# Lab 1

## Overview

# In this lab work, you will learn how to create a window, render a geometry, and process the input.

## Objective

# To write a class **GLWindow** for creating a window

# To write a class **GLRenderSystem** for rendering triangles

# Using the created classes to draw a simple cube (by implementing *renderScene* function) and implement *moveCube* function that translates cube in the scene using keyboard arrows

## Prerequisites

Before starting the work with the Lab, it is recommended the read the following materials:

1. [glfw documentation](#glfw)
2. [OpenGL basics](#OpenGLBasics)

## Infrastructure

To do this, create a **MeshEditor** project in your solution as a console application with main.cpp and an empty main function. Create ThirdParty folder and put glm and glfw libraries there. As a result of doing Lab work, the directory structure showing in the figure below should appear:

|  |  |  |
| --- | --- | --- |
| **.** | **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](#glm)) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

### glfw

This cross-platform library is used to create windows and simplifies working with OpenGL.

### glad

OpenGL functions (core or extension) must be loaded at runtime, dynamically, whenever the function is not a part of the original OpenGL ABI (application binary interface) platform.

The difference between core OpenGL functions and extensions is that the core functions are found in the OpenGL specification, while extensions are functionality that may or may be not available in addition to what the OpenGL version provides. Both extensions and newer version core functions are loaded through the same mechanism.

The GLAD library is needed to get the OpenGL functions from the OpenGL32.dll library.

In order to get each function from OpenGL32.dll not manually, you can use existing generators to get the necessary pointers to the functions. To do this, download the utility *https://github.com/Dav1dde/glad*, which generates three files: glad.h, glad.c, khrplatform.h and add them to the MeshEditor project. Files glad.h, glad.c, khrplatform.h should not be put in the ThirdParty directory, as these files are generated and serve only to conveniently get pointers to the OpenGL functions. In order to get the OpenGL functions from the OpenGL32.dll library it is necessary to call the *gladLoadGLLoader* function after the window is created.

## Task 1

Before creating a class **GLWindow**, let's take a look at how to create and show a window in the code. The code below creates an instance of the **GLWindow** and runs it in the infinite loop until user presses ESC. Any keystrokes are displayed in the console window.

|  |
| --- |
| #include <glfw\glfw3.h>  #include "GLWindow.h"  int main()  {  glfwInit();  GLWindow window("myWindow", 640, 480);  window.setKeyCallback([&](KeyCode key, Action action, Modifier mods)  {  std::cout<<"Key: "<<glfwGetKeyName(key, 0)<<" is pressed."<<std::endl;  });  while (glfwWindowShouldClose(window.getGLFWHandle())  {  glfwSwapBuffers(window.getGLFWHandle());  glfwWaitEvents();  }  glfwTerminate();  return 0;  } |

An important component of the window, besides the display, is the input. To process the input, 4 callbacks are needed: *setKeyCallback*, *setCursorPosCallback*, *setMouseCallback*, *setScrollCallback*, which are implemented as *std::function*.

|  |  |
| --- | --- |
| 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  }; | |

The code below shows how to create a window using *glfwCreateWindow*, which returns a glfw handle. An important point is to call the function *glfwMakeContextCurrent*, which sets the current context for the window. Context is the OpenGL data that is bound to the window and used to display the geometry in the window.

Next, OpenGL functions are loaded from OpenGL32.dll using *gladLoadGLLoader* function. It is necessary that this function is called after the window is created and the context is set. It is needed to do this only once, so that’s why a static boolean variable is used.

Further, with each glfw window, a pointer to **GLWindow** is associated using *glfwSetWindowUserPointer* function. This is necessary in order to get the **GLWidow** via glfw handle in the callbacks and call the required **std::function**.

|  |
| --- |
| 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 \*/);  } |

At this stage a window should be created. Next, rendering of geometric primitives should be implemented.

## Task 2

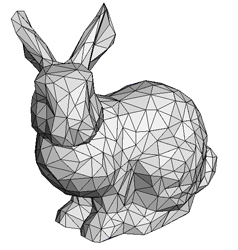
Write a class called **GLRenderSystem** that provides several operations:

1. *renderTriangleSoup* - draw a list of triangles. *A triangle soup is a group of unorganized triangles, with generally no relationship whatsoever*
2. Set a transformation matrix, view and projection matrices
3. Set the light source

The class will look the following:

|  |  |
| --- | --- |
| 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  }; | |

To display geometry in OpenGL, triangles are used. The choice of this geometric primitive is not accidental. On a triangle, mathematical calculations are most effective. Basically, any complex surface can be represented as a bunch of triangles. Each triangle consists of 3 vertices.



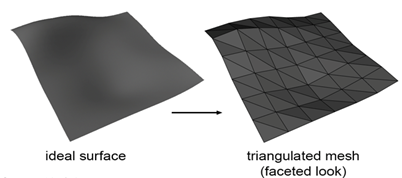
Let’s start with *Vertex declaration*. It contains two fields:

1. position – is needed to define location of the vertex in the space
2. normal – is needed for the lighting calculation

It is also possible to specify other attributes, such as texture coordinates. For simplicity, basic attributes such as position and normal are used.

### Flat Shading vs. Smooth Shading and Vertex Normals

The problem with triangle meshes is that they can't represent smooth surfaces perfectly (unless the triangles are very small). Flat shading is a shading model where the lighting calculations use only triangle’ normals. The problem with this approach is that it gives the object a faceted look (shown in the images below).



To address this problem, Gouraud shading is used. The idea is to produce continuous shading across the surface of a triangular mesh. To do that the concept of vertex normal was introduced. Rather than computing the normal for the triangle, the normal is stored at each vertex of the mesh. The color of the point on the surface is calculated using normal that is linearly interpolated between the vertex normals defined at the triangle's vertices. In order to enable smooth shading and enable depth test (which is off by default in OpenGL) it is needed to call *GLRenderSystem::init* function after window is created.

|  |
| --- |
| void GLRenderSystem::init()  {  glShadeModel(GL\_SMOOTH);  glEnable(GL\_DEPTH\_TEST);  glDepthFunc(GL\_LEQUAL);  } |

### Phong Lighting

The OpenGL API provides functions for setting up the lights:

1. Enable/disable up to 8 individual light sources
2. For each light, specify RGB values for ambient , diffuse and specular
3. It’s possible to specify several different kinds of light sources, including:
   1. A distant point light source at infinity, defined by a vector in 3D space. The Light (L) vector points in the same direction for all vertices, and constant power. L - the unit-length vector from the surface to the light
   2. A directional light source that you must aim with a direction vector
   3. 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 no light attenuation with distance, attenuation by , attenuation by , or a weighted sum

### Phong Materials

The reflectance is specified at each vertex using the [Phong](#PhongShadingModel) materials model. Each material you describe includes , , , values for red, green and blue, plus a single shininess exponent integer . The on-screen color is then:

where

- emitted light coming from material itself, unaffected by illumination;

- ambient illumination from a light, (effects of multiple lights are additive);

- ambient reflectance from the material;

- diffuse illumination strength from a light;

- diffuse reflectance from the material;

*N* - the unit-length surface normal vector for the material, see *Vertex::normal*;

*L* - the unit-length vector from the surface to the light, N·L provides the cosine falloff for surfaces lit from the side, and the *max(0, N·L)* prevents from negative-valued results if the light moves behind the object;

*Att* - the attenuation factor for distance from the light, if any;

- specular illumination strength from the light;

- specular reflectance from the material;

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

*V* - the unit-length vector from the surface to the camera (or eye position);

- Shininess exponent integer value to set the sharpness of the specular reflection.

## Now that it is described how to display triangles and set lights, it will be shown how to move 3D objects in space.

### Transformations

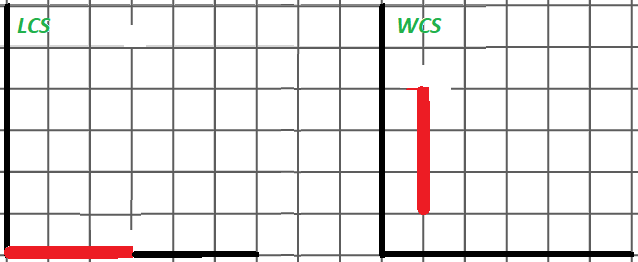
To create a computer-graphics display of an object, we must consider its geometric characteristics, such as size, shape, location, orientation, and its spatial relationship to other objects nearby. If we are to describe, measure, and analyze these characteristics, we must place the object within some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

### World Coordinate System (WCS)

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

### Local Coordinate System (LCS)

We use a local coordinate system to define an object independent of the world (global) coordinate system, without having to specify a location or orientation in the global system.



Let's say you have a cylinder in the **LCS** that looks in the direction (1,0,0), with the bottom point (0,0,0) and the top point (3,0,0). The task is to locate cylinder in the **WCS** so that it looks in the direction (0,1,0) and located in (1,0,0) point. To do this, use the following transformation matrix:

|  |  |  |
| --- | --- | --- |
| LCS to WCS matrix | Bottom point in WCS | Top point in WCS |
|  |  |  |

Now cylinder vertices in the world space. We moved from the local space to the world space (all vertices are given relative to the center of the world). Inverse transformation that convert points from **WCS** to **LCS** looks like this:

|  |  |  |
| --- | --- | --- |
| WCS to LCS matrix | Bottom point in LCS | Top point in LCS |
|  |  |  |

Here we also list several useful transformations that will be needed in the further Lab works:

1. Four-dimensional transforms. There is a compact way to represent transforms above:
2. [Rotation matrix from axis and angle](#RotationMatrixFromAngleAndAxis):

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

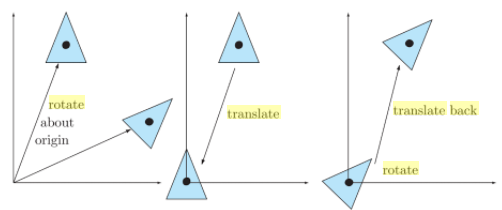
Rotation matrix from two vectors and normal:

double sign = glm::dot(normal, glm::cross(a, b)) > 0 ? 1 : -1;

double angle = sign \* angle(a, b);

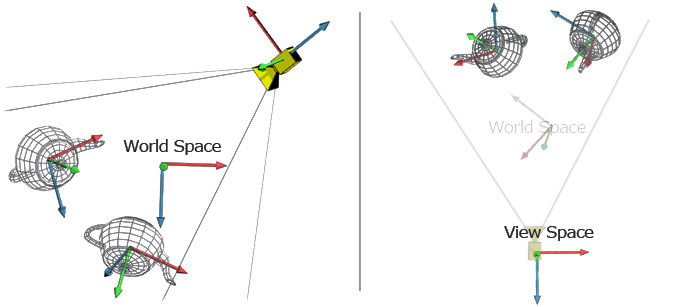
glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f), angle, normal);

1. If, for example, we want to rotate an object around its own axis, it is necessary to translate the object to the origin, then rotate the object, and finally translate back. In the code it looks like this: translation(T)\*rotation(N, angle)\*translation(-T). Note that transformations are often done around the origin.



### View Coordinate System

We locate objects relative to the observer in this coordinate system. Doing this simplifies the mathematics of projecting the object's image onto the picture plane if we arrange it so that we look along one of the principal axes (-Z in OpenGL).

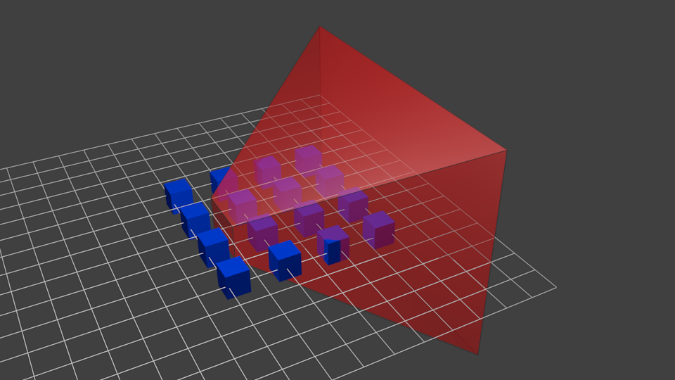


As result all objects are in the camera space (or eye space – the world relative to the camera). This transformation will be explained in detail in Lab 2.

However, when the object is displayed, the distance to the camera plays the key role. For two vertices with the same x and y, a vertex having a larger z value should be displayed closer than the other. This is called a perspective projection.

### Projection Space

The transform that converts eye-space coordinates into clip-space coordinates is known as the projection transform. The projection transform defines a view frustum (red figure in the image below) that represents the region of eye space where objects (blue cubes) are viewable.



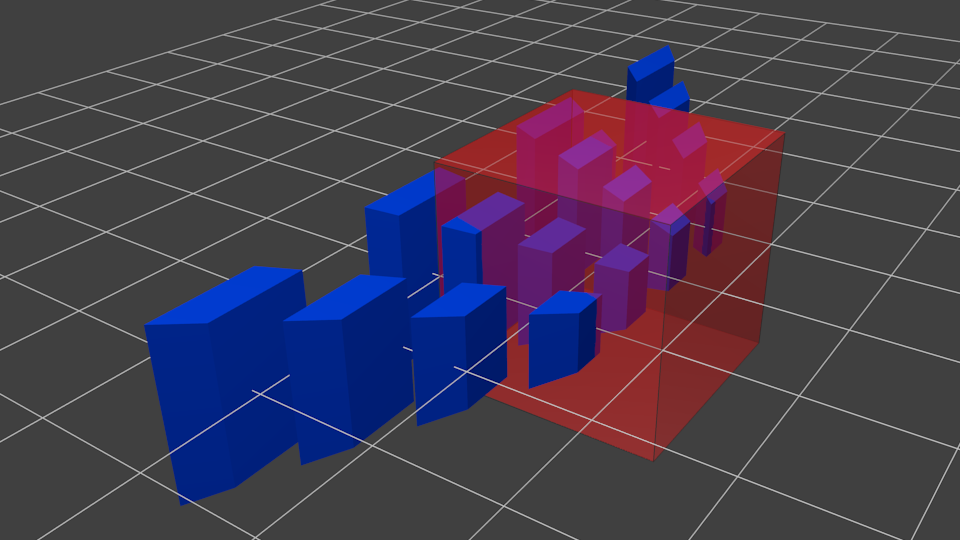
Only polygons, lines, and points that are within the view frustum are potentially viewable when rasterized into an image. In OpenGL, everything that is viewable must be within an axis-aligned cube such that the **x**, **y**, and **z** components of its clip-space position are less than or equal to its corresponding w component.

*−w≤x≤w*

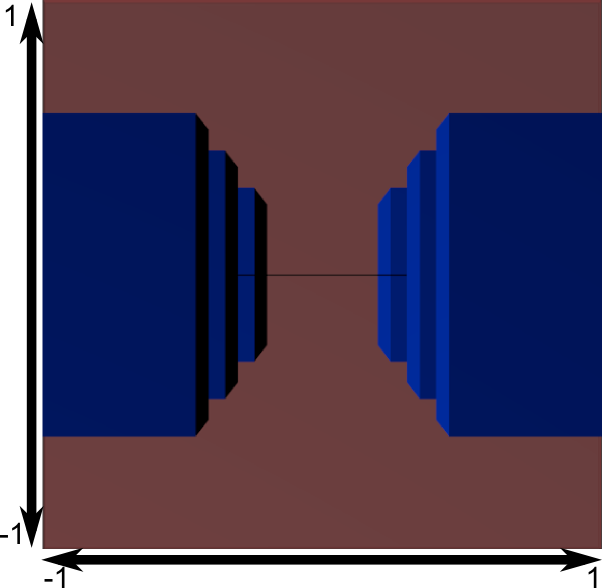
*−w≤y≤w*

*−w≤z≤w*

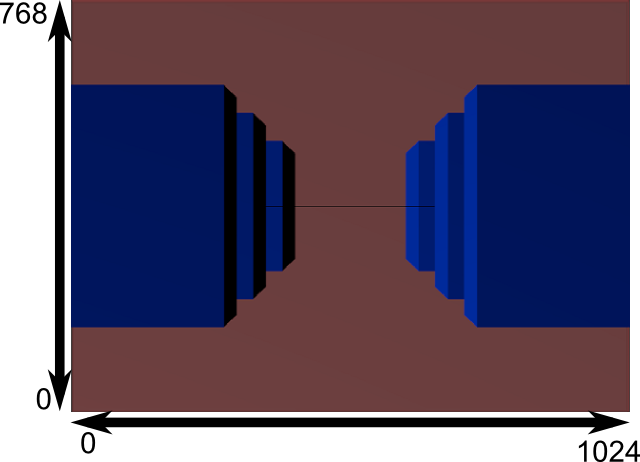
If **x**, **y**, **z** components are divided by w component, then all visible points will be in the range [-1,1] both in **X** and **Y** dimensions. At this stage clip space coordinates are transformed into the normalized device coordinates (all objects are in a cube - everything that is inside the cube is displayed on the screen). This space is very handy for clipping (anything outside the [-1,1] range is outside the camera view area).



Objects that are closer to the camera are displayed larger, and those that are further are small. The projection transformation will be explained in detail in Lab 2.



The resulting image is a square and transformations are applied to stretch the image according to the current window size:



## Task 3

Now implement the *renderScene/moveCube* functions, which draws/moves a cube using **GLRenderSystem**.

|  |  |
| --- | --- |
| 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  }  int main()  {  GLRenderSystem rs;  GLWindow window("myWindow", 640, 480);  window.setKeyCallback([&](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 \*/););  });  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;  } | |

*viewMatrix* sets the view matrix, which is responsible for setting the matrix that transforms World space into the Camera space, and *projMatrix* that sets the perspective projection. For now, it is considered that the meaning of these two variables, is just to see what's on the screen. Their construction will be discussed in Lab work 2.

As a result of doing the lab work an application that displays and moves a cube should be created.

## Exercises

1. As an exercise, try setting different transformation matrices in *setWorldMatrix* and see how to use the transformation matrix in order to place 3D object in the scene, after that create the wall from the cubes
2. Add functions to **GLRenderSystem** that implements directional and point lights
3. Add function to **GLRenderSystem** that sets the material with Phong material parameters

## Resources and Notes

1. <http://www.glfw.org/> - glfw library

1. <https://learnopengl.com/> - OpenGL course

1. <https://glm.g-truc.net> - glm library

1. <https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model> - Phong shading model

1. [https://en.wikipedia.org/wiki/Rotation\_matrix#Rotation\_matrix\_from\_axis\_and\_angle](https://en.wikipedia.org/wiki/Rotation_matrix" \l "Rotation_matrix_from_axis_and_angle) - Rotation matrix from axis and angle