# Lab 5

## Overview

In this Lab work it is necessary to add a translation and rotation manipulators. Then, on their basis, **Triad** manipulator is assembled and **TransformMeshOperator** is implemented for a mesh transformation. Also, it is necessary to implement **EditMeshOperator**. A lab work will result:

1. In a Mesh Editor operation where user can select a Node’s Mesh and change its transformation
2. User can select a triangle and move it along its normal

## Objective

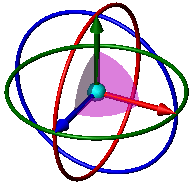
1. To implement **TranslationManipulator**
2. To implement **EditMeshOperator**
3. To implement **RotationManipulator**
4. To implement **Triad** manipulator
5. To implement **TransformMeshOperator**

## Infrastructure

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

## Task 1

Many CAD applications have a **Triad**, which looks like perpendicular to each other three arrows. Such a Triad is usually attached to the geometry and allows a user to move geometry by dragging the arrows. Triad consists of three translation manipulators. In addition to this, Triad can also have perpendicular to each other planes (which are represented in the form of circles), which are responsible for the rotation. Such circles are called rotational manipulators.



Let consider the mechanism of a translation manipulator action. When a user selects an arrow, holds and starts to move along the arrow, then an attached object moves in the given direction. Hence, it is necessary to calculate how far the cursor has moved along the arrow. For this it is needed three states:

1. Push - a user selected an arrow
2. Drag - a user is moving an arrow along its axis
3. Release - a user released an arrow

‘Communication’ with geometry is done via **callback**, to which the **Node** will subscribe and listen to the changes regarding the arrow position. Correspondingly, it is enough to transmit delta (in case of translation manipulator it is a translation matrix, in case of rotational manipulator it is rotation matrix) to the Callback. Delta shows the value that arrow has moved in comparison with the previous frame.

**Manipulator** is a **Node**. It contains three methods: *handleMovement*, *setCallback*, *sendFeedback*.

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

*handleMovement* method is executed when a user has captured the arrow and is carrying out three actions: Push, Drag and Release. *handleMovement* should be subscribed to the **IWindow** events.

*setCallback* method sets callback that will be called when user does Push, Drag or Release.

*sendFeedback* method multiplies **Manipulator’s** transformation matrix by *deltaMatrix* in order to move arrow itself and fires callback with *deltaMatrix*. It is called inside *handleMovement* method.

**TranslationManipulator** is a **Manipulator**. A user sets a position of a manipulator with the help of the transformation matrix. In such a **Node**, there is mesh which contains the arrow geometry in the given direction. Note that arrow direction is set in the **LCS**. Use the *meshArrow* function written in the Lab work 3 (see Exercises section). Transform each vertex of *meshArrow* by rotation matrix that is computed as rotation from (0,0,1) to *dir\_L*.

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

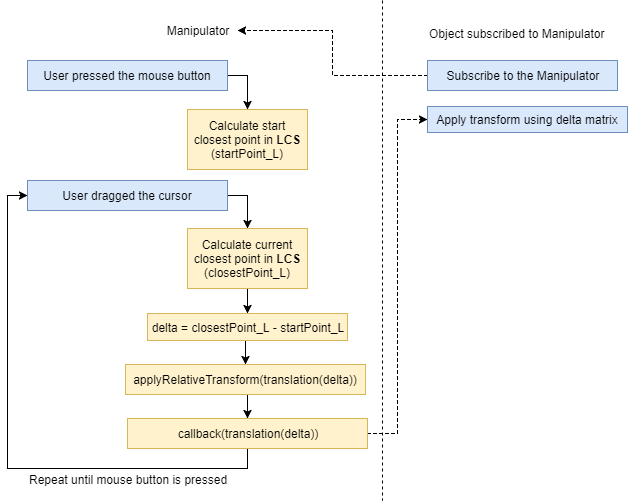
Example usage of translation manipulator:

|  |
| --- |
| // we want to do translation in the local X direction  TranslationManipulator manipulator({1,0,0});  // in the World space it will be in (10,0,0) position  manipulator.setRelativeTransform(translation({ 10,0,0 }));  //objectToMove - is the Node that we want to transform  manipulator.setCallback([&objectToMove](const glm::mat4& delta)  {  objectToMove.applyRelativeTransform(delta); }); |

### handleMovement algorithm

Implementation of handleMovement algorithm is divided into two logical parts:

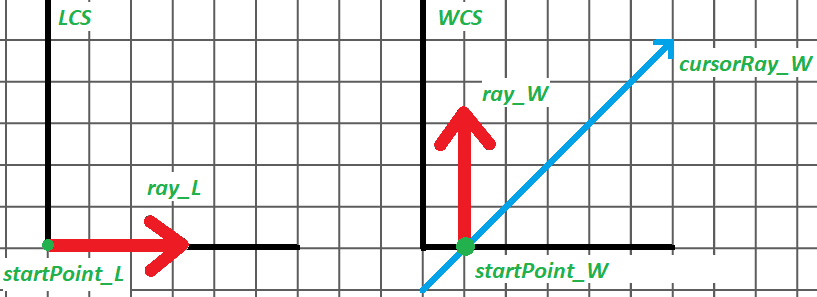
1. User pressed the mouse button – calculate start closest point in **LCS** between cursor’s ray and manipulator’s ray
2. Cursor dragged or released – calculate current closest point in **LCS** between cursor’s ray and manipulator’s ray. Calculate **delta** vector between current closest point and start intersection point. Fire callback with a **delta**. Apply delta to manipulator’s relative matrix to translate the manipulator itself



Manipulator’s ray in **WCS** (*ray\_W*) is obtained by multiplying Manipulator’s **dir\_L** using *Manipulator::calcAbsoluteTransform*.

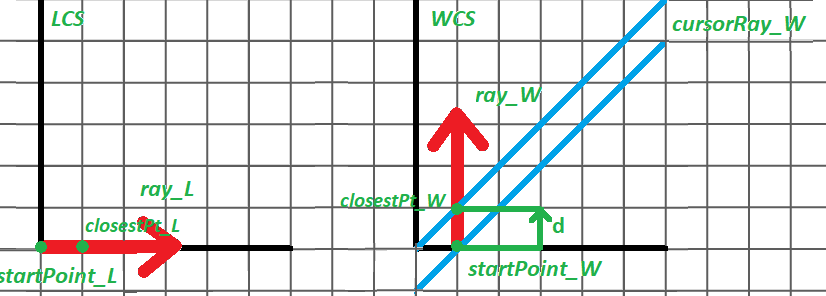
*cursorRay\_W* can be obtained by using *Viewport::calcCursorRay* (see Lab work 2)function.

If *movementType* equals to *MovementType::Push* then compute closest point *startPoint\_W* between *cursorRay\_W* and *ray\_W*. Transform *startPoint\_W* to **LCS** of Manipulator and get *startPoint\_L* as result.



If *movementType* equals to *MovementType::Drag* or *MovementType::Release* then:

1. *c*ompute closest point *closestPt\_W* between *cursorRay\_W* and *ray\_W*. Transform *closestPt\_W* to **LCS** of Manipulator and get *closestPt\_L*.



1. Calculate *delta = translation(closestPt\_L - startPoint\_L)*
2. Call *sendFeedback(translation(delta))*

*sendFeedback* will be rewritten later, when implementing the **Triad**. For now, it is simply multiplies **Manipulator’s** transformation matrix by *deltaMatrix* in order to move arrow itself and fires callback with *deltaMatrix*.

void Manipulator::sendFeedback(const glm::mat4& deltaMatrix)

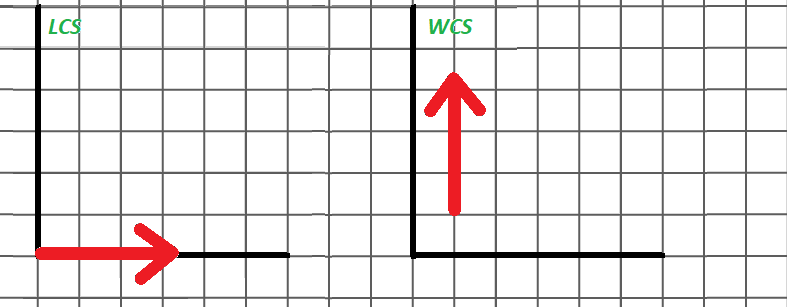
{

Node::applyRelativeTransform(deltaMatrix);

if (callback)

callback(deltaMatrix);

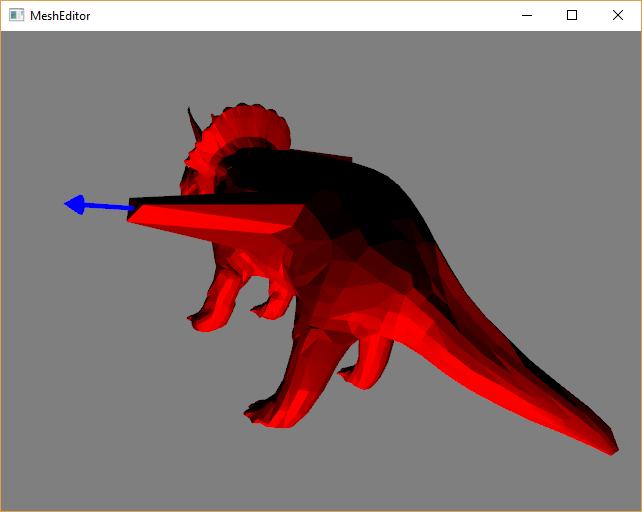
}

**

## Task 2

Implement **EditMeshOperator** with the help of **TranslationManipulator**. Let’s remember the task:

A user presses the **E** key and **EditMeshOperator** turns on. When a user selects a face, there appears a **TranslationManipulator** located in the center of selected triangle with direction that equals to the selected face normal. When a user moves **TranslationManipulator**, so accordingly, the corresponding triangle is moved. A user can select another face, and for it the corresponding **TranslationManipulator** will appear, the old **TranslationManipulator** disappears. When pressing **ESC** - a user offs **EditMeshOperator** and **TranslationManipulator** are removed.



Let’s enumerate all possible states, which **EditMeshOperator** can have, and determine the transitions, where an operator changes the state.

*Idle* state - a user hasn’t selected Manipulator or a Face yet.

*Edit* state - a user has selected a Manipulator and conducts required manipulations (moves arrows of **TranslationManipulator** in the needed way).

### Idle state

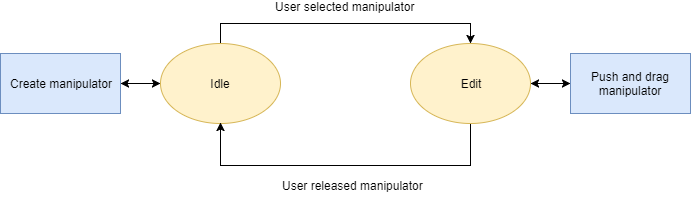
If a user has selected **Node** which is not **Manipulator**, it is necessary to create a **TranslationManipulator** (located in the center) on the selected **Face** and return to the *Idle* state. Otherwise, if a user has selected a **Manipulator**, it is necessary to change state to *Edit* state.

### Edit state

When pushing and dragging it is needed to call *handleMovement* from the captured **TranslationManipulator**. When releasing it is needed to change current state to the *Idle* state. Use corresponding *onMouseMove* and *onMouseInput* events.

In *EditMeshOperator::onEnter* it is necessary to set *Idle* state.

In *EditMeshOperator::onExit* it is necessary to complete the current state and clear all the resources (to delete the attached Translation Manipulator).



Implement the finite state machine shown in the figure above.

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

## Task 3

**RotationManipulator** is also a Tree Node, and uses the same trick to set the geometry in the given direction as the **TranslationManipulator**. Use *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;  }; | |

Implementation of *handleMovement* function repeat the same pattern that was used for **TranslationManipulator**. The only difference is that it is necessary calculate intersection point between **Manipulator’s** ray and plane (which is depicted as torus). Then it is necessary to build vector from origin to intersection point and then calculate rotation between start intersection vector and current intersection vector.

## Task 4

**Triad** is also *tree node* and it is easy to build by simply adding the necessary child nodes to the tree:

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

Now It is necessary to rewrite *sendFeedback* method to support complex manipulators like **Triad**. *sendFeedback* finds the root manipulator and transform its children relative matrices by *deltaMatrix*. In the end it fires callback. This is needed in order to transform child manipulators of the composite manipulator like a **Triad** for example. It is necessary so that if a user has pulled an arrow, so then the remaining arrows of **Triad** could move to the same delta. Otherwise, it would have turned out that the arrow would have come 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);

}

## Task 5

Now let’s consider the way how to implement **TransformMeshOperator** using a **Triad**. Let’s remember the task:

A user presses the **T** key and **TransformMeshOperator** turns on. When a user selects a mesh, there appears a **Triad** located in the center of this Node’s mesh. When a user moves or rotates the **Triad**, so accordingly, the corresponding Node’s mesh moves or rotates. A user can select another **Node**, and for it the corresponding **Triad** will appear, the old **Triad** disappears. When pressing **ESC** - a user offs **TransformMeshOperator** and **Triad** is 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  }; | |

## Exercises

1. Think how to optimize construction of **TranslationManipulator** without modifying result of *meshArrow* function. Hint: **TranslationManipulator** is a Tree Node.
2. Consider cases when the **TranslationManipulator** is parallel to the plane of the screen
3. Develop a mechanism in which it is possible to make a move in such a case. Consider the same case when the **RotationManipulator** is perpendicular to the screen plane
4. Implement the **ScaleManipulator**
5. Add vertex selection and its movement. If a user cursor above the vertex then create translation handle in vertex position with the normal that equal to averaged normal of adjacent faces

## Resources and Notes

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