# Lab 2

## Overview

# In this lab work, it will be shown how to create a camera and a viewport. In the Lab 1, two functions *setModelView* and *setProjectionMatrix* contained hard-coded values that set the model view matrix and the projection matrices so that you could see the unit cube on the screen. In this lab work, it is necessary to add camera movement/rotation using the keyboard. **Camera** class provides the *calcViewMatrix* function, result of which should be passed to *GLRenderSystem::setModelView*. **Viewport** provides the *calcProjectionMatrix*, result of which should be passed to *GLRenderSystem::setProjectionMatrix*.

## Objective

# To write a class **Camera** for creating a camera

# To write a class **Viewport** for creating viewport

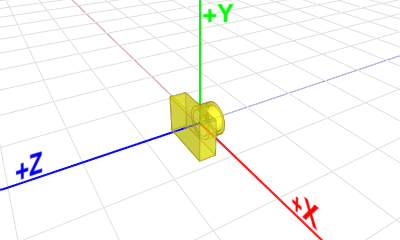
# Using implemented **Camera**/**Viewport** classes add simple camera movement using a keyboard

## Infrastructure

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

## Task 1

OpenGL does not explicitly define neither camera object nor a specific matrix for camera transformation. Instead, OpenGL transforms the entire scene inversely to a space, where a fixed camera is at the origin (0,0,0) and always looking along -Z axis.



This space is called *Eye space*. OpenGL uses a single GL\_MODELVIEW matrix for both object transformations to world space and camera transformation to eye space. Usually [ModelView](#OpenGLTransformations) matrix is broken into two logical sub matrices: *ModelView = View \* Model*

Each object in the scene is transformed with its own *Model* matrix first, then the entire scene is transformed reversely with *View* matrix. In this lab work, only the *View* matrix for a camera transformation in OpenGL will be explained. **Camera** class should be based on View matrix construction theory below.

### LookAt

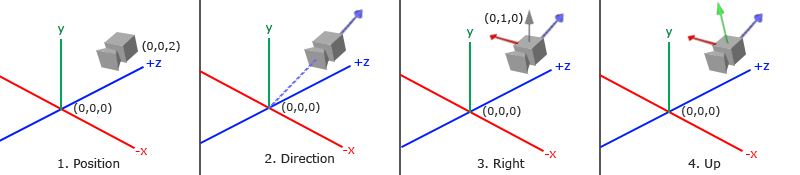
*gluLookAt* will rotate and translate the world in a way, that the camera will be located at (0, 0, 0) and looks towards the negative z-axis. This is the camera setup used in OpenGL. The camera never moves, the world does. *gluLookAt* takes **eye** position (camera location) as an input, the **target** point at which camera looks at and **up** vector which defines direction to the viewer ‘ceiling’. The **eye**, **target** and **up** are defined in the **WCS**. Further it will be described how *LookAt* function is constructed.

Translation of scene is done by matrix that calculated on the basis of inverse position from the eye position to the origin. Scene rotation is performed in such a way that the camera is located at the origin and looks in the -Z direction.

Translation part of *lookAt* is calculated as t = (0,0,0) – eye;

In order to build rotation part of *lookAt*, three additional vectors **right**, **up**, **forward** are constructed. Later using these three vectors rotation matrix will be constructed.

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



Therefore, the rotation part of *lookAt* equals to:

Finally, the view matrix is obtained by multiplying translation and rotation matrices together.

Now, knowing the principles of constructing the *lookAt*, the camera will be implemented based on *lookAt* parameters. The **Camera** is set with the help of **eye** and **target** points and **up** vector.

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

Thus, in order to get the View matrix needed for the **GLRenderSystem**, it is enough to call the *calcViewMatrix* function of the camera:

glm::mat4 Camera::calcViewMatrix() const

{

return glm::lookAt(eye, target, up);

}

And pass the result of the *calcViewMatrix* to *GLRenderSystem::setViewMatrix.*

Implement the following camera operations:

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

In order to move the camera, it is enough to add *delta,* on which you need to move the camera, to *eye* and *target*.

void Camera::setDistanceToTarget(double D)

The function moves the camera by the specified distance to the target point.

eye = target - f \* D;

# void Camera::transform(const glm::mat4& trf)

Transformation is performed by multiplying the **eye**, **target** and **up** by the transformation matrix. Keep in mind the difference between transformation of vector and point coordinates.

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

Rotation around the axis (point, axis) by a given angle can be implemented using the *translate* and *transform* functions. To do this, first move the camera to (0,0,0)-point, calculate the rotation matrix from axis and angle, apply rotation transform and move the camera back to point-(0,0,0).

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

Set new values for **eye**, **target**, and **up**. In addition to the usual assignment of variables, it is necessary to take into account the potential problem when the new **forward** vector is not perpendicular to **newUp**. In this case, it is needed to adjust **up** vector:

Next, consider the standard operations for setting the view planes:

void Camera::setFrontView()

{

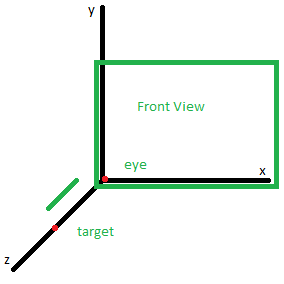
double D = distanceFromEyeToTarget();

setEyeTargetUp(target + glm::vec3{ 0,0,1 }, target, { 0,1,0 });

setDistanceToTarget(D);

}

In order to set Front View, it is necessary to set eye as (0,0,1) and put target to (0,0,0). This way the camera will look in the direction of the negative Z axis. In order to preserve the distance to the target point after the **eye***,* **target***,* **up** calculations set D value using *setDistanceToTarget* function.



void Camera::setRightView()

{

glm::vec3 oldTarget = target;

setFrontView();

rotate(oldTarget, { 0,1,0 }, pi \* 0.5);

}

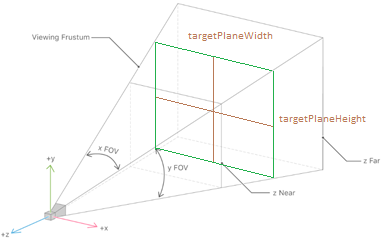
To set Right View, the following trick could be used: save the **target** value, set the front view (the **target** value changes), then rotate the camera about the y axis and the old target point value by .

Use these two described approaches to implement the rest View (Top, Rear, Bottom, Left, Iso) functions.

Next two functions (*pan* and *orbit*) allows you to move camera in a continuous motion. This means that for each mouse movement it is necessary to calculate the current parameters from the mouse coordinates, take the delta between the current and old parameters, apply the operation, and store the current coordinates.

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

Pan transformations result while applying equal vector transformations of both the **eye** point, and the **target** point. The simplest of these are a pan to the right, left, up, or down respective to the current camera view. **u**, **v** is set within the width and the height of the target plane.



Target plane is described in the **Viewport** section below. In order to map mouse coordinates **x**, **y** to **u**, **v** it is necessary to convert mouse point [0;0] x [width; height] range to point on the target plane [-targetPlaneWidth; -targetPlaneHeight] x [targetPlaneWidth; targetPlaneHeight].

Then the algorithm for panning goes like this:

1. Mouse button pressed (correspond to *IWindow::onMouseInput*):

*u0, v0 is calculated from the mouse point*

1. Cursor dragged (correspond to *IWindow::onMouseMove*):

*u, v is calculated from the mouse point and delta is calculated*

*dU = u0 – u*

*dV = v0 – v*

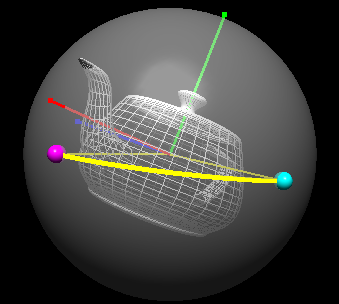
*pan(dU, dV)*

*u0 = u*

*v0 = v*

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

Suppose the object view is set from some arbitrary point in space, where the object remains in a fixed position (**target** position is not changed) and the **eye** position moves from point to point. In order to implement such motion (also called *Arcball*) consider the following mathematical trick:



Imagine a sphere in the center of which is our object, which the camera is looking at. Now if the user selects a point on the sphere and then moves the cursor to another point on the sphere, the angle between these two vectors will give the desired rotation (Orbit matrix) around the object (**target** point).

The final calculations will look like:

In the first equation it is necessary to take backward (-forward) vector and rotate it by using Orbit matrix. Then calculate new **eye** position by adding transformed backward vector to the current **target** value. Next, Orbit matrix construction is shown.

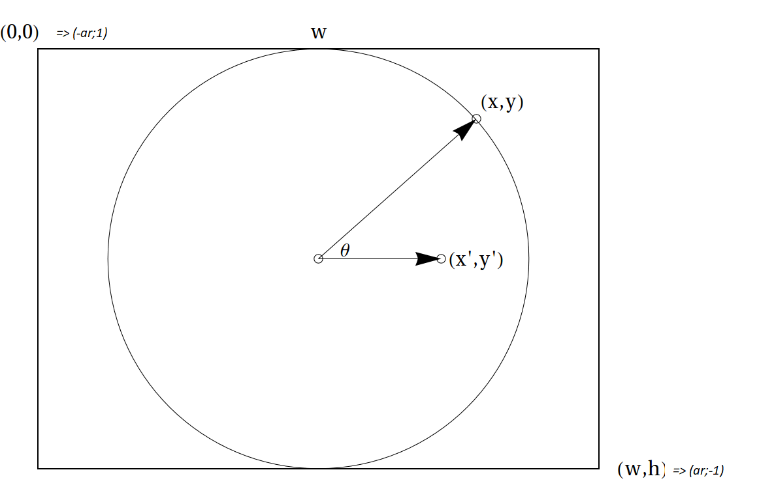
To obtain a rotation in terms of a rotation matrix it is necessary to compute the angle between **a** and **b** vectors:

And the axis of rotation is given by the cross-product of the two vectors:

Note that **a**, **b** vectors in the **LCS** of the sphere. To transform **a**, **b** to the **WCS** it is necessary to take rotation part of *lookAt* function and inverse it. In other words, **a**, **b** is transformed to Camera coordinate system in the **WCS**.

Then final rotation (orbit matrix) is calculated as:

In order to map mouse coordinates x, y to vector on the sphere, it is necessary to convert mouse point [0;0] x [width; height] to [-ar;1] x [ar;-1] range, where ar is the aspect ratio(width/height). This transformation helps to avoid setting the radius of the sphere and assumes that the user makes a rotation around the unit sphere only.



The (x,y) mouse coordinates are used to sample from a z(x,y) function: This means that points inside the circle in figure are mapped on a sphere, while the outer points are mapped in the z=0 plane. This function helps rotating 3D objects when the selection is done inside the circle, while in the borders of the screen the rotation is only around z axis, 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);

Then the algorithm for orbit goes like this:

1. Mouse button pressed (correspond to *IWindow::onMouseInput*):

***a*** *is calculated from the mouse point*

1. Cursor dragged (correspond to *IWindow::onMouseMove*):

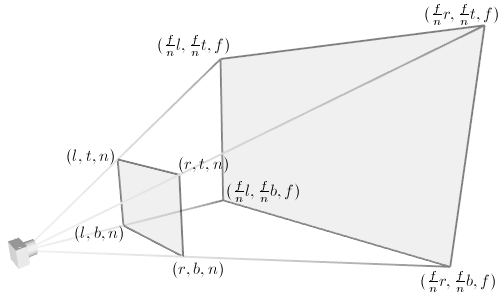
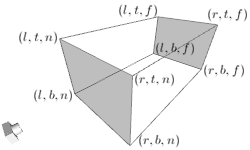
***b*** *is calculated from the mouse point*

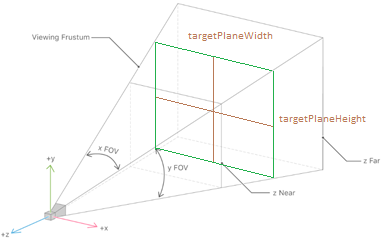
*orbit(****a****,* ***b****)*

***a*** *=* ***b***

## Task 2

GL\_PROJECTION matrix is used to define the frustum. This frustum determines which objects or portions of objects will be clipped out. Also, it determines how the 3D scene is projected onto the screen. OpenGL provides two functions for GL\_PROJECTION transformation. glFrustum is to produce a perspective projection, and glOrtho is to produce an orthographic (parallel) projection. Both functions require six parameters to specify six clipping planes: left, right, bottom, top, near and far planes. Eight vertices of the viewing frustum are shown in the following image.

  
The vertices of the far (back) plane can be simply calculated by the ratio of similar triangles.  
   
For orthographic projection, this ratio will be 1, so the left, right, bottom and top values of the far plane will be same as on the near plane.



You may also use *gluPerspective* with less number of [parameters](#OpenGLProjMatrix): vertical field of view (yFOV), the aspect ratio of width to height and the distances to near and far clipping planes. The equivalent conversion from *gluPerspective* to *glFrustum* is described in the following code:

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

*The construction of* [frustum](#Frustum) *is not the purpose of this lesson, for a more detailed study, see the links at the end of the lab. To build it, use a similar function from the glm.*

Some operations, such as panning, require input parameters within the target plane. The target plane is the plane that looks toward the camera target point and is located at distance from the camera position. To calculate the width and the height of the target plane, it is needed to use formulas that were used to calculate near plane.

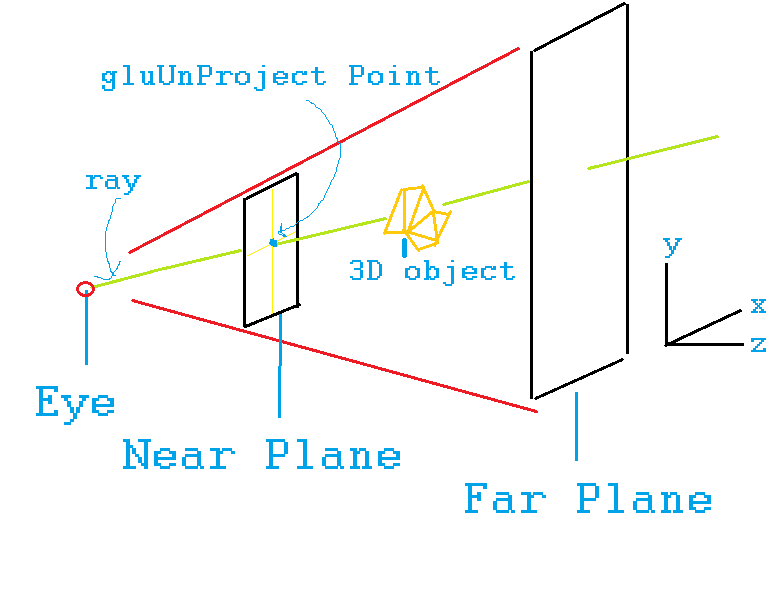
|  |
| --- |
| double Viewport::calcTargetPlaneWidth() const  {  return calcTargetPlaneHeight() \* calcAspectRatio();  }  double Viewport::calcTargetPlaneHeight() const  {  return 2.0 \* camera.distanceFromEyeToTarget() \* tan(radians(FOV / 2.0));  } |

Implement Viewport class using the perspective parameters: FOV, aspectRatio (= width/height), 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  }; | |

The function *calcCursorRay* (green line in the picture below) return a ray (origin, direction) that goes from the near plane to the far plane in the **WCS**. It takes as input mouse position **x**, **y**.

In order to implement *calcCursorRay*, the *unproject* function is used. This function converts the position of the mouse and the given z coordinate (in the clip space [-1;1]) to a point in the world space.



Before going into details with *unproject* function, let’s consider it as an ‘inverse’ function – *project*.

This function takes 3d point (in the **WCS**), world, view and projections matrices as an input and returns point in the screen space [0; width] x [0; height]. Inside it is simply multiples input point by projection\*view\*world matrices. After multiplication result point is in clip space [-1;1]. After that point in the Clip Space is mapped to the screen space [0; width] x [0; height] using simple linear equations.

*unproject* function does the inverse chain of operations. **z** in this case should be set manually. z = -1 corresponds to the near plane, z = 1 corresponds to the far plane.

Now it is very easy to implement *calcCursorRay* on the basis of *unproject* function:

|  |
| --- |
| glm::vec3 a = unproject(x, y, -1.0);  glm::vec3 b = unproject(x, y, 1.0);  ray cursorRay{a, normalize(b - a)}; |

## Task 3

Implement the *moveCamera* function.

|  |  |
| --- | --- |
| 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  }  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([&](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  }  });  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;  } | |

## Exercises

1. Add and implement the method void Viewport::zoomToFit(v3 min, v3 max); Input variables *min, max* describes the bounding box of the entire scene. This function enables you to see your whole scene area (which enclosed in bounding box) on-screen.
2. Implement Panning operation using *Camera::pan* and corresponding mouse events. User should be able to drag camera by holding left mouse button and moving it through the scene. Modify initial code by adding window mouse callbacks
3. Implement Arcball operation using *Camera::orbit* and corresponding mouse events. User should be able to rotate camera by holding right mouse button and moving it through the scene. Modify initial code by adding window mouse callbacks

## Resources and Notes

1. <http://www.songho.ca/opengl/gl_transform.html> - OpenGL transformations

1. <http://www.songho.ca/opengl/gl_projectionmatrix.html> - OpenGL projection matrix

1. <https://en.wikipedia.org/wiki/Viewing_frustum> - Frustum description