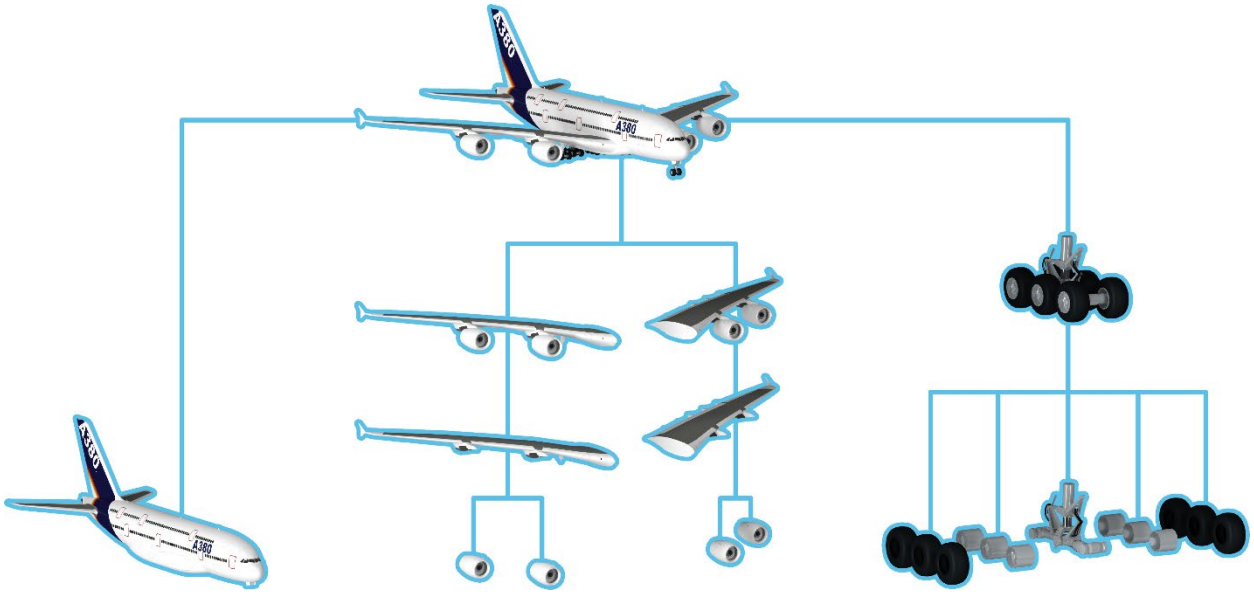


Figure 2. Tree node data structure

A Node should have a function to 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 (see Fig. 3). The *faces* usually consist of triangles but may also be composed of more general concave polygons or polygons with holes. In this course, there are only triangle meshes because this simplifies rendering.

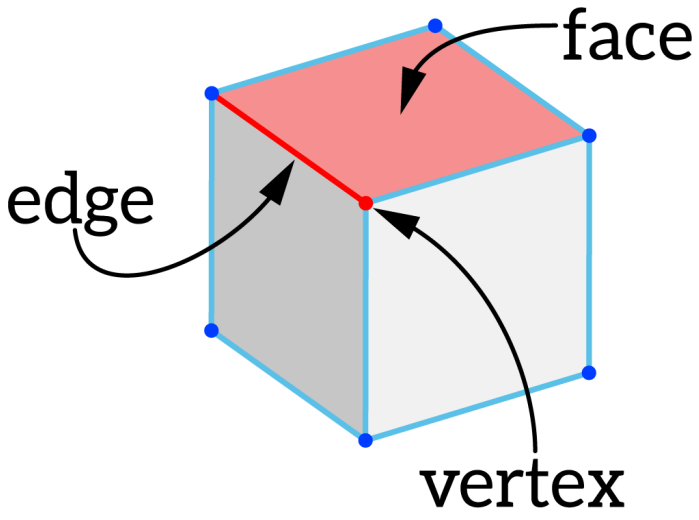


Figure 3. Mesh data structure

Half-edge data structure is necessary to build connections between the faces of the mesh. For example, it is important to have topology information when moving a face, a triangle in our case, because adjacent faces must move accordingly without producing any holes. To be valid, a half-edge mesh must be *manifold* and *orientable*. We use only this kind of meshes is in this course.

To be a *manifold,* a mesh must follow these two rules (see Fig.4):

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

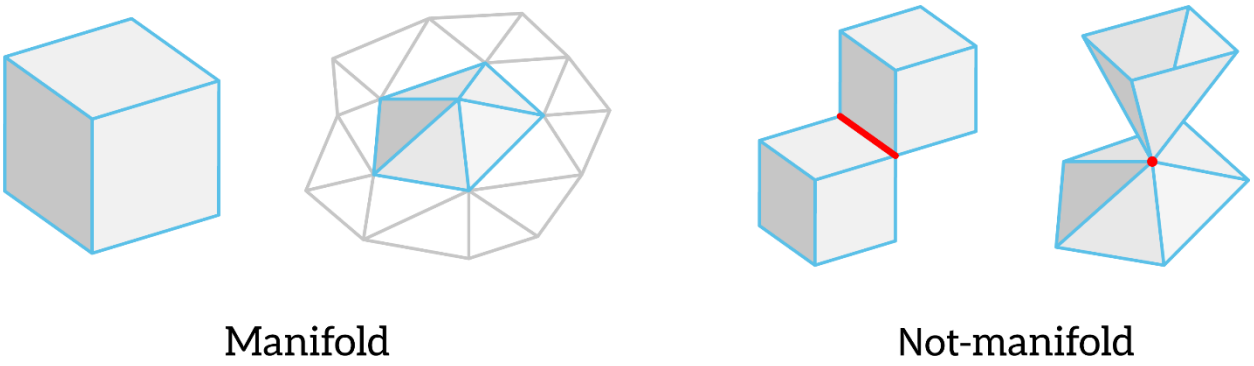


Figure 4. Mesh types

In Lab 3, we will implement the *half-edge data structure* in the `HalfEdgeTable` class.

The `Mesh` class encapsulates `HalfEdgeTable` data structure and its operations. Its `render` function converts `HalfEdgeTable` data into *triangle soup data structure* that `IRenderSystem` understands. This conversion is very expensive. Therefore, it must be applied only in two cases:

1. When a constructor receives `HalfEdgeTable`.
2. In all the operations that require changes in the `Mesh` topology, for example, `deleteFace`.

The `View` is an OpenGL window that renders a `Model` using `IRenderSystem` (Lab 1) and the specified `Camera` settings (Lab 2). Its `raycast` function finds an intersection between the cursor ray and the `Model` and returns an array of *contacts* (Lab 4). `Contact` is a structure that contains a pointer to the intersected face and its corresponding `Node`. The `update` function renders the `Model`. In the parentheses noted the lab with term explanation.

The `Application` class is the main class that manages the objects of an application, and the `main` function scope limits its lifetime. The `Application` class is used to create a `View` and limits its lifetime. A `View` should not outlive the `Model`.

Code Listing 2 shows an assembly of an application that can delete a face in a mesh.

```
#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;
}
```

Code Listing 2. Delete a face command

For the sake of simplicity, we did not discuss other important classes for Mesh Editor implementation in this chapter, like `IRenderSystem` or `Camera`. They are internal parts of MVO, and their explanations are in the Labs.

Table 2 shows the full structure of the finished application.

Table 3. Mesh Editor Structure

.	MeshEditor	GLRenderSystem	HalfEdge	Interfaces	ThirdParty
MeshEditor	TransformNode.h	GLRenderSystem.h	HalfEdge.h	IWindow.h	glm
GLRenderSystem	TransformNode.cpp	GLRenderSystem.cpp	HalfEdge.cpp	IRenderSystem.h	glfw
HalfEdge	EditMesh.h	GLWindow.h	HalfEdge.vcxproj		tinyxml2
Interfaces	EditMesh.cpp	GLWindow.cpp			
ThirdParty	Manipulator.h	Exports.h			
MeshEditor.sln	Manipulator.cpp	Exports.cpp			
	Triad.h	glad.h			
	Triad.cpp	glad.c			
	RotationManipulator.h	khrplatform.h			
	RotationManipulator.cpp	GLRenderSystem.vcxproj			
	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				

Resources and Notes

1. MVC pattern:
   https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller -
2. HOOPS/MVO Technical Overview:
   http://docs.techsoft3d.com/visualize/3df/21/2113/HoopsMVO/tech_overview/TechnicalOverview_C.html
3. The explanation of the s*cene graph* term. In the lab classes, we will implement *Tree node* instead for a simplicity:
   https://en.wikipedia.org/wiki/Scene_graph

Lab 0

Computer Graphics Course

Overview

In this Lab, you will learn how to create a simple console application that processes input and output ASCII STL files. The commands to be implemented are **Cube**, **Sphere**, and **Split**.

Objective

The objective of this Lab is to implement the following classes:

- `Application` that takes input arguments and registers the **Cube**, **Sphere**, and **Split** commands
- `STLParser` that reads and writes ASCII STL files
- `Cube` that creates a tessellated cube and outputs it to an ASCII STL file
- `Sphere` that creates a tessellated sphere and outputs it to an ASCII STL file
- `Split` that splits a tessellated input mesh into two and outputs them to ASCII STL files

The Lab Structure

This Lab has header files for you to implement. Your headers should be the same as in the Lab.

Note: All the code listings do not have header guards (`#pragma once`, or `ifdef X_H/endif`), and you need to place them manually.

Your task is to complete the code under the `TODO` comments and to create the corresponding .cpp file. Sometimes a part of the .cpp file is demonstrated to explain the task. Your implementation may differ from that demonstration.

Commands:

The variable arguments in the commands:

- *dbl_value* is a double value that conforms to the IEEE754 standard
- *string_value* is a simple C ASCII style string

Cube

The syntax:

Cube L = *dbl_value***, origin = (***dbl_value***,** *dbl_value***,** *dbl_value***), filepath = "***string_value***"**

where:

L is a length of a cube side

origin is a position of the cube in a scene where commas separate the coordinates with or without white spaces

filepath is a full path to an output STL file including its name

The result:

A tessellated cube is written to an ASCII STL file.

The return values:

One of these values depending on a case described below:

0 if succeeded

1 if **L** <= 0

2 if the **filepath** is incorrect

3 if one or all arguments are not specified

An example:

Cube L = *10.0***, origin = (***4.5***,***3.4***,***2.1***), filepath = "***D:\cube.stl***"**

A tessellated cube is created with the side length 10.0 at the point (4.5,3.4,2.1), and is written to an ASCII STL file with the full path *D:\cube.stl*.

Sphere

The syntax:

Sphere R = *dbl_value***, origin = (***dbl_value***,** *dbl_value***,** *dbl_value***), filepath = "***string_value***"**

where:

R is a sphere radius

origin is a position of the sphere in a 3D scene where commas with or without white spaces separate the coordinates

filepath is a full path to an output STL file including its name

The result:

A tessellated sphere is written to an ASCII STL file.

The return value:

One of these values depending on a case described:

0 if succeeded

1 if **R** <= 0

2 if the **filepath** is incorrect

3 if one or all arguments are not specified

An example:

Sphere R = *10.0***, origin = (***4.5***,***3.4***,***2.1***),** **filename = "***D:\sphere.stl***"**

A tessellated sphere is created with a radius of 10.0 at the point (4.5,3.4,2.1), and is written to an ASCII STL file with the full path *D:\sphere.stl*.

Split

Syntax:

Split input = "*string_value***", origin = (***dbl_value***,** *dbl_value***,** *dbl_value***), direction = (***dbl_value***,** *dbl_value***,** *dbl_value***), output1 = "***string_value***", output2 = "***string_value***"**

where:

input is a full path to an input STL file, including its name, containing an original tessellated mesh

origin is a position of a normal vector start in a 3D scene where commas separate the coordinates with or without white spaces

direction is a position of the normal vector direction in the 3D scene where commas separate the coordinates with or without white spaces

output1 and **output2** are full paths to output STL files including their names

The result:

Two tessellated meshes are written to the **output1** and **output2** STL ASCII files as the output of splitting the original mesh contained in the **input** file by the plane defined by a normal vector from the **origin** in the **direction**.

The return value:

One of these values depending on a case described:

0 if succeeded

1 if a normal length <= 0

2 if **input**, **output1**, or **output2** is incorrect

3 if one or all arguments are not specified

4 if the plane doesn't intersect the original mesh which becomes the result

An example:

Split input = "*D:\Bunny.STL***", origin = (***1,2,3***), direction = (***0,0,1***), output1 = "***D:\A.STL***", output2 = "***D:\B.STL***"**

Two meshes are written to the *D:\A.STL* and *D:\B.STL* STL ASCII files as the output of splitting the original mesh contained in the *D:\Bunny.STL* file by the plane defined by a normal from **(***0,0,1***)** in the **(***0,0,1***)** direction.

Note: Result meshes must be enclosed and tessellated as well. For the tessellation of cut faces you may want to use monotone polygon triangulation.

Prerequisites

Before you begin to work with this Lab, it is recommended to study the following material:

- The STL File Format

Infrastructure

To set the infrastructure in your solution:

1. In Visual Studio 2015, create MeshEditor project written in C++17 as a command-line program with *main.cpp* and empty `main` function.

Table 1 shows the final directory structure of this Lab.

Table 1. Final directory structure

.	MeshEditor
MeshEditor *MeshEditor.sln*	*Application.h* *Application.cpp* *Command.h* *Parser.h* *Sphere.h* *Sphere.cpp* *Cube.h* *Cube.cpp* *Split.h* *Split.cpp* *STLParser.h* *STLParser.cpp* *main.cpp* *MeshEditor.vcxproj*

Dependencies

There are no dependencies.

Task 1

Before implementing our classes, let's examine the `main` function (see List. 1) where we create our application and register all commands. The application is expandable for new functionality added by

registering custom types of commands. Each command takes input parameters and outputs an ASCII STL file.

```
#include "Application.h"

#include "Sphere.h"
#include "Cube.h"
#include "Split.h"

int main(int argc, char *argv[])
{
    Application app;

    app.registerCommand(std::make_unique<Sphere>());
    app.registerCommand(std::make_unique<Cube>());
    app.registerCommand(std::make_unique<Split>());

    return app.execute(argc, argv);
}
```

Code Listing 1. The `main` function

Start with the implementation of the `Application` class with parsing arguments (see List. 2).

Application.h | MeshEditor

```
#include "Command.h"
#include "Parser.h"

class Application
{
public:
    void registerCommand(std::unique_ptr<Command> command);
    int execute(int argc, char *argv[]);
private:
    // TODO
};
```

Code Listing 2. The `Application` class

There are two main methods:

1. `registerCommand` stores all commands inside the application to enable future extension with new commands.
2. `execute` calls appropriate commands depending on the C arguments received from the `main` function. Also, it converts the arguments from `argc`, `argv` to a dictionary understandable to each command.

To store the commands for calling later, you must choose a container suiting best your implementation.

Now, let's look at the Code Listing 3 which shows the command interface.

Command.h	MeshEditor

```
#include <string>
#include <map>

class Command
{
public:
    virtual ~Command() {}
    virtual const std::string& getName() const = 0;
    virtual int execute(const std::map<std::string, std::string>& args) = 0;
};
```

Code Listing 3. The `Command` interface

Each command has:

1. A name returned by `getName`.
2. The `execute` method which input arguments are a dictionary of {*argument name, value*} string pairs.

   For example, the string **Sphere R = 10.0, origin = (4.5,3.4,2.1), filename = "D:\sphere.stl"** becomes the `args` argument as the dictionary of string pairs:

```
std::map<std::string, std::string> args = {
      {"R", "10.0"},
      {"origin", "(4.5,3.4,2.1)"}
      {"filename", "D:\sphere.stl"}
};
```

Task 2

Using the normal calculation by hand inside of the write function, implement the `STLParser` class needed for reading and writing STL files.

You can test the results, the output STL files, with Mixed Reality Viewer in Windows 10.

STLParser.h	MeshEditor

```
#include <vector>
#include <string>

struct Vec { double x, y, z; };

struct Vertex
{
    Vec pos;
    Vec normal;
};

using TriangleSoup = std::vector<Vertex>;
```

```
class STLParser
{
public:
    TriangleSoup read(const std::string& filename);
    void write(const TriangleSoup& triangleSoup, const std::string& filename);
private:
    // TODO
};
```

Code Listing 4. STLParser implementation

Complying with the `Command` interface, implement the three classes:

1. `Sphere` (see List. 5)

Sphere.h | MeshEditor

```
#include "Command.h"

class Sphere : public Command
{
public:
    const std::string& getName() const override;
    int execute(const std::map<std::string, std::string>& args) override;
private:
    // TODO
};
```

Code Listing 5. The Sphere class

2. `Cube` (see List. 6)

Cube.h | MeshEditor

```
#include "Command.h"

class Cube : public Command
{
public:
    const std::string& getName() const override;
    int execute(const std::map<std::string, std::string>& args) override;
private:
    // TODO
};
```

Code Listing 6. The Box class

3. `Split` (see List. 7)

Split.h | MeshEditor

```
#include "Command.h"

class Split : public Command
```

```
{
public:
    const std::string& getName() const override;
    int execute(const std::map<std::string, std::string>& args) override;
private:
    // TODO
};
```

Code Listing 7. The Split class

Questions

1. What is a scalar product? Name the properties and use of a scalar product?
2. What is a vector product? Name the properties and use of a vector product?
3. How to calculate a normal of a triangle. How to normalize a vector?
4. Write a plane equation from three points.
5. Distance from a point to a plane.
6. The algorithm of the intersection of a triangle with the plane.

Resources and Notes

1. The STL file format:
   https://en.wikipedia.org/wiki/STL_(file_format)
2. Monotone Polygon Triangulation:
   https://en.wikipedia.org/wiki/Polygon_triangulation

Lab 1

Computer Graphics Course

Overview

In this lab class, you will learn how to create a window, render geometry, and process inputs.

Objective

The objective of this Lab is to implement the following classes and functions:

- `GLWindow` class for creating a window
- `GLRenderSystem` class for rendering triangles
- `renderScene` function for drawing a simple cube
- `moveCube` function for translating this cube in the scene with keyboard arrows

Note: GLWindow and GLRenderSystem classes will be extended and used in the further labs. They will be placed in a separate library in the next labs. In this lab, they serve as simple OOP wrappers over glfw C functions.

Prerequisites

Before you begin to work with this Lab, it is recommended to study these materials:

- GLFW documentation
- OpenGL basics

Infrastructure

To set the infrastructure in your solution:

1. Create a MeshEditor project as a command-line program with *main.cpp* and empty `main` function.
2. Create a ThirdParty folder and put GLM and GLFW libraries there.

Table 1 shows the final directory structure of this Lab.

Table 1. Final directory structure

.	MeshEditor	ThirdParty
MeshEditor ThridParty *MeshEditor.sln*	*GLRenderSystem.h* *GLRenderSystem.cpp* *GLWindow.h* *GLWindow.cpp* *glad.h* *glad.c* *khrplatform.h* *main.cpp* *MeshEditor.vcxproj*	glm glfw

Dependencies

GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

GLFW

This is a cross-platform library to create windows and simplify work with OpenGL.

Glad

If an OpenGL function is not a part of the original OpenGL ABI (Application Binary Interface), it must be dynamically loaded at runtime regardless of its type.

There are two types of OpenGL functions:

1. Core functions. They are in the OpenGL specification.
2. Extensions. Their availability depends on an addition to an OpenGL version.

The same loading mechanism applies to both types.

Glad is a multi-language GL/GLES/EGL/GLX/WGL loader-generator based on the official specifications, that can get OpenGL functions from the *OpenGL32.dll* library, so you don't need to get them manually. To get these functions using the utility, follow these steps:

1. Download Glad from https://github.com/Dav1dde/glad.
2. Generate three files: *glad.h*, *glad.c*, *khrplatform.h.*
3. Add these files to the MeshEditor project, do not put them into the ThirdParty directory because they serve only to get pointers to the OpenGL functions conveniently.
4. After window creation and context setting, call the `gladLoadGLLoader` function.

Task 1

Before creating `GLWindow` class, let's look at the Code Listing 1 that demonstrates how to create and show a window programmatically. After its creation, `GLWindow` instance exists in the infinite loop until Esc is pressed. The command line displays the keys pressed.

```
#include <glfw\glfw3.h>

#include "GLWindow.h"

void onKeyCallback(KeyCode key, Action action, Modifier mods)
{
	std::cout << "Key: " << glfwGetKeyName(key, 0) << " is pressed." << std::endl;
}

int main()
{
	glfwInit();
	GLWindow window("myWindow", 640, 480);

	window.setKeyCallback(onKeyCallback);
```

```
	while (glfwWindowShouldClose(window.getGLFWHandle())
	{
		glfwSwapBuffers(window.getGLFWHandle());
		glfwWaitEvents();
	}

	glfwTerminate();
	return 0;
}
```

Code Listing 1. `GLWindow` instance

Another important component of a window is Input that requires 4 callbacks for its processing: `setKeyCallback`, `setCursorPosCallback`, `setMouseCallback`, and `setScrollCallback`, which are implemented as `std::function` (see List. 2).

GLWindow.h | MeshEditor

```
#include "glad.h"

#include <glfw\glfw3.h>

#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 GLWindow
{
```

```
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)>;

	Window(const std::string& title, uint32_t width, uint32_t height);
	~Window();
	uint32_t getWidth() const;
	uint32_t getHeight() const;
	void setKeyCallback(const KeyCallback& callback);
	void setCursorPosCallback(const CursorPosCallback& callback);
	void setMouseCallback(const MouseCallback& callback);
	void setScrollCallback(const ScrollCallback& callback);

	GLFWwindow* getGLFWHandle() const;
private:
	// TODO
};
```

Code Listing 2. Input callbacks

Code Listing 3 shows how to create a window using `glfwCreateWindow`, which returns a `glfw handle`. The subsequent line is very important because it calls `glfwMakeContextCurrent` to set the current *context* for the window. *Context* is OpenGL data that is bound to a window and used to display the geometry in the window.

Only after window creation and setting the context, `gladLoadGLLoader` should load OpenGL functions from *OpenGL32.dll*. The static variable `initGLAD` prevents `gladLoadGLLoader` from a useless second call.

Finally, glfwSetWindowUserPointer associates each glfw window with a pointer to GLWindow. This is necessary to get the GLWindow via glfw handle in the callbacks and to call the required std::function.

Note: please note that you can use friend functions to complete this task. Don't forget that you can change the TODO section in the private section the way you want.

```
GLWindow(const std::string& title, uint32_t width, uint32_t height)
// implementation details
{
	handle = glfwCreateWindow(width, height, title.data(), nullptr, nullptr);
	glfwMakeContextCurrent(handle);

	static bool initGLAD = false;
	if (!initGLAD)
	{
		initGLAD = true;
		gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);
	}

	glfwSetWindowUserPointer(handle, this);
```

```
        glfwSetKeyCallback(handle, /* implementation details */);
        glfwSetMouseButtonCallback(handle,  /* implementation details */);
        glfwSetCursorPosCallback(handle,  /* implementation details */);
        glfwSetScrollCallback(handle,  /* implementation details */);
}
```

Code Listing 3. Window creation using `glfwCreateWindow`

At the end of this task, you must have a GLWindow class, using which a user can create windows. Make sure that GLWindow runs fine on Code Listing 1. The next task is to implement rendering of geometric primitives.

Task 2

To complete this task, you need to implement the `GLRenderSystem` class (see List. 4), able to:

- Draw a list of triangles with `renderTriangleSoup`. *A triangle soup* is a group of unorganized triangles, generally with no relationship at all.
- Set a transformation matrix, view and projection matrices.
- Set the light source.

GLRenderSystem.h | MeshEditor

```
#include <glm/glm.hpp>

#include <glfw/glfw3.h>

struct Vertex
{
        glm::vec3 position;
        glm::vec3 normal;
};

class GLRenderSystem
{
public:
        void init(); //must be called after glfw window creation. Set default GL
settings

        void clearDisplay(float r, float g, float b);
        void setViewport(double x, double y, double width, double height);
        void renderTriangleSoup(const std::vector<Vertex>& vertices);
        void setupLight(uint32_t index, glm::vec3 position, glm::vec3 Ia, glm::vec3 Id,
glm::vec3 Is);
        void turnLight(uint32_t index, bool enable);

        void setWorldMatrix(const glm::mat4& matrix);
        const glm::mat4& getWorldMatrix() const;

        void setViewMatrix(const glm::mat4& matrix);
        const glm::mat4& getViewMatrix() const;

        void setProjMatrix(const glm::mat4& matrix);
        const glm::mat4& getProjMatrix() const;
```

```
private:
	// TODO
};
```

Code Listing 4. `GLRenderSystem` class

OpenGL uses triangles to display geometry because mathematical calculations are the most effective on a triangle with 3 vertices. A bunch of triangles may represent any complex surface (see Fig. 1).

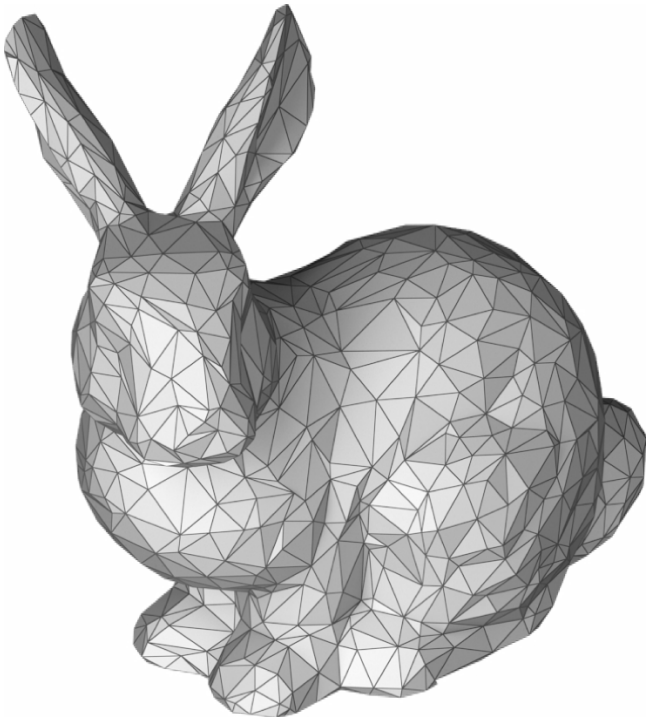


Figure 1. Representation with triangles

Let's start with `Vertex` declaration. It has two basic attributes:

1. `position` – to define the location of a vertex in space.
2. `normal` – to calculate the lighting.

For simplicity, we do not specify other attributes, such as texture coordinates.

Flat Shading vs. Smooth Shading, and Vertex `Normal` Attribute

Unfortunately, the triangle meshes cannot represent smooth surfaces perfectly, unless the triangles are very small. *Flat shading model* uses only the `normal` attribute of a triangle to calculate the lighting. Therefore, it produces the faceted look of an object (see Fig. 2).

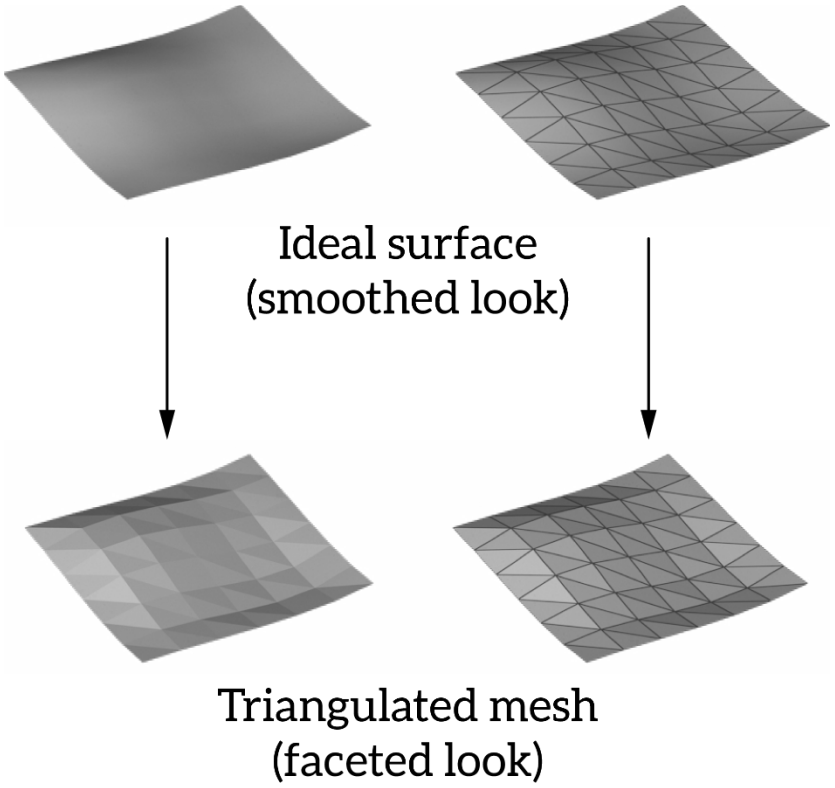


Figure 2. Flat shading model

Gouraud shading solves this problem with continuous shading across the surface of a triangular mesh. The `normal` attribute is specified for each vertex, not for a triangle. Linear interpolation between the `normal` attributes of vertices produces the *normal* for the color calculation of a point on a surface.

By default, in OpenGL the smooth shading and the depth test are disabled. To enable them, you need to call `GLRenderSystem::init` function after window creation (see List. 5).

```
void GLRenderSystem::init()
{
	glShadeModel(GL_SMOOTH);
	glEnable(GL_DEPTH_TEST);
	glDepthFunc(GL_LEQUAL);
}
```

Code Listing 5. GLRenderSystem::init function call

Phong Lighting

The OpenGL API provides the following functions to configure lighting:

1. To enable/disable up to 8 individual light sources.
2. For each light, to break its emitted intensity into 3 components: ambient I_a, diffuse I_d, and specular I_s. Each type of light component consists of 3 color elements, so, for example, I_{rd} denotes the intensity of the red component of diffuse illumination.
3. It's possible to specify several different kinds of light sources, including:

a. A distant point light source at infinity, defined by a unit vector from a surface to a light source in 3D space. This is a Light (*L*) vector that points in the same direction for all vertices and with constant power.
b. A directional light source that you must aim at with a direction vector.
c. A local point light source, defined by a point positioned in 3D space. The *L* vector to this light source from a vertex varies with vertex position. You can specify the *L* vector with distance: no light attenuation, attenuation by $\frac{1}{r}$, attenuation by $\frac{1}{r^2}$, or the weighted sum.

Phong Reflection Model

The *Phong reflection model* specifies reflectance at each vertex, where each component of the RGB color model has K_e, K_a, K_d, K_s values plus a single shininess exponent integer S_e. Equation (1.1) describes the on-screen color with these values.

$$color = K_e + I_a \cdot K_a + I_d \cdot K_d \cdot \max(0, \mathrm{N} \cdot \mathrm{L}) \cdot Att + I_s \cdot K_s \cdot Att \cdot \max(0, \mathrm{R} \cdot \mathrm{V})^{S_e} \quad (1.1)$$

where

$\boldsymbol{K_e}$ — light emitted by the material, unaffected by the illumination

$\boldsymbol{I_a}$ — ambient illumination of light, effects of multiple lights are additive

$\boldsymbol{K_a}$ — ambient reflectance of the material

$\boldsymbol{I_d}$ — diffuse illumination of light

$\boldsymbol{K_d}$ — diffuse reflectance of the material

N — the unit normal vector for the material surface, see `Vertex::normal`

L — the unit vector from the surface to the light

The **N·L** produces the cosine falloff for the surfaces lit from a side, and the *max(0, N·L)* prevents from negative results if the light moves behind the object

Att — the attenuation factor for the distance from the light, if any

$\boldsymbol{I_s}$ — specular illumination of light

$\boldsymbol{K_s}$ — specular reflectance of the material

R — the unit vector from the surface pointing in the mirror-reflection direction computed from the surface normal N and light direction L

V — the unit vector from the surface to the camera (eye) position

$\boldsymbol{S_e}$ — shininess exponent integer to set the sharpness of the specular reflection.

Now, when we know how to display triangles and configure lighting, let's discuss how to move 3D objects in space.

Transformations

To create a computer-graphics display of an object, you should consider its geometric characteristics such as size, shape, location, orientation, and its spatial relationship to other objects nearby. If you need to describe, measure, and analyze these characteristics, you must place the object within some frame of reference — a coordinate system. The computer-graphics displays require a variety of two- and three-dimensional coordinate systems, each identified by name and a unique set of coordinate axes:

1. World Coordinate System (WCS)
2. Local Coordinate System (LCS)
3. View Coordinate System (VCS)
4. Homogeneous Coordinate System (HCS)

Let's have a closer look at each.

World Coordinate System

This is the primary three-dimensional frame of reference in which you define and locate in space all the geometric objects in a scene. You also locate the position of an observer and the viewing direction in the global coordinate system.

Local Coordinate System

The local coordinate system defines an object regardless of the WCS without specifying its location or orientation in the WCS (see Fig. 3).

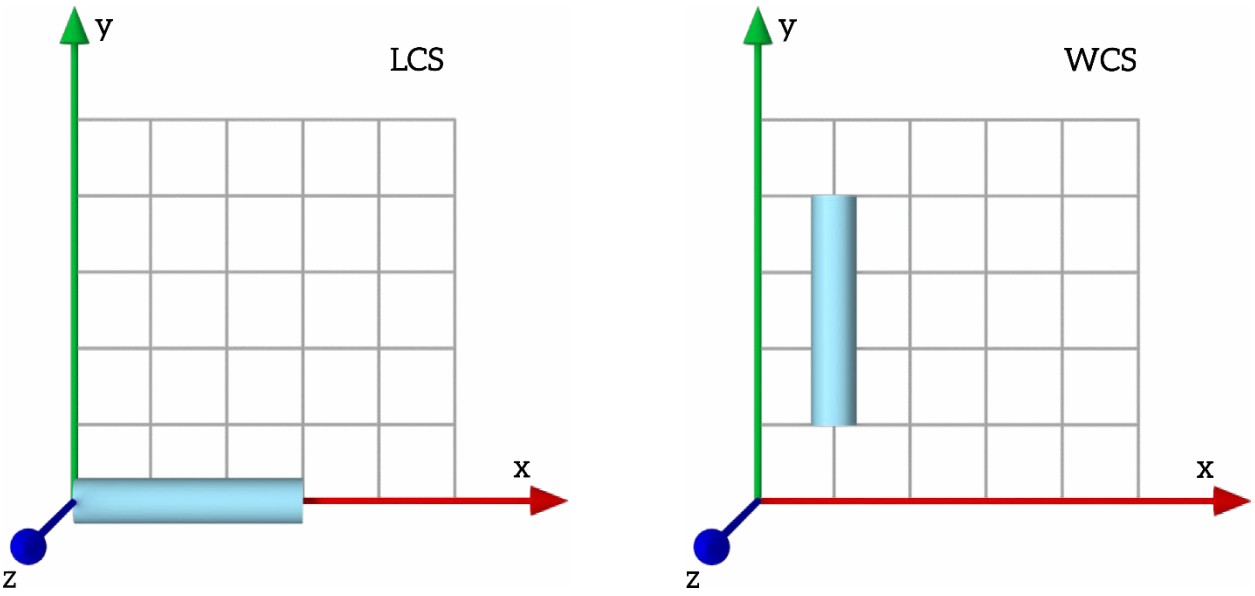


Figure 3. LCS and WCS

Matrix transformations are used to locate an object from LCS to WCS. For example, let's take a cylinder that in the LCS looks in the direction (1,0,0), with the bottom point (0,0,0), and the top point (3,0,0). And locate it in the WCS so that it looks in the direction (0,1,0) and located in (1,0,0) point. Table 2 shows matrix transformations needed to complete this task.

Table 2. LCS to WCS matrix transformations

LCS to WCS matrix	Bottom point in WCS	Top point in WCS
$\begin{pmatrix} \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$	$\begin{pmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 0 \\ 1 \end{pmatrix}$

With these transformations, all the cylinder vertices are moved from LCS to WCS and set for each vertex a relative to the center of the world. Table 3 shows the inverse transformation that converts points from WCS to LCS.

Table 3. WCS to LCS matrix transformations

WCS to LCS matrix	Bottom point in LCS	Top point in LCS
$\begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{-1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Below there are some useful transformations for the further Labs:

- Four-dimensional transformations (1.2) and (1.3), which are a compact way to represent the transformations above.

$$F = \begin{pmatrix} M & T \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{M_{xx}} & \boldsymbol{M_{xy}} & \boldsymbol{M_{xz}} & \boldsymbol{T_x} \\ \boldsymbol{M_{yx}} & \boldsymbol{M_{yy}} & \boldsymbol{M_{yz}} & \boldsymbol{T_y} \\ \boldsymbol{M_{zx}} & \boldsymbol{M_{zy}} & \boldsymbol{M_{zz}} & \boldsymbol{T_z} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix} \tag{1.2}$$

$$F^{-1} = \begin{pmatrix} M^{-1} & -M^{-1}T \\ 0 & 1 \end{pmatrix} \tag{1.3}$$

- Rotation matrix from axis and angle:

```
glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f), angle, axis);
```

  Rotation matrix from two vectors and normal:

```
double sign = glm::dot(normal, glm::cross(a, b)) > 0 ? 1 : -1;
double angle = sign * angle(a, b);
glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f), angle, normal);
```

- To rotate an object around its own axis, follow the steps below (see Fig. 4):
  1. Translate the object to its origin.
  2. Rotate the object.
  3. Translate it back.

  In code, it looks like this: `translation(T)*rotation(N, angle)*translation(-T).`

  Note: Transformations are often done around the origin.

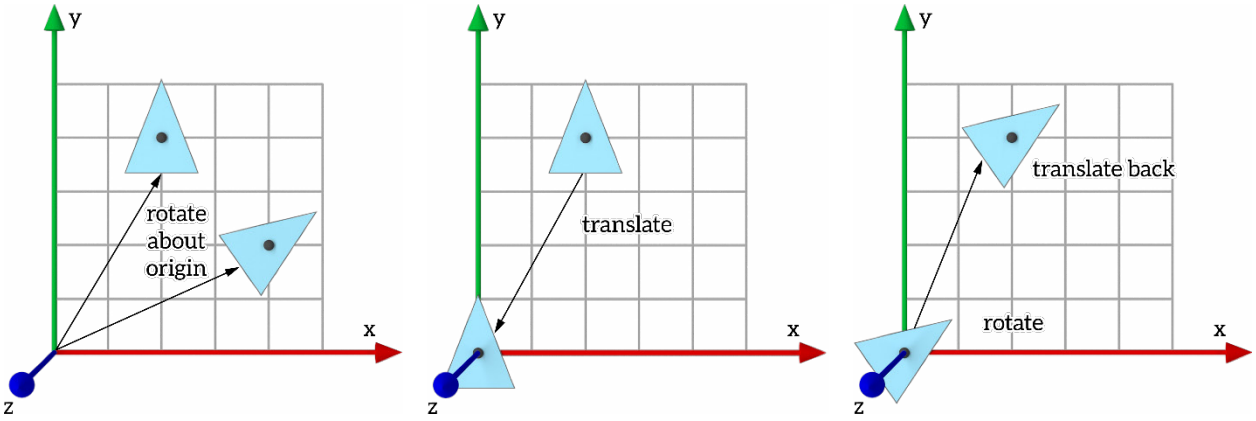


Figure 4. Object rotation procedure

View Coordinate System

In the VCS, objects are relative to the observer to simplify the mathematics of their image projection onto the picture plane when we look along one of the principal axes (see Fig. 5). In Open GL, this axis is Z.

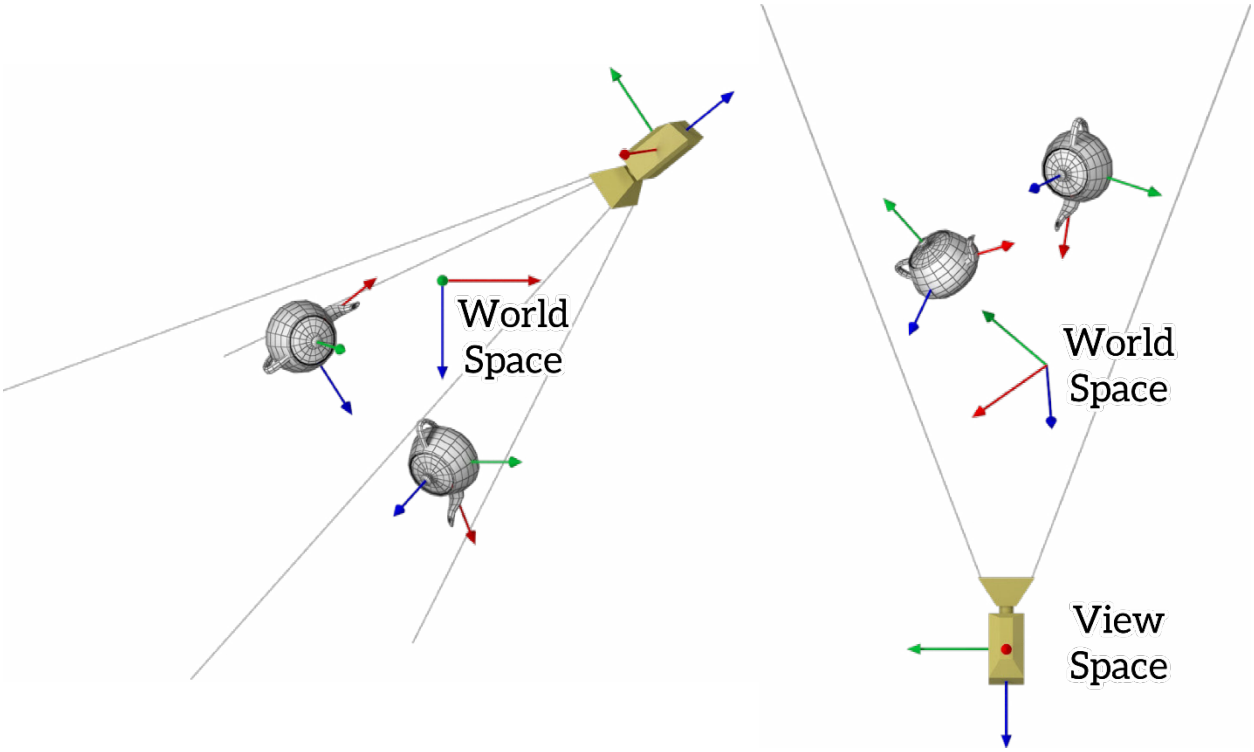


Figure 5. WCS and VCS

In Lab 2, we will discuss the object transformation to the VCS, particularly to the *camera, or eye space*. Here the camera and eye are interchangeable words.

In the VCS, the distance from an object to a camera plays the key role. For two vertices with the same x and y values, the vertex with the biggest z value is closer on a screen than the other. This is called *perspective projection*.

Homogeneous Coordinate System

The camera can see only a part of the scene. This visible part is called *view frustum*, and it usually has the shape of a sliced pyramid (see Fig. 6).

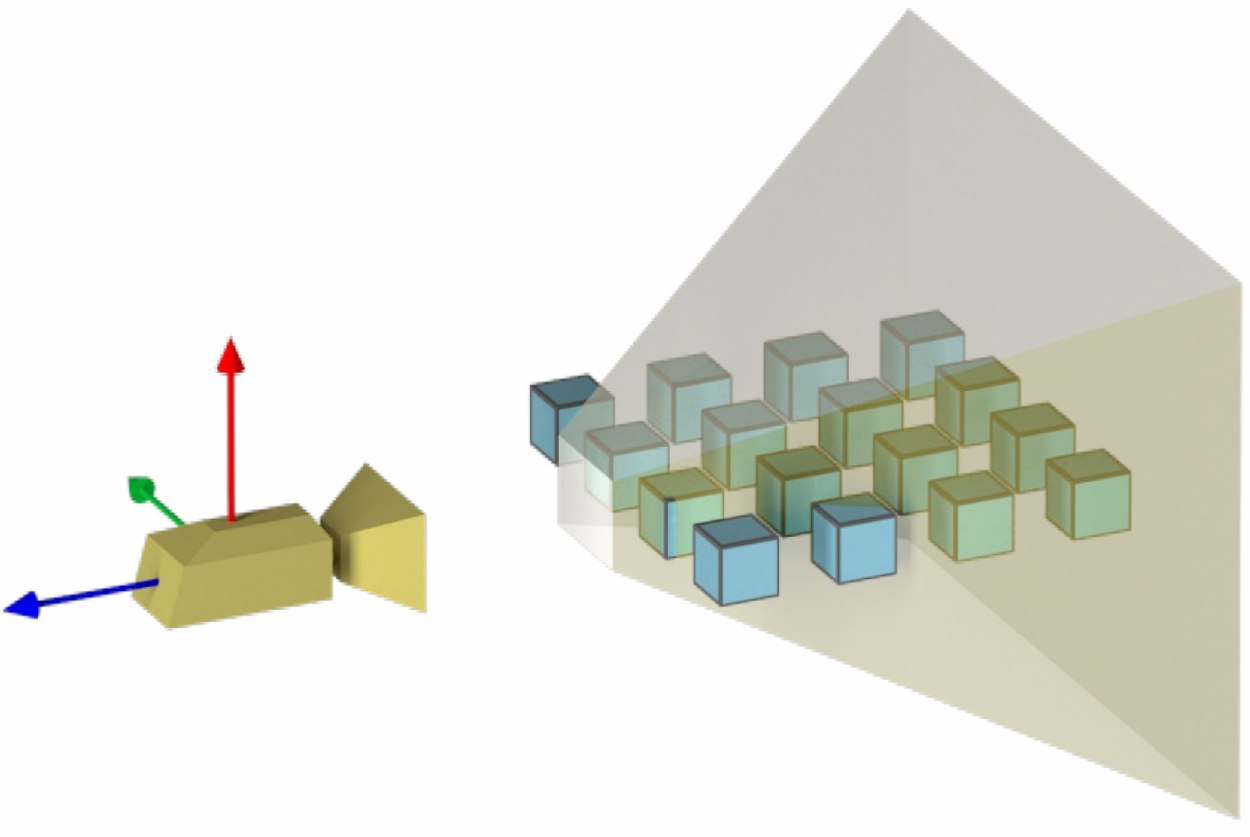


Figure 6. View frustum

Open GL can show on a screen only the objects in the HCS, where the view frustum has the shape of a perfect cube aligned with the principal axis.

To transform the scene to HCS, you must multiply everything by the *projection matrix*. Figure 7 shows the result of this transformation. The view frustum is now a perfect cube, and all blue objects are deformed. Thus, the objects that are near the camera are bigger, and the others are smaller, like in real life. In Lab 2 will discuss this transformation in detail.

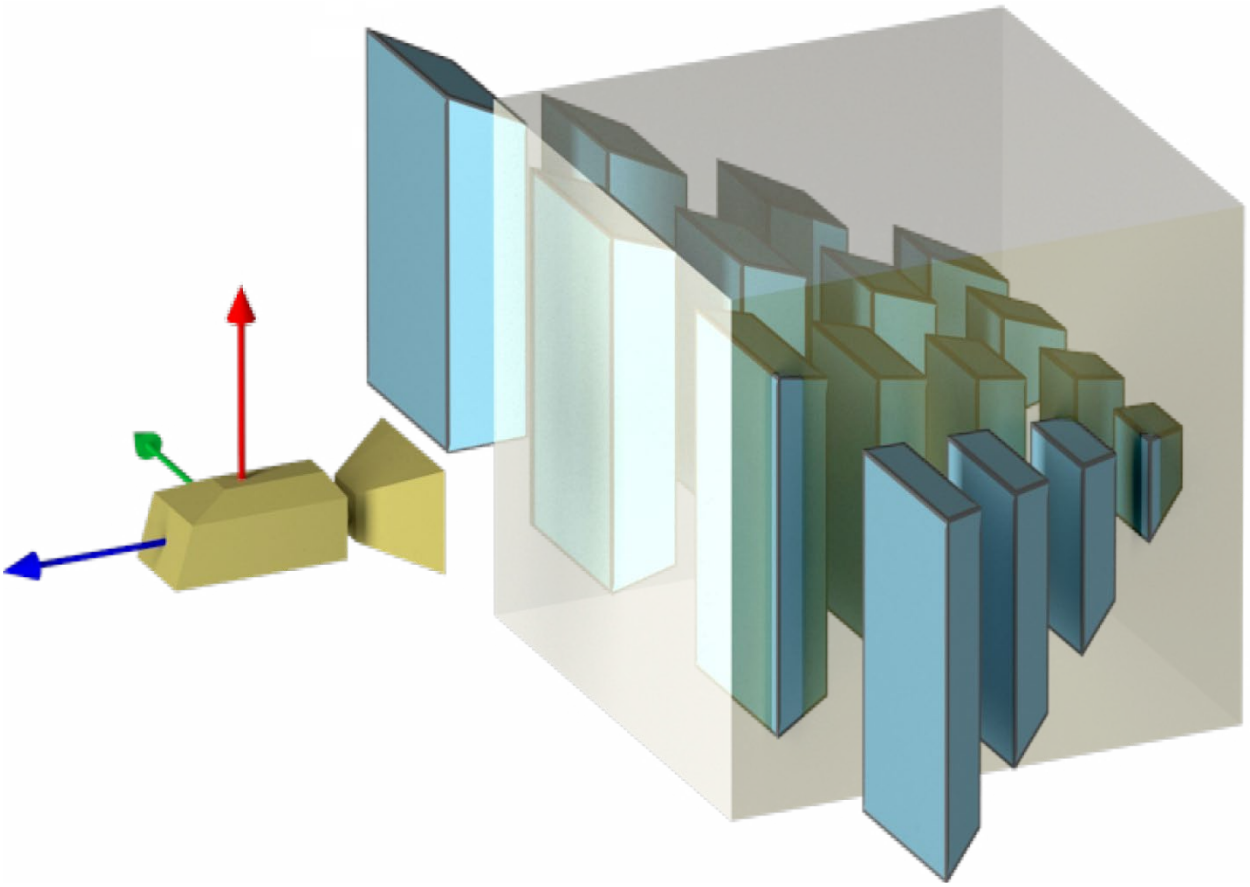


Figure 7. Homogeneous Coordinate System

Figure 8 shows the rendered image behind the view frustum in the HCS.

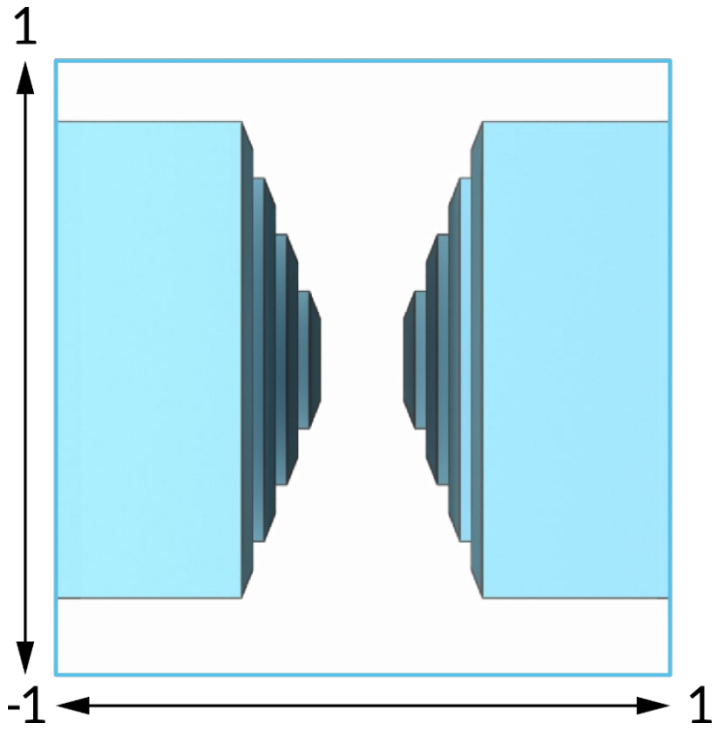


Figure 8. Rendered image in the HCS

This square image doesn't fit the actual window size. To solve this problem, Open GL automatically applies another mathematical transformation (see Fig. 9).

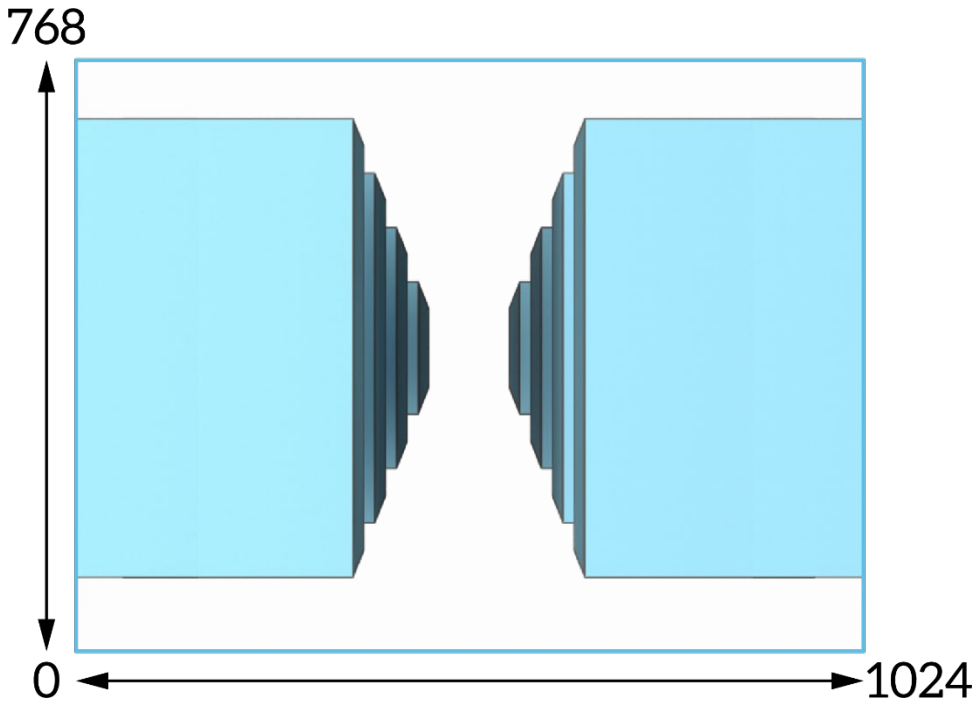


Figure 9. Final image

Task 3

To complete this task, you need to implement the `renderScene` and `moveCube` functions to draw and to move a cube using `GLRenderSystem` (see List. 6).

Note: Please note that `glVertex` is enough to implement the `renderScene` function. Using VBO is acceptable, but it requires additional interface functions in `GLRenderSystem`, which are not part of this lab series. Make sure to use `GLWindow` and `GLRenderSystem` classes from the previous task.

main.cpp | MeshEditor

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>

#include "GLWindow.h"
#include "GLRenderSystem.h"

void renderScene(GLRenderSystem& rs)
{
	// TODO
}

void moveCube(GLRenderSystem& rs, v3 offset)
{
	// TODO
}

void onKeyCallback(KeyCode key, Action action, Modifier mods)
{
	if (key == KeyCode::UP)
		moveCube(rs, /* TODO */);

	if (key == KeyCode::DOWN)
		moveCube(rs, /* TODO */););

	if (key == KeyCode::LEFT)
		moveCube(rs, /* TODO */););

	if (key == KeyCode::RIGHT)
		moveCube(rs, /* TODO */););
}

int main()
{
	GLRenderSystem rs;
	GLWindow window("myWindow", 640, 480);

	window.setKeyCallback(onKeyCallback);

	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::lookAt(glm::vec3(0.0f, 0.0f, -10.0f),
        glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        rs.setViewMatrix(viewMatrix);

        glm::mat4 projMatrix = glm::perspective(glm::radians(60.0f), 640.0f / 480.0f,
        0.1f, 500.f);
        rs.setProjectionMatrix(projMatrix);

        while (glfwWindowShouldClose(window.getGLFWHandle())
        {
                rs.setViewport(0, 0, window.getWidth(), window.getHeight());
                rs.clearDisplay(0.5f, 0.5f, 0.5f);
                renderScene(rs);
                glfwSwapBuffers(window.getGLFWHandle());
                glfwWaitEvents();
        }

        return 0;
}
```

Code Listing 6. `renderScene` and `moveCube` functions

There are two new variables in the Code Listing 6:

- `viewMatrix` sets the view matrix to transform the WCS into the VCS
- `projMatrix` sets the projection matrix to transform the VCS into the HCS

Currently, it is enough to know their meanings. We will discuss their construction in Lab 2.

The result of this Lab is the application able to display and move a cube.

Exercises

1. Use different transformation matrices in `setWorldMatrix`, learn how to use a transformation matrix to place a 3D object in the scene, then create a wall of cubes.
2. Add functions to `GLRenderSystem` to implement directional and point lights.
3. Add a function to `GLRenderSystem` to configure a material with Phong material parameters.

Questions

1. What is a transformation matrix? How does the usage of the transformation matrix to a vector and a point differ? Where are the coefficients in charge of the translation in the transformation matrix? Where are the coefficients in charge of rotation and scaling? Why do we need the transpose operation? How to invert an orthonormal matrix quickly? For the set matrix (the examiner gives a matrix) find a translation vector, basis, and scaling coefficients.
2. Using the given transformation matrix and a normal, calculate the transformed normal. Do the same for the point.

Resources and Notes

1. GLFW library:
   http://www.glfw.org/
2. OpenGL course:
   http://nehe.gamedev.net/tutorial/lessons_01__05/22004/
3. GLM library:
   https://glm.g-truc.net
4. Phong reflection model:
   https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model
5. Rotation matrix from axis and angle:
   https://en.wikipedia.org/wiki/Rotation_matrix#Rotation_matrix_from_axis_and_angle -

Lab 2

Computer Graphics Course

Overview

In this Lab, you will learn how to create a camera, a viewport, and to move/rotate the camera with a keyboard.

Objective

In Lab 1, we hard-coded two functions for cubes to be visible on a screen:

- `GLRenderSystem::setModelView` to set the model view matrix
- `GLRenderSystem::setProjectionMatrix` to set the projection matrix

The objective of this lab is to add camera movement/rotation with a keyboard. This functionality is the job of two new functions of the new classes:

- `Camera::calcViewMatrix` that returns the parameters for `GLRenderSystem::setModelView`
- `Viewport::calcProjectionMatrix` that returns the parameters for `GLRenderSystem::setProjectionMatrix`

Infrastructure

Table 1 shows the final directory structure of this Lab. The new files are green.

Table 1. Final directory structure

.	MeshEditor	ThirdParty
MeshEditor	**Viewport.h**	glm
ThirdParty	**Viewport.cpp**	glfw
MeshEditor.sln	**Camera.h**	
	Camera.cpp	
	GLRenderSystem.h	
	GLRenderSystem.cpp	
	GLWindow.h	
	GLWindow.cpp	
	glad.h	
	glad.c	
	khrplatform.h	
	main.cpp	
	MeshEditor.vcxproj	

Task 1

OpenGL defines neither camera object nor a specific matrix explicitly for camera transformation. Instead, OpenGL transforms the entire scene inversely to so-called *Eye space*, where a fixed camera is at the origin (0,0,0) and always looks along negative Z-axis (see Fig. 1).

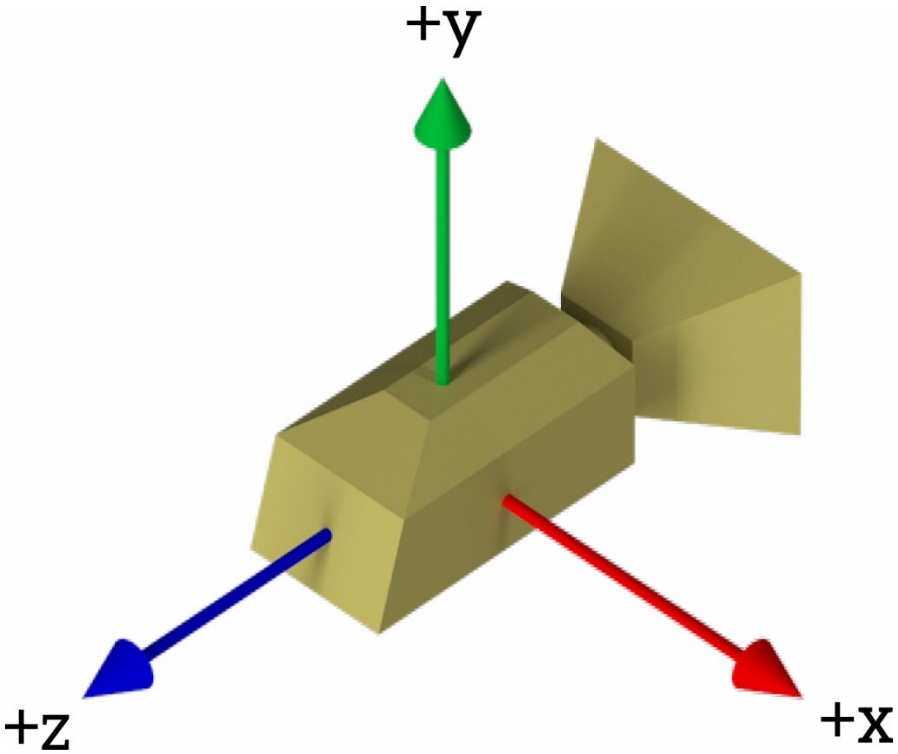


Figure 1. Eye space

OpenGL uses the same `GL_MODELVIEW` matrix for the transformations of an object to the world space and a camera to the eye space. Usually, the ModelView matrix has two sub-matrices (1.1).

$$ModelView = Model * View \quad (1.1)$$

Scene transformation with these sub-matrices has two steps:

1. The *Model* matrix of each object transforms this object in the scene.
2. The *View* matrix reversely transforms the entire scene.

The *View* matrix is the basis for our `Camera` class. You may find the discussion of its construction below. The *Model* matrix is not used in this Lab.

`gluLookAt`

The `gluLookAt` function provides the default camera configuration in OpenGL. It transforms the scene so that the camera is located at (0, 0, 0) and looks towards the negative Z-axis. The camera never moves, the scene does. This function takes the following parameters defined in the `WCS`:

- Eye position (camera location)
- Target point at which camera looks
- Up vector which defines the up direction from the camera

The `gluLookAt` uses the view matrix which consists of *scene translation* and *rotation* matrices; each is a mathematical equivalent of the corresponding action.

The formulas below express the computation of the *scene translation*, where translation (2.1) is the product of the eye position deduction from the origin and the translation matrix (2.2) computed with the inverse value of this translation.

$$t = (0,0,0) - \text{eye position} \tag{2.1}$$

$$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

According to the instructions below, the first step of the rotation is to construct three additional vectors: *direction*, *right* and *up* (see Fig. 2).

1. Compute and normalize the *direction* vector from the *eye position* to the *target position.*
2. Compute and normalize the *right* vector as a cross product between *up* and *direction* vector.
3. Adjust the *up* vector to make it orthogonal to other vectors, so all three vectors become orthonormal.

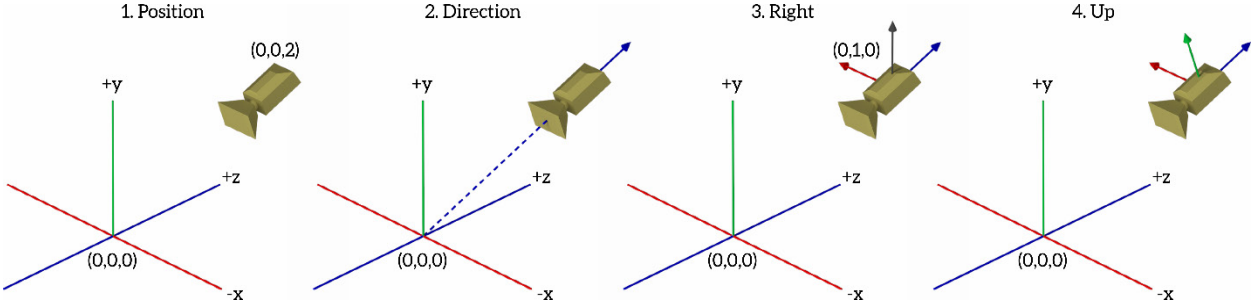


Figure 2. Camera rotation vectors

The second step of the rotation is to compute the rotation matrix with these three vectors (3.1).

$$\begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^T \tag{3.1}$$

Finally, multiplication of translation and rotation matrices produce the view matrix.

Now when we know how `gluLookAt` works under the hood, let's use it to implement the `Camera` class, and three vectors `eye`, `target` and `up` to configure this class (see List. 1).

Camera.h | MeshEditor

```
#include <glm/glm.hpp>

class Camera
{
public:
	glm::mat4 calcViewMatrix() const;
```

```
        glm::vec3 calcForward() const;
        glm::vec3 calcRight() const;
        double distanceFromEyeToTarget() const;
        const glm::vec3& getEye() const;
        const glm::vec3& getTarget() const;

        void setFrontView();
        void setTopView();
        void setRearView();
        void setRightView();
        void setLeftView();
        void setBottomView();
        void setIsoView();

        void orbit(glm::vec3 a, glm::vec3 b);
        void pan(double u, double v);
        void zoom(double factor);

        void translate(glm::vec3 delta);
        void setDistanceToTarget(double D);
        void transform(const glm::mat4& trf);
        void rotate(glm::vec3 point, glm::vec3 axis, double angle);
        void setEyeTargetUp(glm::vec3 newEye, glm::vec3 newTarget, glm::vec3 newUp);
private:
        glm::vec3 eye{ 0, 0, 1 };
        glm::vec3 target;
        glm::vec3 up{ 0, 1, 0 };
};
```

Code Listing 1. `Camera` header file

As you can see in the code snippet below, the `Camera::calcViewMatrix` function returns the view matrix, which is passed then to `GLRenderSystem::setViewMatrix`.

```
glm::mat4 Camera::calcViewMatrix() const
{
        return glm::lookAt(eye, target, up);
}
```

You can start to work on the `Camera` class with the implementation of the functions to move and rotate the `Camera`:

- `void Camera::translate(glm::vec3 delta)`

  To move the camera toward `eye` and `target` on the specified distance, it is enough to pass this distance as `delta`.

- `void Camera::setDistanceToTarget(double D)`

  This function moves the camera on the specified distance to the target point. Forward (`f`) vector is calculated as ||target – eye||

```
eye = target - f * D;
```

- `void Camera::transform(const glm::mat4& trf)`

  Multiplication of `eye`, `target` and `up` produces the transformation matrix which performs the transformation. **Keep in mind the difference between the transformation of *vector* and *point coordinates*.**

- `void Camera::rotate(glm::vec3 point, glm::vec3 axis, double angle)`

  Implementation of *rotation* around the axis (`point`, `axis`) by a specified `angle` uses the `translate` and `transform` functions. To perform this task, first move the camera to (0,0,0)-`point`, then calculate the rotation matrix from the `axis` and the `angle`, apply rotation transformation and finally move the camera back to `point`-(0,0,0).

- `void Camera::setEyeTargetUp(glm::vec3 newEye, glm::vec3 newTarget, glm::vec3 newUp)`

  It is not enough just to set new values for `eye`, `target`, and `up` because the new *direction* vector may not be perpendicular to `newUp`. This function solves this case by adjusting the `up` vector (4.1).

$$up' = rotation\left(d \times up, \frac{\pi}{2}\right) * d \tag{4.1}$$

The next piece of work on the `Camera` class is the implementation of the standard functions for configuring the view planes:

- `void Camera::setFrontView()`

```
{
	double D = distanceFromEyeToTarget();
	setEyeTargetUp(target + glm::vec3{ 0,0,1 }, target, { 0,1,0 });
	setDistanceToTarget(D);
}
```

  When setting the Front View (see Fig. 3), you must set `eye` position at (0,0,1), `target` at (0,0,0) and `up`, at (0,1,0). This way the camera will look in the direction of the negative Z-axis. To keep the distance to the target point after calculations of the `eye`, `target`, and `up`, you need to set the `D` value using the `setDistanceToTarget` function.

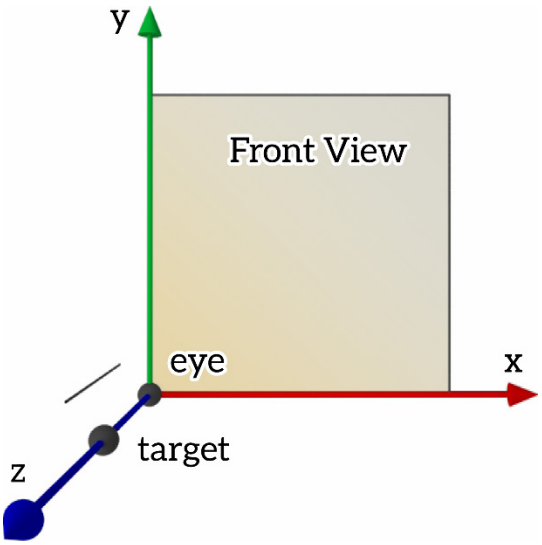


Figure 3. Setting the Front View

- ```
  void Camera::setRightView()
  {
  	glm::vec3 oldTarget = target;
  	setFrontView();
  	rotate(oldTarget, { 0,1,0 }, pi * 0.5);
  }
  ```

  To set the Right View, follow these steps: save the `target` value because the next step will change it, then set the front view, and finally rotate the camera around the old `target` value and the Y-axis by $\frac{\pi}{2}$.

- Use the two approaches mentioned above to implement the remaining View functions: Top, Rear, Bottom, Left, and Iso.

Your next task is to enable continuous motion of the camera movement with two functions `pan` and `orbit`. Thus, for each mouse event, it is necessary to:

1. Calculate the current parameters of the mouse coordinates.
2. Take the delta between the current and the old parameters.
3. Apply the operation.
4. Store the current coordinates.

```
void Camera::pan(double u, double v);
```

Pan transformation is the result of equal vector transformations of the `eye` and the `target` points, expressed in the formulas (5.1) and (5.2), where ***u*** and ***v*** are the coordinates of the panned point on the *target plane*. Respective to the current camera view, the simplest pan is to the right, left, up, or down.

$$eye' = eye + right * u + up * v \tag{5.1}$$

$$target' = target + right * u + up * v \tag{5.2}$$
```

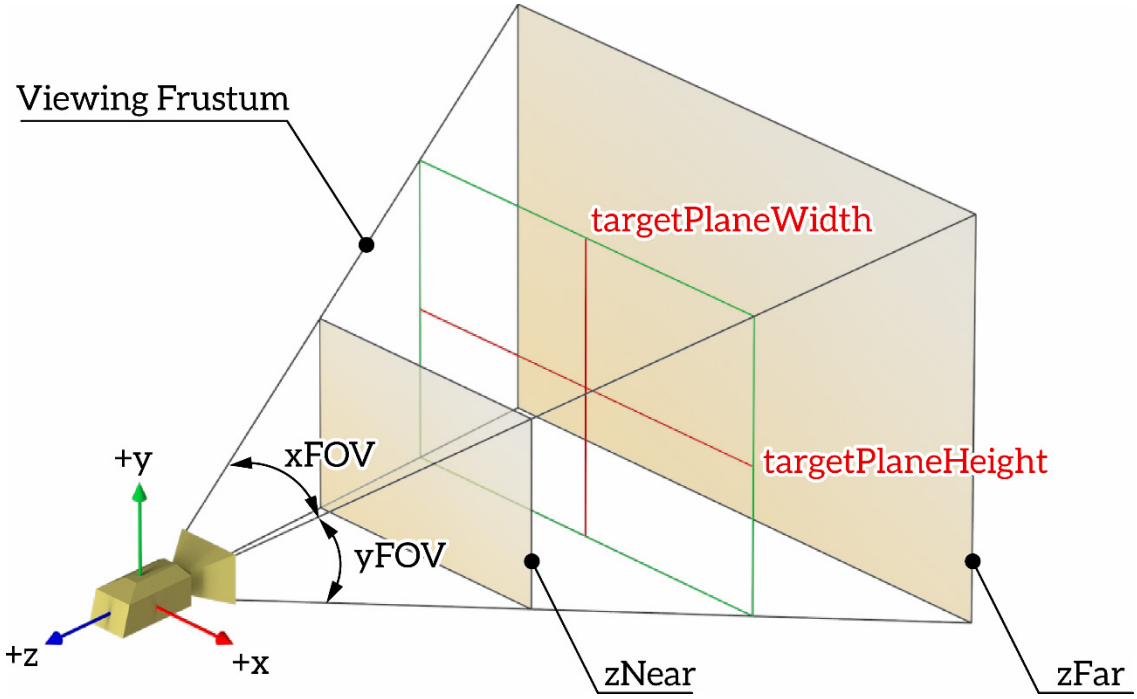


Figure 4. Pan transformation

The Viewpoint section below provides a detailed discussion of the *target plane*.

To map the *x*, *y* mouse coordinates to the *u*, *v* coordinates, it is necessary to convert the mouse point [0;0] x [width; height] ange to the point on the target plane [-targetPlaneWidth; -targetPlaneHeight] x [targetPlaneWidth; targetPlaneHeight].

The pan algorithm is described below (6.1):

1. The first click triggers `IWindow::onMouseInput` and a calculation of the original *u0*, *v0* coordinates from the original mouse point.
2. Then, a drag triggers `IWindow::onMouseMove` and a calculation of the new *u*, *v* coordinates, and delta.

$$dU = u0 - u \quad (6.1)$$
$$dV = v0 - v$$
$$pan(dU, dV)$$
$$u0 = u$$
$$v0 = v$$

```
void Camera::orbit(glm::vec3 a, glm::vec3 b);
```

Orbit camera motion, also called *Arcball*, is the movement of the `eye` position from a point to a point while the object and `target` positions stay unchanged.

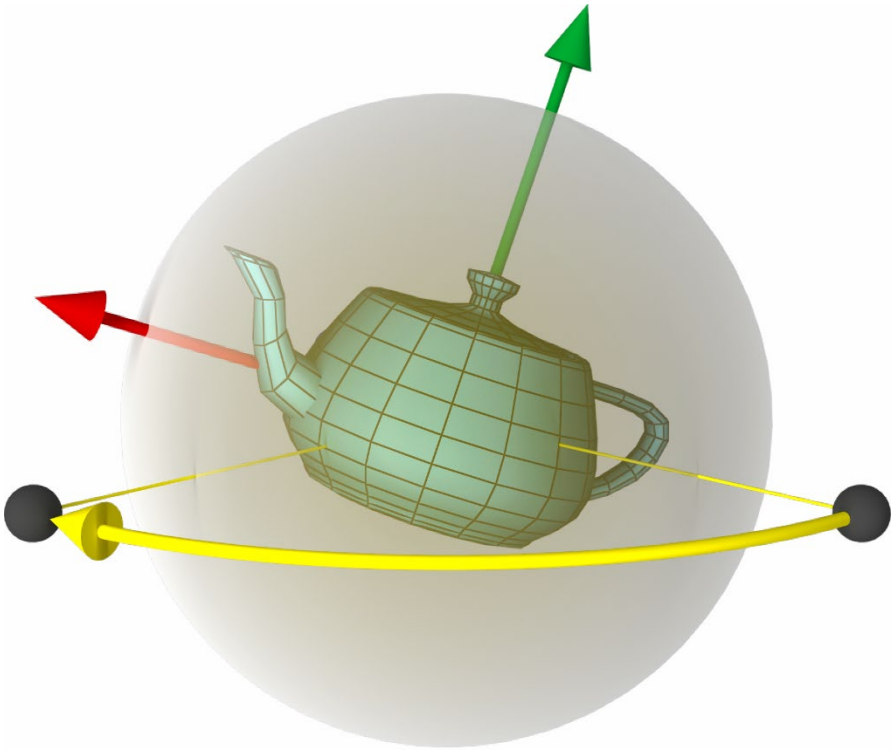


Figure 5. Orbit camera motion

Imagine, that our camera lookis at the object placed in the center of a sphere (Fig. 5). You select a point on this sphere and then another one. The angle between these two vectors produces the rotation (*orbit matrix*) around the object (`target` point).

Let's discuss the mathematical computations of the rotation

$$eye' = target + O * (eye - target) \tag{7.1}$$

$$up' = O * up \tag{7.2}$$

In our first equation (7.1) we use the *orbit matrix* to rotate the *direction vector* which we add then to the current *target* value to produce the new *eye* position.

To obtain this *orbit matrix*, you need to complete several computations:

1. Compute the angle between the *a* and *b* vectors (8.1).

$$\alpha = \cos^{-1}(a \cdot b) \tag{8.1}$$

2. Multiply the *a* and *b* vectors to produce the axis of rotation (8.2).

$$axis = b \times a \tag{8.2}$$

3. The *a* and *b* vectors are in the LCS of the sphere. To transform them to the WCS, inverse the rotation part of the `lookAt` function (8.3) and (8.4). In other words, there must be the transformation of these vectors to the Camera coordinate system in the WCS.

$$toWorldCameraSpace = transpose\left(rotationComponent\left(lookAt(eye, target, up)\right)\right) \quad (8.3)$$

$$axis' = toWorldCameraSpace * axis \quad (8.4)$$

4. Calculate the orbit matrix (8.5).

$$O = rotation(axis', \alpha) \quad (8.5)$$

To map the (*x*, *y*) mouse coordinates to a vector on the sphere, convert the mouse point *[0; 0] x [width; height]* to *[-ar;1] x [ar;-1]* range, where ***ar*** is the aspect ratio (*width/height*). This transformation prevents from setting the radius of the sphere and assures that the rotation is around the sphere only.

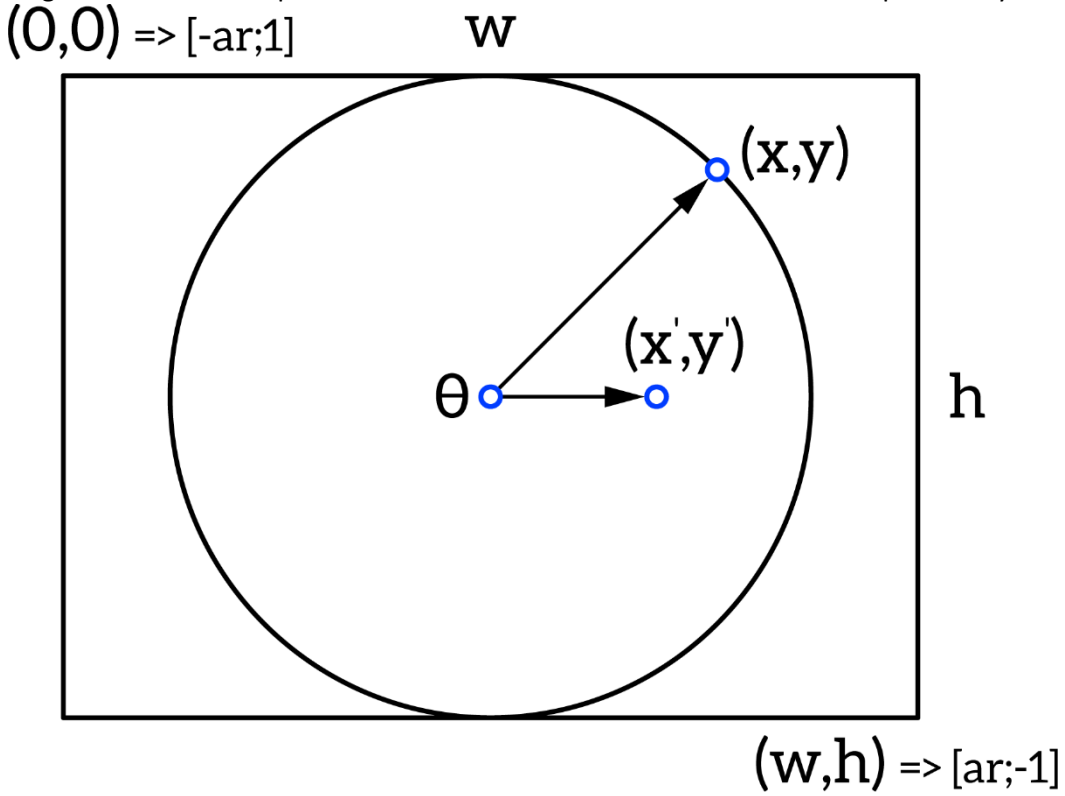


Figure 6. Mapping the mouse coordinates

The `z(x,y)` function uses the (*x*, *y*) mouse coordinates to map the inner points on a sphere and the outer ones on the z=0 plane. This function helps to rotate a 3D object when a selection is inside the sphere. Then within the screen borders, the rotation is around Z-axis only, being z=0.

```
glm::vec3 pointOnSphere{ xInTargetPlane, yInTargetPlane, 0 };

if (length(pointOnSphere) >= 1.0) // value outside sphere (R = 1), project it onto sphere
	pointOnSphere = normalize(pointOnSphere);
else // value inside sphere
	pointOnSphere[2] = sqrt(1.0 - pow(pointOnSphere[0],2) - pow(pointOnSphere[1],2);
```

The orbit algorithm is described below(9.1):

1. The first click triggers `GLWindow::onMouseInput` and calculation of *a* from the mouse point.
2. Then, a drag triggers `GLWindow::onMouseMove` and calculation of *b* from the mouse point.

$$orbit(a, b) \qquad (9.1)$$
$$a = b$$

Task 2

The `GL_PROJECTION` matrix defines the view frustum which determines the visible part of the scene and how this part is projected onto the screen. OpenGL has two functions for `GL_PROJECTION` transformation:

- `glFrustum` for a perspective projection
- `glOrtho` for an orthographic (parallel) projection

Both functions require six parameters representing six clipping planes: left, right, bottom, top, near, and far. Figure 7 shows eight vertices of the view frustum.

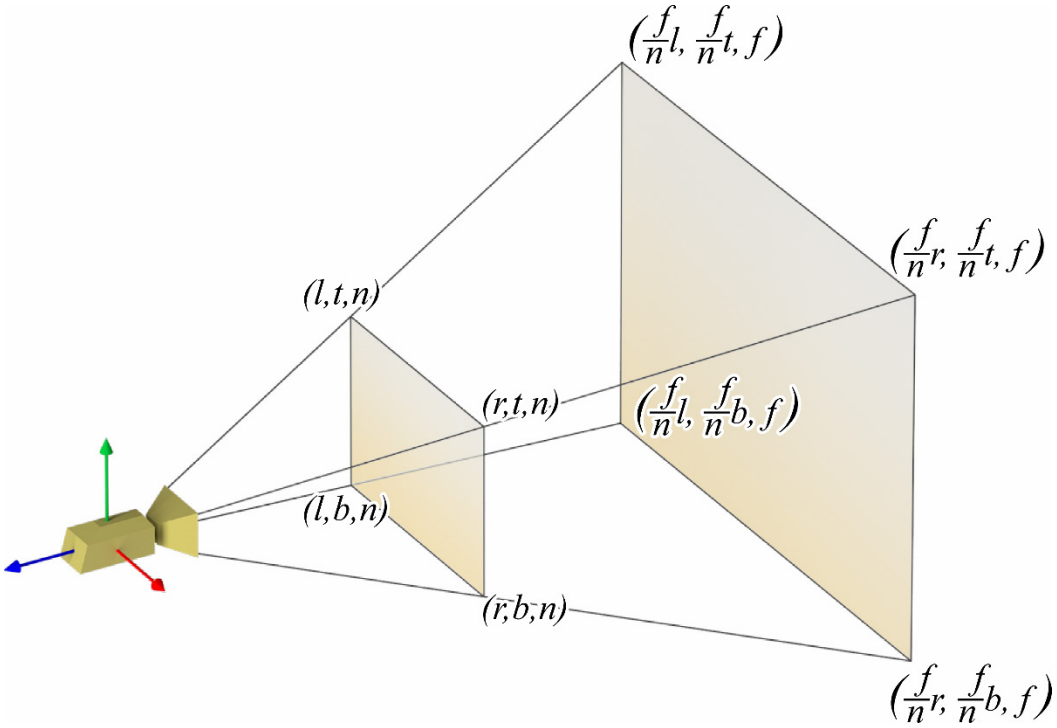


Figure 7. Eight vertices of the view frustum

A simple calculation of the ratio of similar triangles produces the vertices of the far (back) plane.

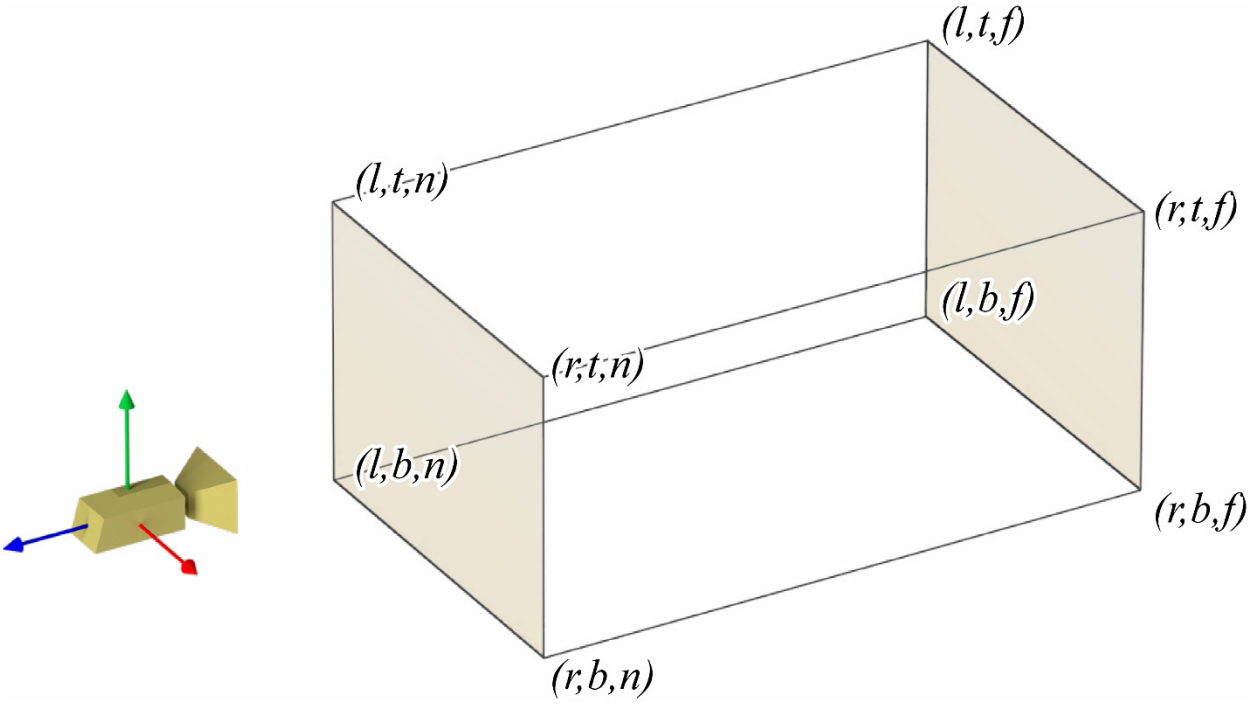


Figure 8. Orthographic view frustum

For orthographic projection, this ratio will be 1, so the left, right, bottom, and top values of the far plane will be the same as on the near plane.

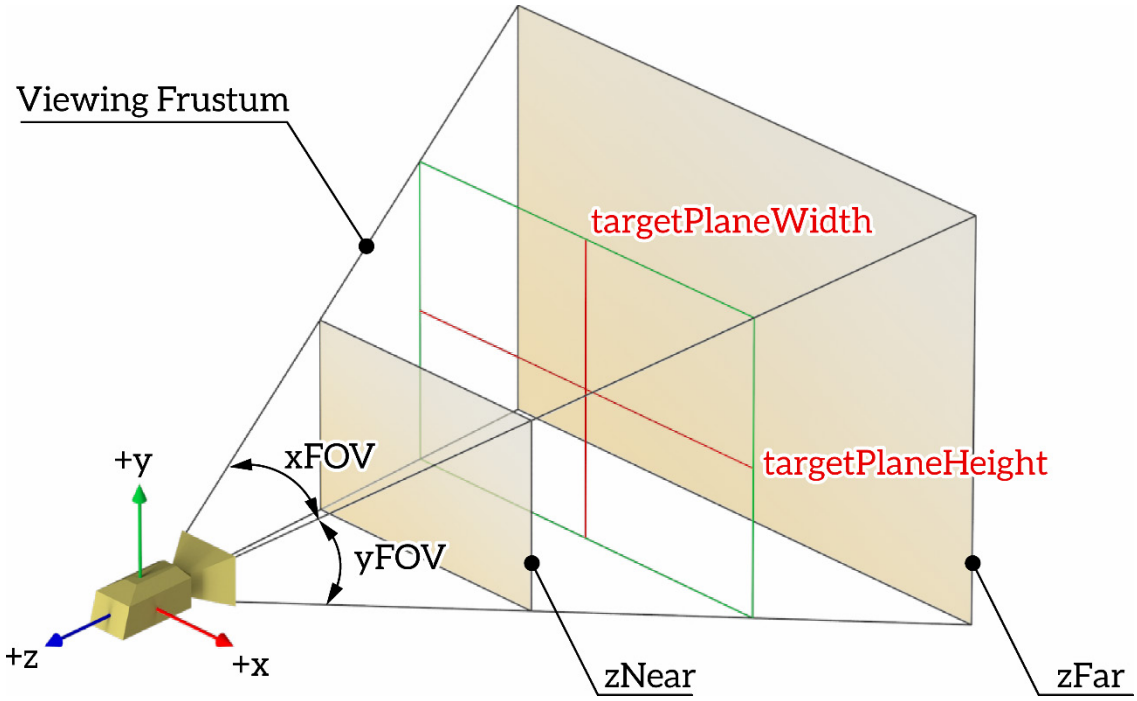


Figure 9. Perspective view frustum

Another approach is to use `gluPerspective` with less number of parameters (see List. 2):

- Vertical field of view (yFOV)
- The Width to the height aspect ratio
- Distances to near and far clipping planes

```
glm::mat4 perspective(double fovY, double aspectRatio, double zNear, double zFar)
{
	double tangent = tan(radians(fovY / 2));
	double height = zNear * tangent;
	double width = height * aspectRatio;

	return frustum(-width, width, -height, height, zNear, zFar);
}
```

Code Listing 2. Conversion from `gluPerspective` to `glFrustum`

Note: The construction of frustum is not the objective of this course. To get more details, follow the links at the end of this Lab. To build frustum, use a similar function from the glm.

Some operations, such as panning, require input parameters which belong to the target plane. The *Target plane* is the plane that looks toward the camera target point and stands at $|target - eye|$ distance from the camera position. To calculate the width and the height of the target plane, use the formulas for the near plane calculation (see List. 3).

```
double Viewport::calcTargetPlaneWidth() const
{
	return calcTargetPlaneHeight() * calcAspectRatio();
}

double Viewport::calcTargetPlaneHeight() const
{
	return 2.0 * camera.distanceFromEyeToTarget() * tan(radians(FOV / 2.0));
}
```

Code Listing 3. Calculation of the target plane

Use the following perspective parameters to implement the `Viewport` class (see List.4):

- `FOV`
- `aspectRatio`
- `zNear`
- `zFar`

Viewport.h	MeshEditor

```
#include "Camera.h"

struct ray
{
	glm::pec3 orig;
	glm::vec3 dir{ 0,0,1 };
};
```

```
class Viewport
{
public:
	glm::mat4 calcProjectionMatrix() const;

	void setViewportSize(uint32_t inWidth, uint32_t inHeight);
	void setFOV(double inFOV);
	void setZNear(double inZNear);
	void setZFar(double inZFar);
	void setParallelProjection(bool use);

	double getZNear() const;		// 0.01 by default
	double getZFar() const;			// 500 by default
	double getFov() const;			// 60 in degrees by default
	double getWidth() const;		// 1 by default
	double getHeight() const;		// 1 by default
	bool isParallelProjection() const; // false by default

	ray calcCursorRay(double x, double z) const;

	double calcTargetPlaneWidth() const;
	double calcTargetPlaneHeight() const;
	double calcAspectRatio() const;

	Camera& getCamera();
	const Camera& getCamera() const;
private:
	// TODO
};
```

Code Listing 4. Viewport header file

The `calcCursorRay` function takes two parameters of the mouse position (`x`, `y`) and returns a ray (`origin`, `direction`) from the near plane to the far plane in the WCS. This ray is the green line on Figure 10.

Implementation of `calcCursorRay` uses the `unproject` function which converts the mouse position and the specified Z coordinate (in the clip space [-1;1]) to a point in the world space.

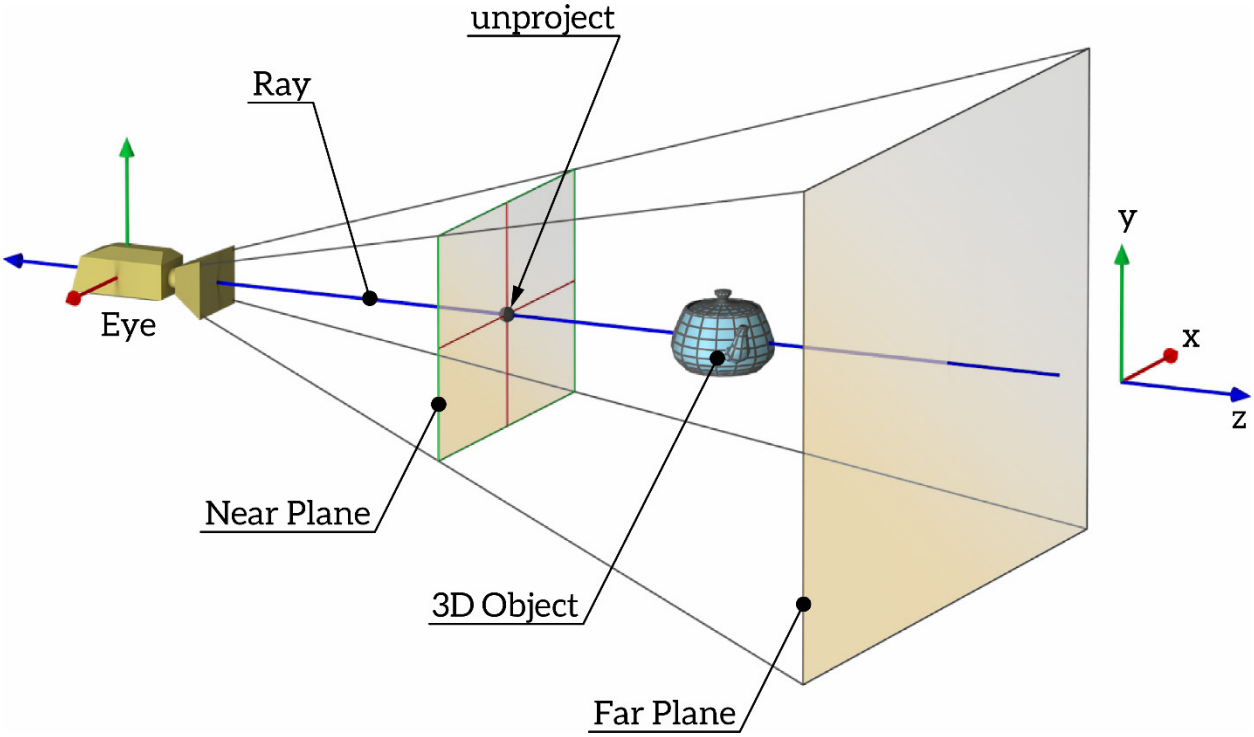


Figure 10. Cursor ray

To understand the `unproject` function, let's inverse it to a `project` function which takes a 3D point in the WCS, world, view, and projections matrices and returns a point in the screen space [0; width] x [0; height]. Multiplication of the input point by all the matrices produces a point in the clip space [-1;1] which is then mapped to the screen space [0; width] x [0; height] using simple linear equations.

The `unproject` function does the inverse chain of computations but it requires to specify the `z` value manually:

- `z = -1` for the near plane
- `z = 1` for the far plane

Implementation of `calcCursorRay` using the `unproject` is easy to perform (see List. 5).

```
glm::vec3 a = unproject(x, y, -1.0);
glm::vec3 b = unproject(x, y, 1.0);

ray cursorRay{a, normalize(b - a)};
```

Code Listing 5. Implementation of the calcCursorRay function

Task 3

Implement the `moveCamera` function (see List. 6).

main.cpp	MeshEditor

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>
```

```
#include <glfw/glfw3.h>

#include "GLWindow.h"
#include "GLRenderSystem.h"

#include "Viewport.h"

void renderScene(GLRenderSystem& rs) { /* implemented in the prev lab */ }

void moveCube(GLRenderSystem& rs, glm::vec3 offset) { /* implemented in the prev lab */
}

void moveCamera(Camera& camera, glm::vec3 offset)
{
	//TODO
}

void onKeyCallback(KeyCode key, Action action, Modifier mods)
{
	if (key == KeyCode::UP)
		moveCube(rs, /* implemented in the previous lab */);

	if (key == KeyCode::DOWN)
		moveCube(rs, /* implemented in the previous lab */);

	if (key == KeyCode::LEFT)
		moveCube(rs, /* implemented in the previous lab */);

	if (key == KeyCode::RIGHT)
		moveCube(rs, /* implemented in the previous lab */);

	if (key == KeyCode::W)
		moveCamera(viewport.getCamera(), /* TODO */);

	if (key == KeyCode::S)
		moveCamera(viewport.getCamera(), /* TODO */);

	if (key == KeyCode::A)
		moveCamera(viewport.getCamera(), /* TODO */);

	if (key == KeyCode::D)
		moveCamera(viewport.getCamera(), /* TODO */);

	if (/* key is between F1 and F7 */)
	{
		// set front, top, rear, right, left, bottom, iso views
		// TODO
	}

	if (/* key is F8 */)
	{
		// set parallel projection
		// TODO
	}
}
```

```
int main()
{
	GLRenderSystem rs;
	GLWindow window("myWindow", 640, 480);

	Viewport viewport;
	viewport.setViewportSize(640, 480);
	viewport.setFOV(60);
	viewport.setZNear(0.01);
	viewport.setZFar(500);

	window.setKeyCallback(onKeyCallback);

	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);

	while (glfwWindowShouldClose(window.getGLFWHandle())
	{
		rs.setViewport(0, 0, window.getWidth(), window.getHeight());
		rs.clearDisplay(0.5f, 0.5f, 0.5f);

		rs.setViewMatrix(viewport.getCamera().calcViewModel());
		rs.setProjectionMatrix(viewport.calcProjectionMatrix());
		renderScene(rs);
		glfwSwapBuffers(window.getGLFWHandle());
		glfwWaitEvents();
	}

	return 0;
}
```

Code Listing 6. Main.cpp file with TODO tasks

Exercises

1. Implement the `void Viewport::zoomToFit(v3 min, v3 max)` function which parameters describe the box enclosing the entire scene. This function shows the whole scene on the screen.
2. Implement Panning using `Camera::pan` and corresponding mouse events. Add window mouse callbacks to the initial code. The completed function enables to drag the camera while holding a left-click.
3. Implement Arcball using `Camera::orbit` and corresponding mouse events. Add window mouse callbacks to the initial code. The completed function enables to rotate the camera while holding a right-click.

Questions

1. Why do we need the world/view/projection matrices? How to build a view matrix? You will get a bonus for answering the following question: how to build a projection matrix (you can just name coefficients that are zero)?
2. How to implement project/unproject functions? Why do we need them?

Resources and Notes

1. OpenGL transformations:
   http://www.songho.ca/opengl/gl_transform.html
2. OpenGL projection matrix:
   http://www.songho.ca/opengl/gl_projectionmatrix.html
3. Frustum description:
   https://en.wikipedia.org/wiki/Viewing_frustum

Lab 3

Computer Graphics Course

Overview

In this Lab, you will master the *half-edge data structure* implementation. As we discussed in Lab 1, mesh rendering uses *triangle soup,* which is a very effective data structure optimized for all video cards. But this triangle soup has a big disadvantage – the absence of topology information required for mesh manipulations such as moving or deleting triangles. For example, you cannot query all the triangles adjacent to a specified triangle.

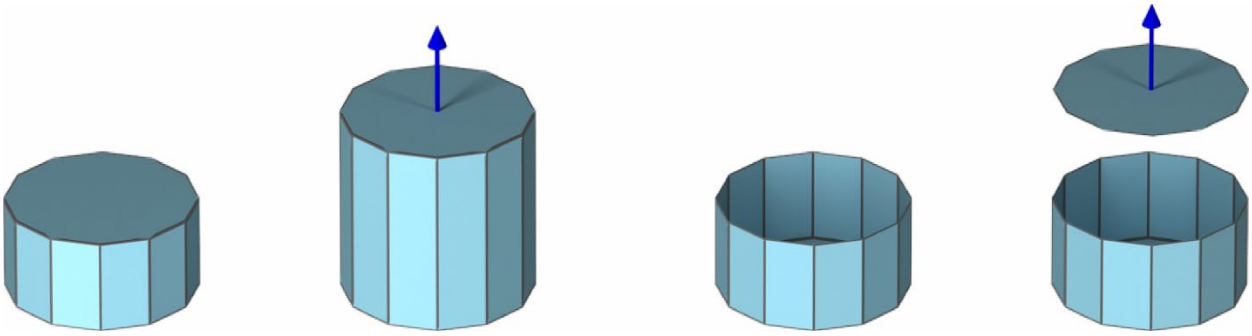


Figure 1. Moving a triangle in a triangle soup mesh

When you move a specified triangle in a mesh presented using *triangle soup*, the adjacent triangles do not move, and a hole appears in the mesh (see Fig. 1). Also, in most cases, the analysis of the adjacent triangles requires more work and therefore increases the performance cost. The *half-edge data structure* solves these problems.

Objective

The objective of this Lab is to:

1. Separate the written code into `MeshEditor` (.exe) and `GLRenderSystem` (.dll) modules.
2. Implement the `HalfEdge` data structure adding another module to `HalfEdge` (static lib).
3. Create the `Mesh` class that wraps the *half-edge data structure*, has the `render` method to render the mesh, and methods to delete and transform triangles.

Infrastructure

Table 1 shows the final directory structure of this Lab. The new files are green.

Table 1. Final directory structure

.	MeshEditor	GLRenderSystem	HalfEdge	Interfaces	ThirdParty
MeshEditor	Camera.h	GLRenderSystem.h	**HalfEdge.h**	**IWindow.h**	glm
GLRenderSystem	Camera.cpp	GLRenderSystem.cpp	**HalfEdge.cpp**	**IRenderSystem.h**	glfw
HalfEdge	Viewport.h	GLWindow.h	**HalfEdge.vcxproj**		
Interfaces	Viewport.cpp	GLWindow.cpp			
ThirdParty	**Mesh.h**	Exports.h			
MeshEditor.sln	**Mesh.cpp**	Exports.cpp			
	DynamicLibrary.h	glad.h			
	DynamicLibrary.cpp	glad.c			
	main.cpp	khrplatform.h			
	MeshEditor.vcxproj	GLRenderSystem.vcxproj			

Task 1

Add to your solution a new `GLRenderSystem` project with a dynamic library type and move `GLWindow` and `GLRenderSystem` with the needed dependencies (GLFW, Glad) there. `GLRenderSystem`

It is necessary to define the appropriate `IWindow` and `IRenderSystem` interfaces implementations of which will be in *GLRenderSystem.dll*:

1. Create *IWindow.h* and *IRenderSystem.h* in the Interfaces folder in the root directory of the project (see List. 2 and 3).
2. Add the *Exports.h* and *Exports.cpp* files to the `GLRenderSystem` project with the functions needed to create `IRenderSystem` and `IWindow` in the main `MeshEditor` module (see List. 1). Later these functions are used in the `View` implementation.
3. Add OGL_RENDER_SYSTEM_EXPORT in the preprocessor definitions in the **GLRenderSystem** project settings.

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);
```

Code Listing 1. Exports header file of the GLRenderSystem project

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;
};
```

Code Listing 2. `IWindow` header file of the GLRenderSystem project

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;
};
```

Code Listing 3. `IRenderSystem` header file of the GLRenderSystem project

The corresponding `GLWindow` and `GLRenderSystem` classes are now inherited from `IWindow` and `IRenderSystem` interfaces.

To implement the dynamic loading of *GLRenderSystem.dll* when `MeshEditor` starts, add:

1. `DynamicLibrary` class to `MeshEditor` project (see List. 4).
2. `PLATFORM=PLATFORM_WIN32` in the preprocessor definitions in the `MeshEditor` project settings.

Notes: The solution in the Listing 4 works for the Windows platform only. To implement dynamic loading, change `PLATFORM` macro and provide the implementation using `ifdef/elseif` directives in *DynamicLibrary.cpp*.

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;
}
```

Code Listing 4. Implementation of dynamic loading of GLRenderSystem.dll

As result, `MeshEditor` may use different `RenderSystem`, DirectX for example, by replacing just one line at the program startup (see List. 5).

main.cpp | MeshEditor

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>

#include "../Interfaces/IWindow.h"
#include "../Interfaces/IRenderSystem.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;
}
```

Code Listing 5. Main.cpp of `MeshEditor`

Task 2

The Half-Edge Data Structure

Triangle soup perfectly suits for simple mesh operations like loading a mesh from a drive or drawing it on a screen, but bad for many geometry processing tasks. A good tradeoff between simplicity and sophistication is the *half-edge data structure* used in this Lab.

HalfEdge

The basic idea behind the *half-edge data structure* is that here an edge consists of two half-edges, that are (see Fig. 2):

- Directed edges between the same two vertices
- Pointing in opposite directions

Each half-edge holds pointers to:

- Its *twin* half-edge
- The face it belongs
- The *next* half-edge of its face in the counter-clockwise winding (CCW) order
- The vertex of the half-edge is pointing to

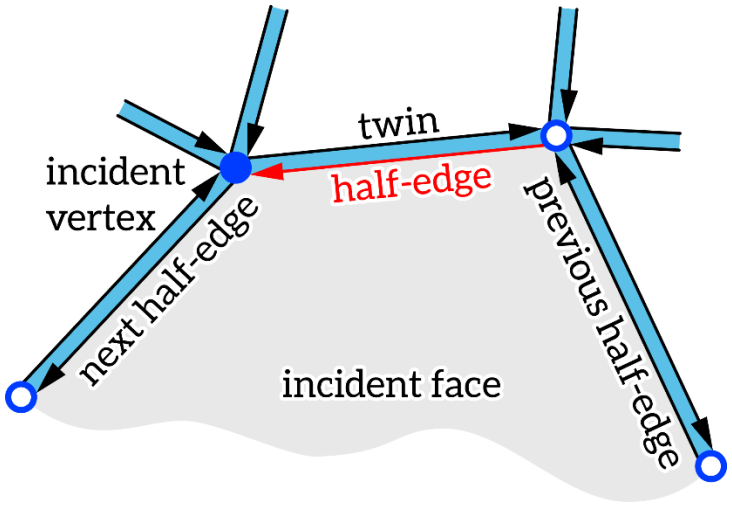


Figure 2. Half-edge

The half-edge is a boundary if its pointer to a face equals -1.

```
struct HalfEdgeHandle { int64_t index = -1; };
struct VertexHandle { int64_t index = -1; };
struct FaceHandle { int64_t index = -1; };


struct HalfEdge
{
	//If the face the half-edge belongs to is invalid (== -1) then its a boundary edge
	FaceHandle fh;
	//The vertex the half-edge directed at is always valid
	VertexHandle dst;
	//The twin half-edge is always valid
	HalfEdgeHandle twin;
	//The next HalfEdge in the CCW order is always valid
	HalfEdgeHandle next;
	//The previous HalfEdge in the CCW order can be stored for the optimization
purposes. For the triangle meshes prev = next->next->next
	HalfEdgeHandle prev;
	//HalfEdge is deleted if isDeleted equals true
```

```
        bool isDeleted;
};
```

The following conditions must always be true:

assert(heh != twin(heh));

assert(heh != next(heh));

assert(heh == twin(twin(heh));

vh is a shorthand for a vertex handle

heh is a shorthand for a half-edge handle

fh is a shorthand for a face handle

Face

Step by step the `heh=next(heh)` iteration outputs the *chain of half-edges,* which forms a `Face` (see Fig.3). A `Face` has a reference to one of its `HalfEdges(heh)`, which defines this face.

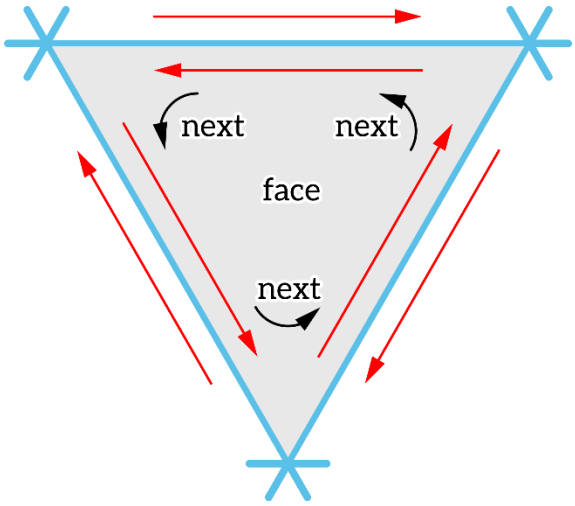


Figure 3. Face in the half-edge data structure

```
struct Face
{
        //One of the HalfEdges belonging to a Face is always valid
        HalfEdgeHandle heh;
        //Face is deleted if isDeleted equals true
        bool isDeleted;

};
```

Vertex

Step by step the `heh = next(twin(heh))` iteration outputs a vertex (see Fig. 4). A vertex has a reference to one of its outgoing `HalfEdges(heh)` which defines this vertex.

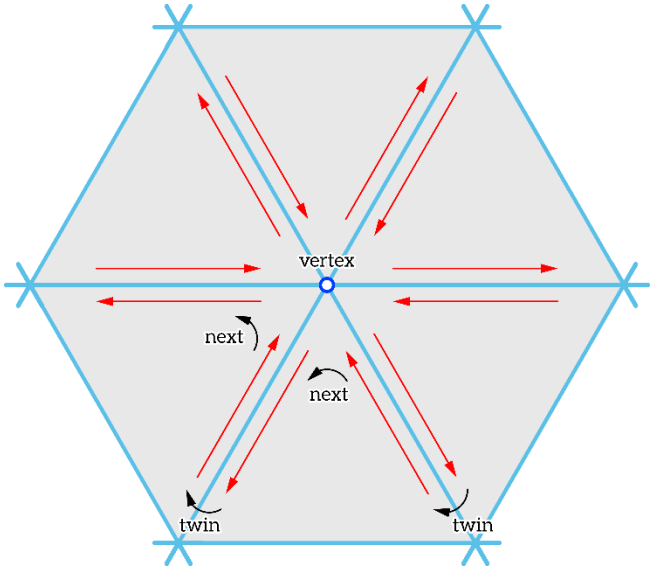


Figure 4. Vertex in the half-edge data structure

```
struct Vertex
{
	//An outgoing HalfEdge that starts at this vertex. A vertex is isolated if it is
== -1
	HalfEdgeHandle heh;
	//Vertex is deleted if isDeleted equals true
	bool isDeleted;

};
```

Let's define the main helper functions of the half-edge data structure (see List. 6): `deref`, `handle`, `next`, `twin`, `sourceVertex`, and `destVertex`. To implement `deref`/`handle` functions efficiently, `HalfEdgeHandle`, `VertexHandle`, and `FaceHandle` simply wrap `int` which is an index in the `halfEdges` array. Here `Handle` is like a C pointer. The `deref` function returns a value by the handle, and the `handle` does the opposite thing, returns a *pointer* to the `Vertex`, `Edge` or `Face` entity. A 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));
}
```

Code Listing 6. Definitions of the main helper functions

The `Face` and `Vertex` data structures need the similar functions except for `sourceVertex` and `destVertex`.

The following conditions must always be true:

assert(sourceVertex(heh) == destVertex(twin(heh));

assert(sourceVertex(heh) == twin(next(sourceVertex(heh)));

Traversal operations

For example, let's visit all the vertices of a specified `Face`. You take a `HalfEdge` (`heh`) of the face and move the *next* pointer until meeting the initial `HalfEdge` (see List. 7).

```
HalfEdgeHandle heh = deref(fh).heh;
HalfEdgeHandle start_heh = heh;
HalfEdgeHandle next_heh = heh;

do
{
	visit(destVertex(next_heh));	//do something with the vertex
	next_heh = next(next_heh);
} while (next_heh != start_heh);
```

Code Listing 7. Visiting vertices of a Face

Another example is visiting all the vertices adjacent to a specified `Vertex`. You should start with any outgoing `HalfEdge` (`heh`), move to its *twin*, then to the *next* outgoing `HalfEdge` and so on until meeting the initial `HalfEdge` (see Fig. 5 and List. 8).

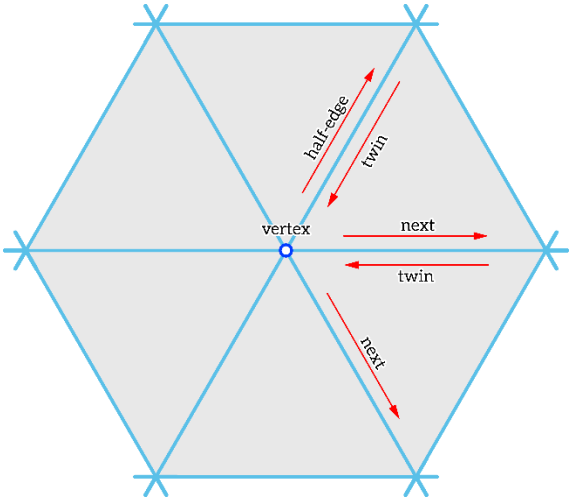


Figure 5. Visiting vertices adjacent to a Vertex

```
HalfEdgeHandle heh = deref(vh).heh;
HalfEdgeHandle start_heh = heh;
HalfEdgeHandle next_heh = heh;

do
{
	visit(destVertex(next_heh));	//do something with the vertex
	next_heh = next(twin(next_heh));
} while (next_heh != start_heh);
```

Code Listing 8. Visiting vertices adjacent to a Vertex

The `HalfEdge` representation requires a mesh to be manifold and orientable to be valid.

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, is invalid (== -1) if a boundary edge
	FaceHandle fh;
	//The vertex it points to is always valid
	VertexHandle dst;
	HalfEdgeHandle twin;
	//The next HalfEdge in the CCW order is always valid
	HalfEdgeHandle next;
	//The previous HalfEdge in the CCW order can be stored for the optimization
purposes. For the triangle meshes prev = next->next->next
	HalfEdgeHandle prev;
```

```
	//HalfEdge is deleted if isDeleted equals true
	bool isDeleted;
};

struct Face
{
	//One of the HalfEdges belonging to the Face, always valid
	HalfEdgeHandle heh;
	//Face is deleted if isDeleted equals true
	bool isDeleted;
};

struct Vertex
{
	//An outgoing HalfEdge from this vertex. It is == -1 if the vertex is isolated
	HalfEdgeHandle heh;
	//Vertex is deleted if isDeleted equals true
	bool isDeleted;
};

class HalfEdgeTable
{
public:
	//Adds vertex
	VertexHandle addVertex(glm::vec3 position);
	//Adds triangulated face
	FaceHandle addFace(VertexHandle vh0, VertexHandle vh1, VertexHandle vh2);
	//Adds quad face
	FaceHandle addFace(VertexHandle vh0, VertexHandle vh1, VertexHandle vh2,
VertexHandle vh3);
	//Builds twins for half-edges. This function must be called in the end
	void connectTwins();

	//Marks face as deleted. Doesn't remove it from array.
	void deleteFace(FaceHandle fh);
public:
	//For a given half-edge returns previous linked half-edge
	HalfEdgeHandle prev(HalfEdgeHandle heh) const;
	//For a given half-edge returns next linked half-edge
	HalfEdgeHandle next(HalfEdgeHandle heh) const;
	//For a given half-edge returns twin half-edge
	HalfEdgeHandle twin(HalfEdgeHandle heh) const;
	//For a given half-edge returns end vertex
	VertexHandle destVertex(HalfEdgeHandle heh) const;
	//For a given half-edge returns start vertex
	VertexHandle sourceVertex(HalfEdgeHandle heh) const;

	//For a given half-edge handle returns half-edge
	HalfEdge& deref(HalfEdgeHandle vh);
	const HalfEdge& deref(HalfEdgeHandle vh) const;
	//For a given half-edge returns half-edge handle
	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;

        //For a given vertex handle set/get point
        const glm::vec3& getPoint(VertexHandle handle) const;
        void setPoint(VertexHandle handle, glm::vec3 data);

        //For a given half-edge handle set/get start point (vertex)
        const glm::vec3& getStartPoint(HalfEdgeHandle handle) const;
        void setStartPoint(HalfEdgeHandle handle, glm::vec3 data);

        //For a given half-edge handle set/get end point (vertex)
        const glm::vec3& getEndPoint(HalfEdgeHandle handle) const;
        void setEndPoint(HalfEdgeHandle handle, glm::vec3 data);

        const std::vector<Vertex>& getVertices() const;
        const std::vector<Face>& getFaces() const;
private:
        // TODO
};
```

Code Listing 9. An example of a HalfEdgeTable definition

Implementation of the addVertex function

This function simply adds a new Vertex with heh equal to -1. This field must be set in the addFace function later.

Implementation of the addFace function

This function adds a new Face and adds new half-edges that enclose this face. For each of new half-edges, next/prev fields are set. For each new vertex, heh field is set and it points to its outgoing half-edge.

Implementation of the connectTwins function - how to build twins?

To fill a twin member of half-edges you need to traverse the array of half-edges and collect the pairs of initial and final vertices (si, ei). Then again traverse all half-edges and check if there is a half-edge with the vertices (ei, si). If there are two half-edges with the same vertices, but in reverse order, then they are twins.

Listing 10 shows an example of HalfEdgeTable 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);
```

```
	halfEdgeTable.connectTwins();

	return halfEdgeTable;
}
```

Code Listing 10. Example of HalfEdgeTable usage

Task 3

Finally, implement the `Mesh` class that the `IWindow` window will render (see List. 11). Leave the `getBoundingBox` function for the Lab 4.

Note: deleted faces mustn't be rendered.

Mesh.h

```
#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
};
```

MeshEditor

Code Listing 11. The Mesh class header file

Note: The `applyTransformation` function applies the transformation matrix to all the vertices of the specified face (`fh`).

Listing 12 shows the main rendering code rewritten.

main.cpp

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>

#include "../Interfaces/Window.h"
#include "../Interfaces/RenderSystem.h"
```

MeshEditor

```
#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;
```

```
}
```

Code Listing 12. The main rendering code rewritten

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 must look in the (0,0,1) direction.
5. Make sure that the above-implemented functions return a correctly built half-edge data structure. Try to change a vertex position and check if there is a hole.

Questions

1. How to build a half-edge data structure for the given mesh (the examiner draws a mesh)?

Resources and Notes

1. A half-edge data structure tutorial:
   http://kaba.hilvi.org/homepage/blog/halfedge/halfedge.htm
2. Another half-edge data structure tutorial:
   The Half-Edge Data Structure
3. There is an interesting discussion of the software design challenges of building the half-edge data structure, but their implementation differs from ours:
   OpenMesh

Lab 4

Computer Graphics Course

Overview

In this Lab, you will learn how to implement the Mesh Editor following MVO pattern. After successful completion, the Mesh Editor will be able to load, save, delete a face, and display a mesh using various settings of a camera and a viewport.

Objective

The objective of this Lab is to implement the following classes and functions:

1. `saveModel`/`loadModel` functions to import/export a COLLADA (.dae) file to/from the `Model` class.
2. A `View` that uses the `Viewport` and the `Camera` from Lab 2:
    a. An `Application` class to manage the `View` and the `IRenderSystem` implemented in Lab 3.
    b. An `Operator` class and add operators for the functions in the `Camera` and `Viewport` classes (see Table 1).
3. A `DeleteFaceOperator`, this is a separate item describing the problem of Collision Detection

Command	Key
Display a model in a separate window	V
Save the result to a COLLADA file, named initial name + modified.dae	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

Table 1. Operator functions

Infrastructure

Table 2 shows the final directory structure of this Lab. The new files are green.

Table 2. Final directory structure

.	MeshEditor	GLRenderSystem	HalfEdge	Interfaces	ThirdParty
MeshEditor	**Application.h**	GLRenderSystem.h	HalfEdge.h	IWindow.h	**tinyxml2**
GLRenderSystem	**Application.cpp**	GLRenderSystem.cpp	HalfEdge.cpp	IRenderSystem.h	glm
HalfEdge	**View.h**	GLWindow.h	HalfEdge.vcxproj		glfw
Interfaces	**View.cpp**	GLWindow.cpp			
ThirdParty	**FilterValue.h**	Exports.h			
MeshEditor.sln	**Contact.h**	Exports.cpp			
	ColladaParser.h	glad.h			
	ColladaParser.cpp	glad.c			
	TrackBall.h	khrplatform.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.vcxproj			

Dependencies

Add TinyXML-2 to parse the .dae files.

Task 1

Let's start our work with the creation of the `saveModel` and `loadModel` methods (see List. 1). Explicit lifetime management of the `Model` result is not necessary because `loadModel` returns a `unique_ptr` or a `nullptr` if a loading error occurs.

A COLLADA file may contain multiple mesh objects because it describes a scene graph that is a hierarchical representation of all objects in a scene (meshes, cameras, lights, etc.) and transformations of their coordinates.

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);
```

Code Listing 1. The `loadModel` and `saveModel` methods

A `Model` is a tree node built from the loaded COLLADA file where each `Node` stores a `Mesh`, a relative matrix transformation, and an array of pointers to other `Node`s (see List. 2).

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;
};
```

Code Listing 2. `Model`

A `Node` is a *tree node* which contains a `Mesh`, its transformation into a scene, an array of child `Nodes`, and a pointer to the parent `Node` (see List. 3). Let's discuss the `Node` methods:

- `calcAbsoluteTransform` calculates absolute transformation matrix of the `Node` recursively applying all parent transformations up by the hierarchy.
- `attachNode` adds a `newNode` to the array of children by moving the input variable. A `newNode` doesn't have a parent, or if it has, throws the `logic_error` exception.
- `deleteFromParent` pops back an item from an array of children of its parent and deletes it. **Note:** Do not use a pointer after calling this function!
- `applyRelativeTransform` multiplies current relative transformation 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
};
```

Code Listing 3. Node

Note: Use pointers cautiously for different relations:

- a raw pointer (`*`) for *points to, BUT does not own*
- a unique pointer `std::unique_ptr<T>` for *points to AND owns*

COLLADA Format

COLLADA is an XML schema for transferring between different 3D applications information about models, materials, scenes, cameras, lights, etc. Most of the mainstream 3D development tools support this standard. As a starting point to support COLLADA, we need to implement in this Lab the part of the COLLADA Release 1.4.1 specification regarding the geometric information of models only.

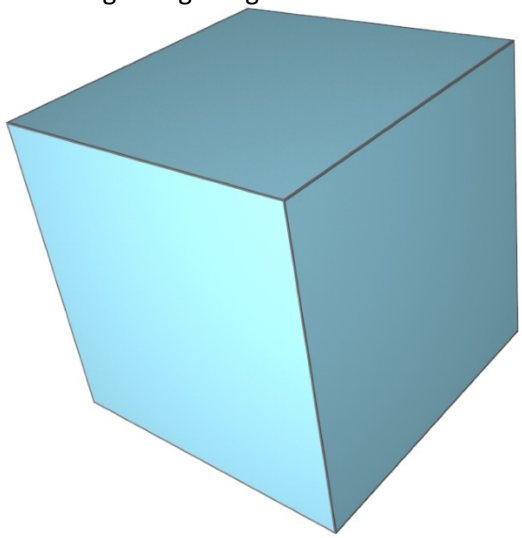


Figure 1. The 3D cube described in the List. 4

Let's discuss an example of a COLLADA XML (see List. 4) describing the 3D cube model shown in Figure 1.

```
<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
          </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>
```

Code Listing 4. COLLADA XML of the 3D cube at Fig. 1

The start `<collada>` and the end `</collada>` tags wrap all COLLADA information about the cube model.

There are several elements here to pay attention on:

1. `<library_geometries>` keeps geometry:
   a. `<geometry id="Cube-mesh">` defines geometry name `"Cube-mesh"`.
   b. `<mesh>` describes a mesh with eight vertices (the 24 floats are the x, y and z coordinates of these vertices) and six four-sided polygons created from these vertices.
   c. `<p>` holds integers, indices in the vertex array.
   d. `<vcount>` specifies that the values in the `<p>` element must be interpreted as six four-sided polygons.
2. `<library_visual_scenes>` keeps the cube instantiation:
   a. `<node>` instantiates the `"#Cube-mesh"` geometry and translates it to the (32;42;16) location in the WCS.

This example produces a `Model` object that contains:

- one `Node` with relative transformation matrix that corresponds to translation ({32;42;16}).
- `Mesh` which encapsulates the `HalfEdgeTable` consisting of 8 vertices, 12 faces, and 38 half-edges.

Task 2

The `Application` is the main class that manages objects of an application, including the creation of a `View` and limiting its lifetime (see List. 5). The `View` must not outlive the `Model`. The scope of the `main` function limits the lifetime of the `Application`. The `run` function starts an outer infinite loop that lasts until a 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
};
```

Code Listing 5. Application class header file

View

Code Listing 6 shows the header file of the `View`. The `update` function renders the `Model`. To implement the rendering:

1. Recursively traverse Nodes in the `Model`.
2. Get transformation of the node in the scene using `Node::calcAbsoluteMatrix`.
3. Pass the node transformation to `IRenderSystem::setWorldMatrix`.
4. Call `Mesh::render`.

The `View` owns a pointer to the `IWindow` interface, and its constructor creates a window. The `raycast` method is described in detail while `DeleteFaceOperator` implementation.

View.h	MeshEditor

```
#include <memory>
#include <vector>
#include <string>

#include "Viewport.h"
#include "Model.h"
#include "OperatorDispatcher.h"

#include "../Interfaces/IRenderSystem.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
};
```

Code Listing 6. View class header file

The `View` has four `AddOperator` functions. Some examples of their usage are described below:

- `view->addOperator(ButtonCode::Button_Left, std::make_unique<PanOperator>());`
  Activates/deactivates the pan operator when a left click starts/ends
- `view->addOperator(KeyCode::E, KeyCode::ESCAPE,`
  `std::make_unique<EditMeshOperator>());`
  Activates/deactivates `EditMeshOperator` when E/Esc is pressed
- `view->addOperator(KeyCode::F1, [](View& view, Action, Modifier)`
  `{view.getViewport().getCamera().setFrontView(); });`
  Sets the Front View operator when F1 is pressed and deactivates it immediately

The `Application::createView` function sets all the operators needed.

The `OperatorDispatcher` class follows keyboard commands and mouse events and triggers the corresponding operator. It has three private functions visible to the View class only, which are called by a constructor (see List. 7).

```
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);
});
```

Code Listing 7. Three private functions of the `OperatorDispatcher` class

Implement the `OperatorDispatcher` class (see List. 8). Start with the one-click operators because their implementation is the easiest. Then work on the complex operators with the different run and stop keys.

Note: If an operator is a bind with a key that another operator already uses, then `logic_error` exception is thrown. For example, the chain of the following calls with F1:

1. addOperator(`KeyCode::F1, myOperator1`);
2. addOperator(`KeyCode::F1, KeyCode::F2, myOperator2`);

must throw 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
};
```

Code Listing 8. `OperatorDispatcher header file`

The `Operator` must be configured to perform an action on the `Model` and the `View` (see List. 9).

Operator.h | MeshEditor

```
#include "View.h"

#include "../Interfaces/IWindow.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) {}
};
```

Code Listing 9. Specifying the Operator

Start/finish of the `Operator` calls the `onEnter`/`onExit` functions respectively.

Task 3

Implement `DeleteFaceOperator` following this logic (see List. 10):

1. Find the closest contact to the camera (intersected face and a pointer to the node) between the user ray and the tree nodes.
2. In case of intersection, take the intersected node and face.
3. Call `deleteFace` to delete the 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);
                        }
                }
        }

};
```

Code Listing 10. `DeleteFaceOperator`

Contact Detection

You need to implement the `View::raycast` function that takes mouse position and `FilterValue` which can be a `Node` or a `Manipulator` (see List. 11).

FilterValue.h	MeshEditor

```
enum class FilterValue : uint32_t
{
	Node = 1,
	Manipulator = 2,
	NM = (uint32_t)(Node | Manipulator), //Node and Manipulator
};
```

Code Listing 11. FilterValue header file

This function returns an array of *Contacts* between a `ray` and a `Model` (tree node) sorted by the distance to the observer along the `ray`.

`Contact` has two fields (see List. 12):

1. `face`, the intersected face
2. `node`, the corresponding intersected node

Contact.h	MeshEditor

```
struct Contact
{
	HalfEdgeLib::FaceHandle face;
	Node* node;
};
```

Code Listing 12. Contact header file

Figure 2 shows a contact detection:

- The green dotes are contacts.
- The purple line is a cursor ray.
- The blue box is a clip space.

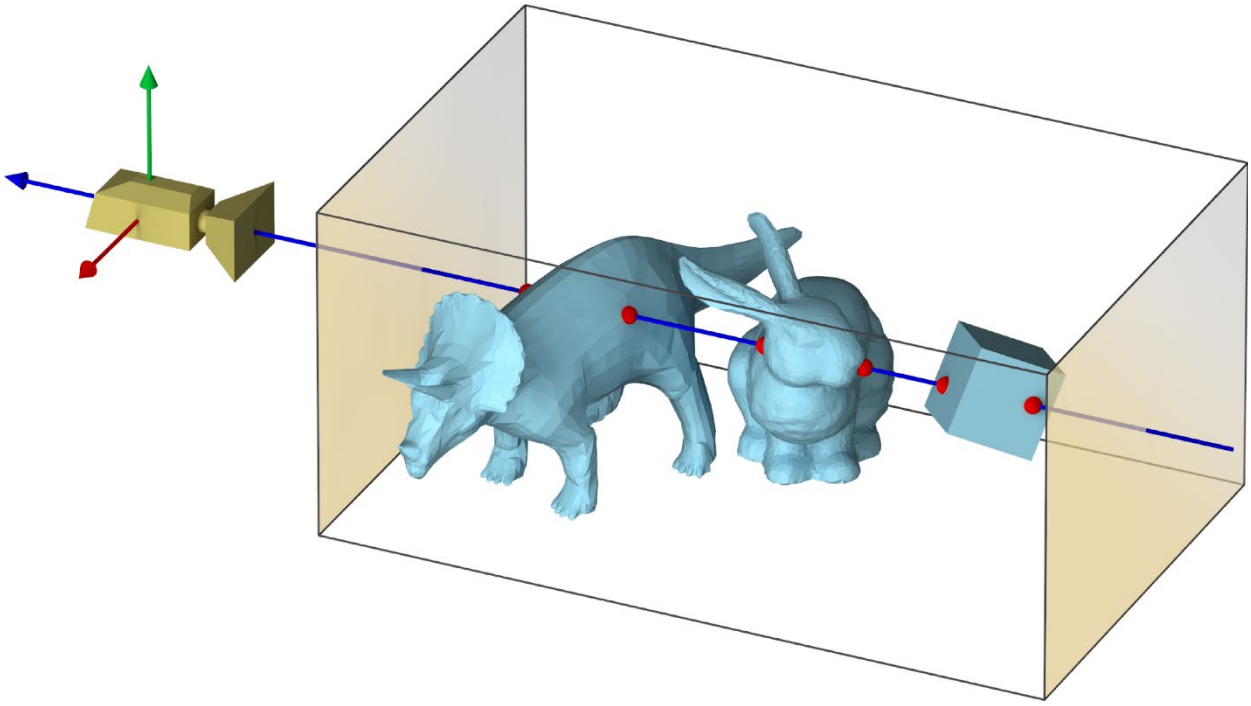


Figure 2. Contact detection

The Brute Force Algorithm

The simple brute force algorithm has big complexity and low performance because it checks all faces in all Meshes of all Nodes:

For each i-th object in the Scene:

1. *Calculate the intersection between the ray and i-th object.*
2. *In case of intersection, add the intersected face and node to the contacts list.*
3. *Sort the contacts by the distance to the observer along the ray.*

This algorithm consists of two phases:

1. The Broad phase performs the following tasks:
   a. Approximates geometries of all Nodes with simple convex geometries like *Bounding Boxes* or *Spheres*.
   b. Checks the ray for a simple geometry; this is cheaper than checking each face in the `Mesh`.
   c. Outputs an array of Nodes. Their bounding volumes intersect with the ray.
2. The Narrow phase checks the ray against the Node `Mesh` for each Node in the array the Broad phase outputs.

The Broad phase

There is a trick to speed up the process of finding potential tree node candidates for intersecting the ray:

1. Build bounding boxes for each node with the `getBoundingBox` function of a `Mesh` that returns a bounding box.
2. Calculate the intersection between the ray and bounding boxes of the nodes.
   Note: Consider the transformation matrix in the process of bounding box calculation.

Here instead of checking all faces in all meshes, you check their bounding boxes only. The Broad phase outputs an array of nodes and the potential candidates.

The Narrow phase

The Narrow phase checks the ray against `Mesh` of each candidate from the previous step to find the closest intersected triangle to the camera. The output is an array of intersected triangles sorted by the distance to the camera.

Finally, filter the array with the `filterValue` and depending on the user selection provide the required actions:

- For intersection only with Manipulators, remove all the nodes from the array that are not Manipulators.
- For intersection only with `Node` (geometry), remove all the nodes from the array that are Manipulators.

Exercises

1. Think how to optimize `Node::calcAbsoluteTransform` for the large hierarchies.
2. Implement the `Pan` operator using the exercises solutions from Lab 2.
3. Implement the `TrackBall` operator using the exercises solutions from Lab 2.
4. Implement `Broad phase` using Octree and measure the performance gain. Think how to add `Octree` without removing the brute force algorithm for checking a list of bounding boxes.

Questions

1. For a given COLLADA file build Node data structure.

Resources and Notes

1. COLLADA 1.4.1 specification:
   https://www.khronos.org/files/collada_spec_1_4.pdf
2. Octree data structure:
   https://en.wikipedia.org/wiki/Octree
3. Broad phase vs Narrow phase:
   http://planning.cs.uiuc.edu/node214.html
4. Barycentric coordinates and ray-triangle intersection:
   https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

Lab 5

Computer Graphics Course

Overview

In this Lab, you will learn how to implement:

- The `Translation` and the `Rotation` manipulators as a basis to assemble the `Triad` manipulator, which is used to implement `TransformMeshOperator` for a mesh transformation.
- The `EditMeshOperator` using `TranslationManipulator`.

In the final version of the Mesh Editor you can:

- Select a Mesh of a Node and change its transformation.
- Select a triangle and move it along its normal.

Objective

The objective of this Lab is to implement five classes:

1. `TranslationManipulator`.
2. `EditMeshOperator`.
3. `RotationManipulator`.
4. The `Triad` manipulator.
5. `TransformMeshOperator`.

Infrastructure

Table 1 shows the final directory structure of this Lab. The new files are green.

Table 1. Final directory structure

.	MeshEditor	GLRenderSystem	HalfEdge	Interfaces	ThirdParty
MeshEditor	**TransformNode.h**	GLRenderSystem.h	HalfEdge.h	IWindow.h	glm
GLRenderSystem	**TransformNode.cpp**	GLRenderSystem.cpp	HalfEdge.cpp	IRenderSystem.h	glfw
HalfEdge	**EditMesh.h**	GLWindow.h	HalfEdge.vcxproj		tinyxml2
Interfaces	**EditMesh.cpp**	GLWindow.cpp			
ThirdParty	**Manipulator.h**	Exports.h			
MeshEditor.sln	**Manipulator.cpp**	Exports.cpp			
	Triad.h	glad.h			
	Triad.cpp	glad.c			
	RotationManipulator.h	khrplatform.h			
	RotationManipulator.cpp	GLRenderSystem.vcxproj			
	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				

Task 1

Many CAD applications have a *Triad* – three arrows perpendicular to each other and attached to geometry, so dragging them moves geometry (see Fig. 1). A *Triad* consists of three translation manipulators and three rotational manipulators. The latter are perpendicular to each other planes in the form of circles.

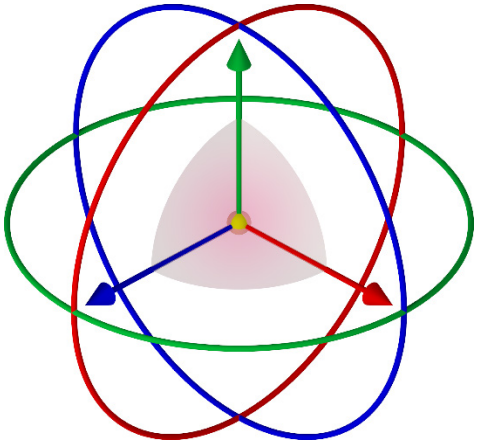


Figure 1. Triad

Let's discuss the mechanism of translation. To move an attached object in the specified direction, a user selects an arrow for that direction and *drags* it — *moves along this arrow while holding it*. Hence, translation implementation requires calculating the distance of the cursor movement along the arrow. This movement consists of three actions:

1. `Push`, user selects an arrow.
2. `Drag`, user moves the arrow along its axis while holding it.
3. `Release`, the user releases the arrow.

To communicate with geometry, a `Node` subscribes to `callback` and monitors the changes of the arrow position. If translation manipulator is a translation matrix and if rotational manipulator is a rotation matrix, it is enough to pass *delta* to the `callback`. *Delta* is the value of the arrow movement relative to the previous frame.

`Manipulator` is a `Node` which has three methods (see List. 1):

1. `handleMovement`, calls `sendFeedback` and must be subscribed to the `IWindow` events to be called when a user does `Push`, `Drag` or `Release`.

2. `setCallback`, sets `callback` to be called when a user does `Push`, `Drag` or `Release`.
3. `sendFeedback`, multiplies transformation matrix of the `Manipulator` by `deltaMatrix` to move the arrow and calls `callback` passing `deltaMatrix` as a parameter.

Manipulator.h | MeshEditor

```
#include "Viewport.h"
#include "Node.h"

enum class MovementType { Push, Drag, Release };

class Manipulator : public Node
{
public:
	virtual void handleMovement(MovementType movementType, const Viewport& viewport,
double x, double y) {}

	void setCallback(const std::function<void(const glm::mat4&)>& inCallback);
protected:
	void sendFeedback(const glm::mat4& deltaMatrix);
private:
	std::function<void(const glm::mat4&)> callback;
};
```

Code Listing 1. `Manipulator`

`TranslationManipulator` is a `Manipulator` (see List. 2). A user sets a manipulator position with the transformation matrix. This `Node` has a mesh with arrow geometry in the specified direction. Since this geometry is in the `LCS`, you need to use the `meshArrow` function from the Lab 3 Exercises to transform each vertex of `meshArrow` to `dir_L` by rotation matrix computed as the rotation from (0,0,1).

TranslationManipulator.h | MeshEditor

```
#include "Manipulator.h"

class TranslationManipulator : public Manipulator
{
public:
	TranslationManipulator(glm::vec3 dir_L);
	void handleMovement(MovementType movementType, const Viewport& viewport, double
x, double y) override;
	void setDirection(glm::vec3 inDir_L);
private:
	glm::vec3 startPoint_L;
	glm::vec3 dir_L;
};
```

Code Listing 2. `TranslationManipulator`

Code Listing 3 shows an example of translation manipulator usage.

```
// We want to translate in X direction in LCS
TranslationManipulator manipulator({1,0,0});
```

```
// In WCS this position is (10,0,0)
manipulator.setRelativeTransform(translation({ 10,0,0 }));
//objectToMove is the Node we want to transform
manipulator.setCallback([&objectToMove](const glm::mat4& delta)
{
	objectToMove.applyRelativeTransform(delta);
});
```

Code Listing 3. An example of the `TranslationManipulator` usage

`handleMovement` algorithm

The `handleMovement` algorithm has two logical parts (see Fig. 2):

1. In case of a click, the closest point of intersection between cursor and manipulator rays is set as the start point in `LCS`.
2. In case of a cursor drag or a cursor release:
    a. The closest point of intersection between cursor and manipulator rays is set as the current point in `LCS`.
    b. `delta` vector between the current and the start points is calculated.
    c. `callback` is called with `delta` passed as a parameter.
    d. `delta` is applied to the relative matrix of the manipulator to translate this manipulator.

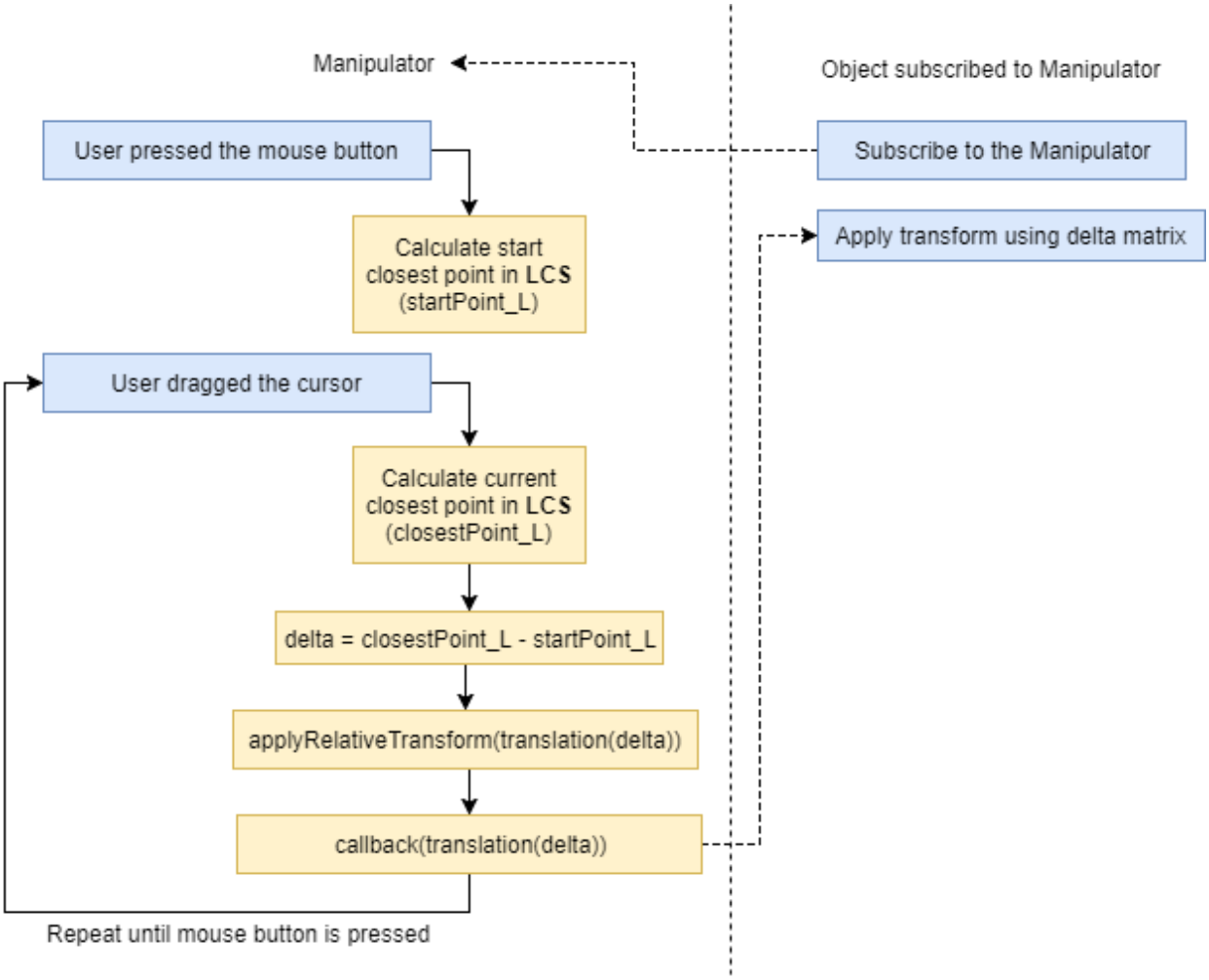


Figure 2. Algorithm of `handleMovement`

Let's implement this algorithm. Your task here is to follow the instructions.

At first, multiply `dir_L` of the Manipulator using `Manipulator::calcAbsoluteTransform` to obtain `ray_W`, the ray of the Manipulator in `WCS`.

Then, use the `Viewport::calcCursorRay` function from Lab 2 to obtain `cursorRay_W`.

This part depends on the `movementType`:

1. If it equals to `MovementType::Push` (see Fig. 3):
    a. Compute `startPoint_W` as the closest point of the intersection between `cursorRay_W` and `ray_W`.
    b. Transform `startPoint_W` to `LCS` of the Manipulator.
    c. Get `startPoint_L` as result.

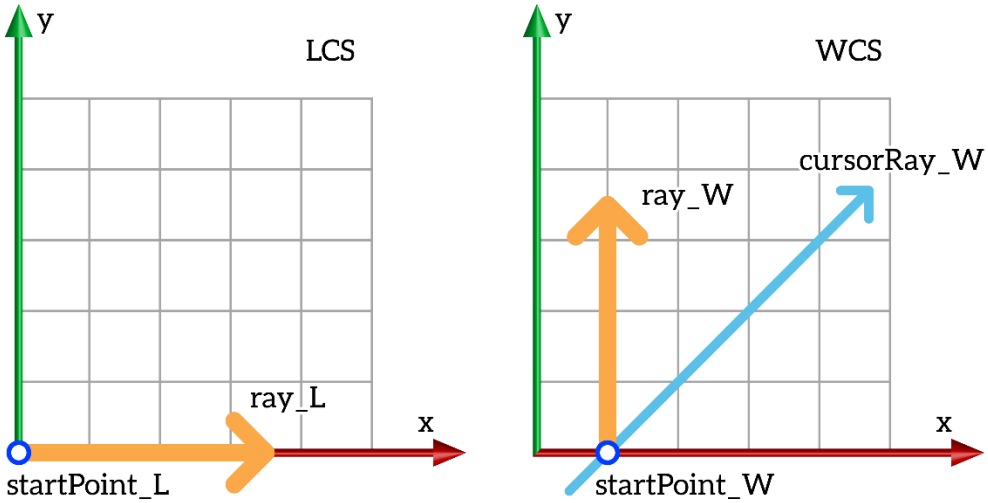


Figure 3. `startPoint_W` and `startPoint_L`

2. If it equals to `MovementType::Drag` or `MovementType::Release` (see Fig. 4):
    a. Compute `closestPt_W` as the closest point of the intersection between `cursorRay_W` and `ray_W`.
    b. Transform `closestPt_W` to `LCS` of the Manipulator.
    c. Get `closestPt_L` as result.

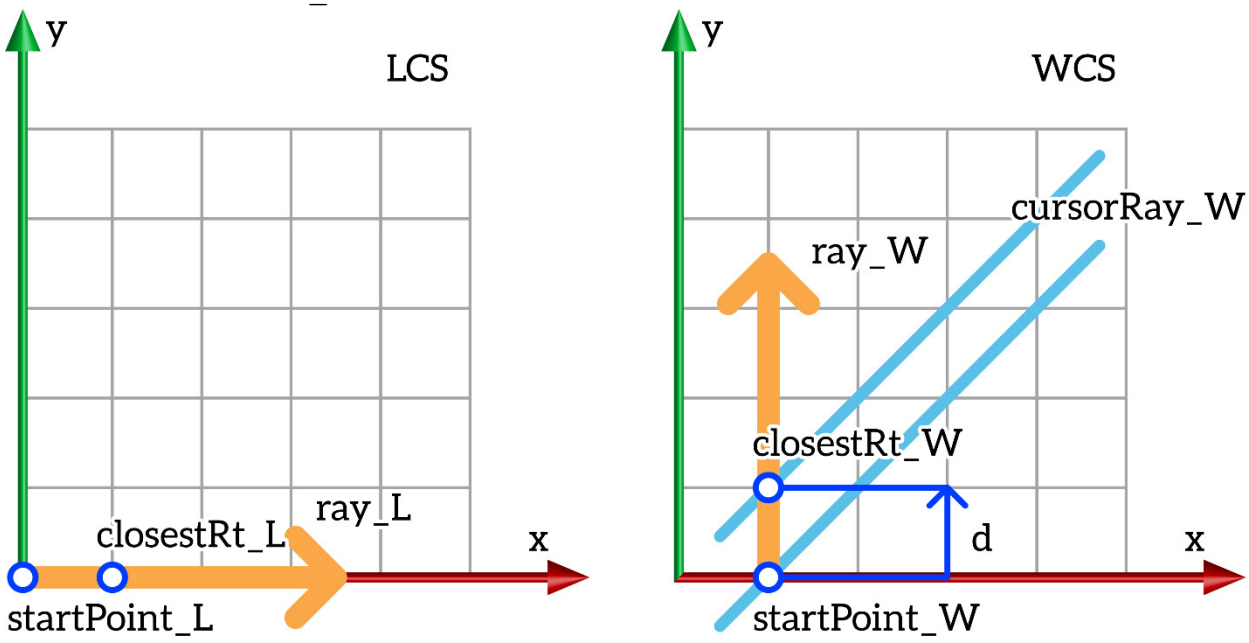


Figure 4. The current and the start points in LCS and WCS

    d. Calculate `delta = translation(closestPt_L - startPoint_L)`.
    e. Call `sendFeedback(translation(delta))`.

Later, when implementing the `Triad`, you will rewrite `sendFeedback`. For now, it simply multiplies the transformation matrix of the `Manipulator` by `deltaMatrix` to move the arrow, and to call the callback with `deltaMatrix` (see List. 4 and Fig. 5).

```
void Manipulator::sendFeedback(const glm::mat4& deltaMatrix)
{
	Node::applyRelativeTransform(deltaMatrix);

	if (callback)
		callback(deltaMatrix);
}
```

Code Listing 4. `sendFeedback`

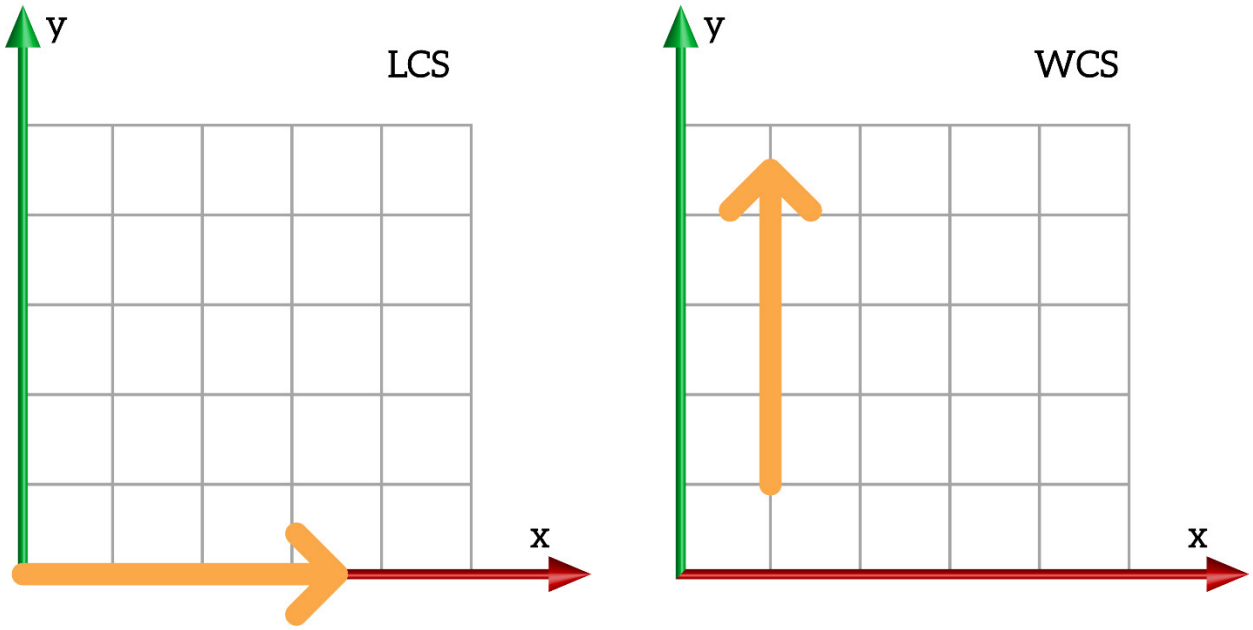


Figure 5. `delta` transformation

Task 2

To complete this task, you must implement `EditMeshOperator` using `TranslationManipulator` to select and move a triangle in the given direction (see List. 5).

This process is complex:

1. You start this feature when press E, the `EditMeshOperator` is launched.
2. When you select a face, a `TranslationManipulator` appears in its center with direction equal to the selected face normal.
3. When you move the selected `TranslationManipulator`, its face moves correspondingly (see Fig. 6).
4. When you select another face, a new `TranslationManipulator` appears in its center, and the previous `TranslationManipulator` disappears.

5. You stop this feature when press Esc, both `EditMeshOperator` and `TranslationManipulator` are removed.

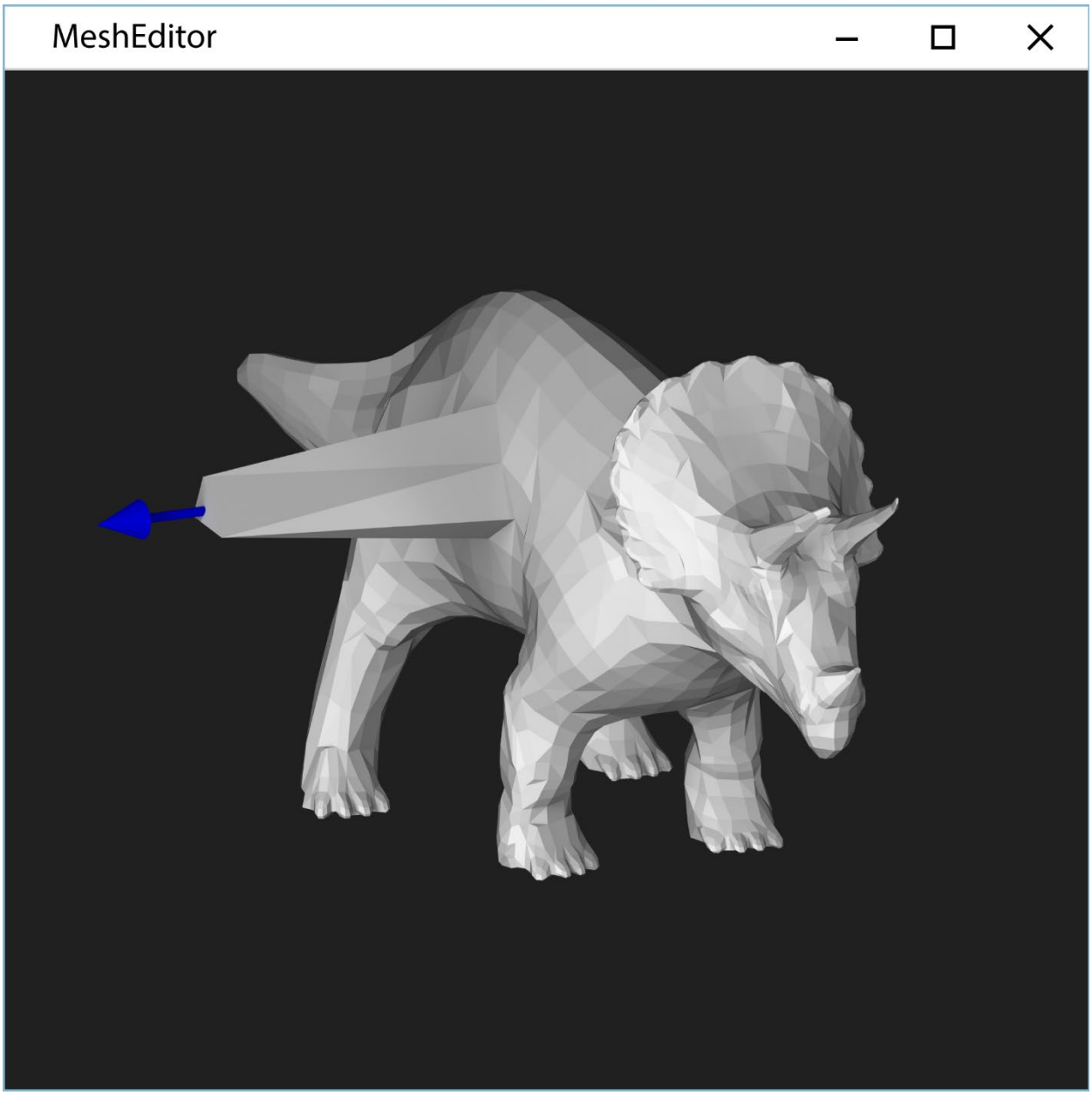


Figure 6. Feature to select and move a triangle in the given direction

The `EditMeshOperator` can be in one of two states only:

- *Idle*, when you have not selected a `Manipulator` or a `Face` yet.
- *Edit*, when you select a `Manipulator` and move arrows of the `TranslationManipulator`.

Idle state

Actions depend on whether a selected `Node` is a `Manipulator`, or not:

- If it is, change its state to *Edit.*
- If it is not, create a `TranslationManipulator` in the center of the selected `Face` and return to the *Idle* state.

Edit state

There are four possible events for you to handle:

1. Push and drag, call `handleMovement` from the captured `TranslationManipulator`.
2. Release, change the current state to *Idle.*
3. Press **E**, `EditMeshOperator::onEnter`, set the *Idle* state.
4. Press **Esc**, `EditMeshOperator::onExit`, complete the current state and clear all the resources to delete the attached Translation Manipulator.

Use the corresponding `onMouseMove` and `onMouseInput` events for the first two cases.

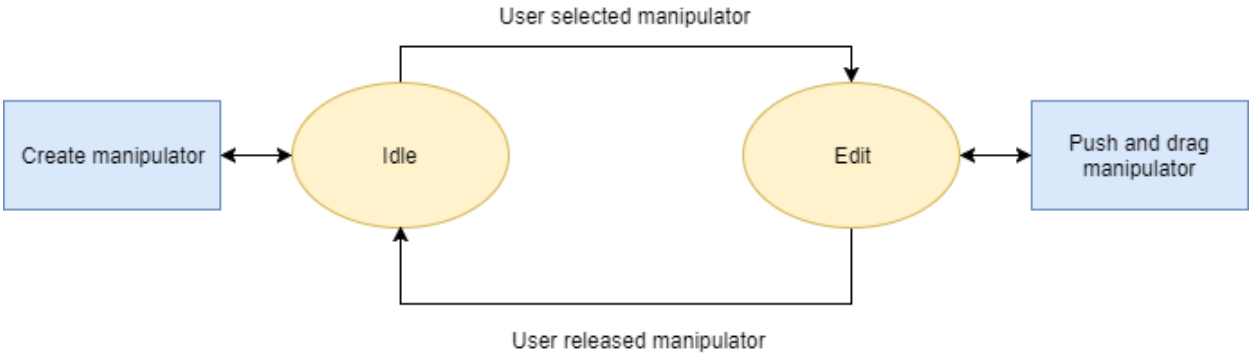


Figure 7. The finite state machine

Implement the finite state machine shown at the Figure 7.

EditMeshOperator.h	MeshEditor

```
#include "../Operator.h"
#include "../Manipulator.h"

#include <map>

class EditMeshOperator : public Operator
{
public:
	//TODO: if needed
private:
	void onEnter(View& view) override;
	void onExit(View& view) override;

	void onMouseMove(View& view, double x, double y) override;
	void onMouseInput(View& view, ButtonCode button, Action action, Modifier mods,
double x, double y) override;
```

```
	//TODO
};
```

Code Listing 5. `EditMeshOperator`

Task 3

`RotationManipulator` is also a `Manipulator` and uses the same algorithm to set the geometry in the specified direction as the `TranslationManipulator` (see List. 6) with the only difference that there is `meshTorus` instead of `meshArrow` function.

RotationManipulator.h | MeshEditor

```
#include "Manipulator.h"

class RotationManipulator : public Manipulator
{
public:
	RotationManipulator(glm::vec3 dir_L);
	void handleMovement(MovementType movementType, const Viewport& viewport, double
x, double y) override;
	void setDirection(glm::vec3 inDir_L);
private:
	glm::vec3 startPoint_L;
	glm::vec3 dir_L;
};
```

Code Listing 6. `RotationManipulator`

Implementation of `handleMovement` function sticks to the same pattern used for `TranslationManipulator`. The only difference is that you need to:

1. Calculate an intersection point between a ray of the `Manipulator` and a plane in the form of a torus.
2. Build a vector from the origin to this intersection point.
3. Calculate the rotation between the start and current intersection vectors.

Task 4

`Triad` is a *tree node*, therefore to build it, just add the necessary child nodes to the tree (see List. 7).

Triad.h | MeshEditor

```
#include "Manipulator.h"
class Triad : Manipulator
{
public:
	Triad()
	{

	Manipulator::attachNode(std::make_unique<TranslationManipulator>(glm::vec3{
1,0,0 }));
```

```
	Manipulator::attachNode(std::make_unique<TranslationManipulator>(glm::vec3{
0,1,0 }));

	Manipulator::attachNode(std::make_unique<TranslationManipulator>(glm::vec3{
0,0,1 }));

		Manipulator::attachNode(std::make_unique<RotationManipulator>(glm::vec3{
1,0,0 }));
		Manipulator::attachNode(std::make_unique<RotationManipulator>(glm::vec3{
0,1,0 }));
		Manipulator::attachNode(std::make_unique<RotationManipulator>(glm::vec3{
0,0,1 }));
	}
};
```

Code Listing 7. `Triad`

To support complex manipulators like `Triad`, you need to rewrite the `sendFeedback` method. So that it is able to transform the child manipulators of a composite manipulator (see List. 8):

1. Find the root manipulator.
2. Transform the relative matrices of the manipulator children by `deltaMatrix`.
3. Call callback.

After successful implementation of these steps, if you pull an arrow, the rest arrows of the `Triad` move to the same delta. Otherwise, this arrow comes off the `Triad`.

```
void Manipulator::sendFeedback(const glm::mat4& deltaMatrix)
{
	auto root = dynamic_cast<Manipulator*>(getParent());

	if (root)
		for (auto& sn : root->Node::getChildren())
			sn->applyRelativeTransform(deltaMatrix);
	else
		Node::applyRelativeTransform(deltaMatrix);

	if (callback)
		callback(deltaMatrix);
}
```

To support complex manipulators like `Triad`, you need to rewrite `sendFeedback` method so, that it is able to transform the child manipulators of a composite manipulator (see List. 8):

Task 5

In this task, use a `Triad` to implement `TransformMeshOperator` (see List. 9).

This process is complex:

1. You start this feature when press **T**, the `TransformMeshOperator` is launched.

2. When you select a mesh of a `Node`, a `Triad` appears in its center.
3. When you move or rotate the `Triad`, the selected mesh moves or rotates correspondingly.
4. When you select another `Node`, a new `Triad` appears in its center, and the previous `Triad` disappears.
5. You stop this feature when press **Esc**, both `TransformMeshOperator` and `Triad` are removed.

TransformNodeOperator.h	MeshEditor

```
#include "../Operator.h"
#include "../Manipulator.h"

#include <map>

class TransformNodeOperator : public Operator
{
public:
	//TODO: if needed
private:
	void onEnter(View& view) override;
	void onExit(View& view) override;

	void onMouseMove(View& view, double x, double y) override;
	void onMouseInput(View& view, ButtonCode button, Action action, Modifier mods,
double x, double y) override;
	//TODO
};
```

Code Listing 9. `TransformMeshOperator`

Exercises

1. Think how to optimize the construction of `TranslationManipulator` without modifying the outputs of the `meshArrow` function.
   Hint: `TranslationManipulator` is a Tree `Node`.
2. Find the cases when the `TranslationManipulator` is parallel to the screen plane.
3. Develop a mechanism that enables to make a move in the case described in the previous exercise.
4. Find the case when the `RotationManipulator` is perpendicular to the screen plane.
5. Develop a mechanism that enables to make a move in the case described in the previous exercise.
6. Implement the `ScaleManipulator`.
7. Add vertex selection and its movement, if a user's cursor is above a vertex, create the translation handle in the vertex position with its normal equal to the normal averaged of the adjacent faces.

Questions

1. Explain the algorithms of how rotation and translation manipulators work.

Resources and Notes

1. http://www.osgart.org/index.php/Object_Manipulation