Einführung in C++ – Übung 9
Testatgruppe A (Isaak)

Rasmus Diederichsen

17. Dezember 2014

Aufgabe 9.1 Exception-Handling

Fehlerabfragen machen meines Erachtens nur beim Aufruf von `Vertex::normalize()` Sinn, und auch da nur beschränkt. Alle Arrayzugriffe durch den `[]`-operator geschehen durch Literale, weshalb hier ebenfalls kein `try`-Block nötig ist.

src/exceptions/BaseException.hpp

```cpp
/**
 * @file BaseException.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 08.12.2014
 * @brief Contains the BaseException class declaration
 */


#ifndef BASEEXCEPTION_H

#define BASEEXCEPTION_H

#include <stdexcept>
#include <string>

namespace asteroids
{
    /**
     * @class BaseException
     * @brief A base class for exceptions
     */
    class BaseException : public std::runtime_error
    {
        protected:
            /**
             * @brief Constructor
             * @param s Message string
             */
            BaseException(std::string s);
    };
} /* namespace asteroids */
```

```

#endif /* end of include guard: BASEEXCEPTION_H */
```

src/exceptions/BaseException.cpp

```
#include "BaseException.hpp"

namespace asteroids
{
   BaseException::BaseException(std::string s) : std::runtime_error
       (s) {}
} /* namespace BaseException */
```

src/exceptions/DivisionByZeroException.hpp

```
/**
 * @file DivisionByZeroException.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 08.12.2014
 * @brief Contains DivisionByZeroException class declaration
 */

#ifndef DIVISIONBYZEROEXCEPTION_H

#define DIVISIONBYZEROEXCEPTION_H

#include "BaseException.hpp"

namespace asteroids
{
   /**
    * @class DivisionByZeroException
    * @brief A class for runtime errors due to division by zero.
    */
   class DivisionByZeroException : public BaseException
   {
      public:
         /**
          * @see BaseException
          */
         DivisionByZeroException(std::string s);
   };
} /* namespace asteroids */


#endif /* end of include guard: DIVISIONBYZEROEXCEPTION_H */
```

src/exceptions/DivisionByZeroException.cpp

```
#include "DivisionByZeroException.hpp"

namespace asteroids
{
   DivisionByZeroException::DivisionByZeroException(std::string s)
       : BaseException(s) {}
} /* namespace asteroids */
```

src/exceptions/OutOfBoundsException.hpp

```
/**
 * @file OutOfBoundsException.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 08.12.2014
 * @brief Contains OutOfBoundsException class declaration
 */

#ifndef OUTOFBOUNDSEXCEPTION_H

#define OUTOFBOUNDSEXCEPTION_H

#include "BaseException.hpp"

namespace asteroids
{
   class OutOfBoundsException : public BaseException
   {
      public:
         /**
          * @see BaseException
          */
         OutOfBoundsException(std::string s);
   };
} /* namespace asteroids */

#endif /* end of include guard: OUTOFBOUNDSEXCEPTION_H */
```

src/exceptions/OutOfBoundsException.cpp

```
#include "OutOfBoundsException.hpp"

namespace asteroids
{
   OutOfBoundsException::OutOfBoundsException(std::string s) :
       BaseException(s) {}
} /* namespace asteroids */
```

src/math/Vertex.cpp

```
#include "Vertex.hpp"
#include <stdio.h>
#include <string>
#include "exceptions/DivisionByZeroException.hpp"
#include "exceptions/OutOfBoundsException.hpp"

namespace asteroids {

Vertex::Vertex()
{
        // Default values
        x = y = z = 0.0;
}


Vertex::Vertex(float _x, float _y, float _z)
{
        // Set the given values
        x = _x;
        y = _y;
        z = _z;
}

void Vertex::normalize()
{
        // Normalize the vector
        float mag2 = x * x + y * y + z * z;
        if (fabs(mag2 - 1.0f) > TOLERANCE)
        {
                float mag = sqrt(mag2);
      // to_string is c++ 11
      if (mag == .0f)
         throw DivisionByZeroException("Vector␣to␣normalise␣has␣0␣length.");
      x /= mag;
      y /= mag;
      z /= mag;
   }
}

Vertex Vertex::operator+(const Vertex vec) const
{
   // Add value to value
   float tx = x + vec.x;
   float ty = y + vec.y;
   float tz = z + vec.z;
   return Vertex(tx, ty, tz);
}

Vertex Vertex::operator-(const Vertex vec) const
{
   // Subtract value from value
   float tx = x - vec.x;
   float ty = y - vec.y;
   float tz = z - vec.z;
   return Vertex(tx, ty, tz);
}


float Vertex::operator[](const int &index) const
{

   // Get the wanted value
   if(index == 0)
   {
      return x;
   }

   if(index == 1)
   {
      return y;
   }

   if(index == 2)
   {
      return z;
   }

   // to_string is c++ 11
   throw OutOfBoundsException("Wrong␣index␣" + std::to_string(index
       ));
}


float& Vertex::operator[](const int &index)
{

   if(index == 0)
   {
      return x;
   }

   if(index == 1)
   {
      return y;
   }

   if(index == 2)
   {
      return z;
   }

   throw OutOfBoundsException("Wrong␣index␣" + std::to_string(index
       ));
}




float Vertex::operator*(const Vertex vec) const
{
   // Calculate the result
   return (x * vec.x + y * vec.y + z * vec.z);
}

Vertex Vertex::operator*(float scale) const
{
   // Calculate the result
   float tx = x * scale;
   float ty = y * scale;
   float tz = z * scale;
   return Vertex(tx, ty, tz);
}

void Vertex::operator+=(Vertex v)
{
   // Add value to value
   x += v.x;
   y += v.y;
   z += v.z;
}
```

```

} // namespace cpp2014
```

src/math/Quaternion.cpp

```
#include "Quaternion.hpp"
#include "exceptions/DivisionByZeroException.hpp"

namespace asteroids
{

   Quaternion::Quaternion()
   {
      // Default Quaternion
      x = 1.0;
      y = 0.0;
      z = 0.0;
      w = 0.0;
   }

   Quaternion::~Quaternion()
   {
      // Do nothing
   }

   Quaternion::Quaternion(Vertex vec, float angle)
   {
      // Calculate the quaternion
      fromAxis(vec, angle);
   }

   Quaternion::Quaternion(float _x, float _y, float _z, float _angle)
   {
      // Set the values
      x = _x;
      y = _y;
      z = _z;
      w = _angle;
   }

   Quaternion::Quaternion(float* vec, float _w)
   {
      // Set the values
      x = vec[0];
      y = vec[1];
      z = vec[2];
      w = _w;
   }

   void Quaternion::fromAxis(Vertex axis, float angle)
   {
      float sinAngle;
      angle *= 0.5f;

      // Create a copy of the given vector and normalize the new vector
      Vertex vn(axis.x, axis.y, axis.z);
      try
      {
         vn.normalize();
      } catch (DivisionByZeroException &divex)
      {
         std::cout << divex.what() << std::endl;
      }

      // Calculate the sinus of the given angle
      sinAngle = sin(angle);

      // Get the quaternion
      x = (vn.x * sinAngle);
      y = (vn.y * sinAngle);
      z = (vn.z * sinAngle);
      w = cos(angle);
   }

   Quaternion Quaternion::getConjugate()
   {
      // Conjugate the given quaternion
      return Quaternion(-x, -y, -z, w);
   }


   Quaternion Quaternion::operator* (const Quaternion rq)
   {
      // Calculate the new quaternion
      return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
            w * rq.y + y * rq.w + z * rq.x - x * rq.z,
            w * rq.z + z * rq.w + x * rq.y - y * rq.x,
            w * rq.w - x * rq.x - y * rq.y - z * rq.z);
   }

   Vertex Quaternion::operator* (Vertex vec)
   {
      // Copy the vector and normalize the new vector
      Vertex vn(vec);
      try
      {
         vn.normalize();
      } catch (DivisionByZeroException &divex)
      {
         std::cout << divex.what() << std::endl;
      }

      // Fill the first quaternion and...
      Quaternion vecQuat, resQuat;
      vecQuat.x = vn.x;
      vecQuat.y = vn.y;
      vecQuat.z = vn.z;
      vecQuat.w = 0.0f;

      // calculate the new quaternion
      resQuat = vecQuat * getConjugate();
      resQuat = *this * resQuat;
      return (Vertex(resQuat.x, resQuat.y, resQuat.z));
   }

}
```

Aufgabe 9.2 Timestamps und Logging

Timestamp

src/time/Timestamp.hpp

```
/**
 * @file Timestamp.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 08.12.2014
 */


#ifndef TIMESTAMP_H

#define TIMESTAMP_H

#include <iostream>
#include <sys/time.h>

namespace asteroids
{
   /**
    * @class Timestamp
    * @brief Represents a point in time.
    */
   class Timestamp
   {
      public:
         /**
          * @brief Constructor. Time is initialised to current system time.
          */
         Timestamp();

         /**
          * @brief Get current system time.
          * @return Current system time in milliseconds (from UNIX epoch)
          */
         unsigned long getCurrentTimeInMs() const;

         /**
          * @brief Get time elapsed since instance creation.
          * @return The time elapsed since instance creation in milliseconds
          * (from UNIX epoch)
          */
         unsigned long getElapsedTimeInMs() const;

         /**
```

```
          * @see Timestamp::getCurrentTimeInMs()
          */
         unsigned long getCurrentTimeInS() const;

         /**
          * @see Timestamp::getElapsedTimeInMs()
          */
         unsigned long getElapsedTimeInS() const;

         /**
          * @brief Reset the timer to current system time.
          */
         void resetTimer();

         /**
          * @brief Get string representation of time elapsed since creation.
          * @return The elapsed time as a string.
          */
         std::string getElapsedTime() const;

         /**
          * @brief Operator to print to a stream.
          */
         friend std::ostream& operator<<(std::ostream& os, const Timestamp& ts); // why is friend necessary?

      private:
         unsigned long m_startTime;
   };
} /* namespace asteroids */

#endif /* end of include guard: TIMESTAMP_H */
```

src/time/Timestamp.cpp

```
#include "Timestamp.hpp"
#include <cstddef>
#include <stdexcept>
#include <sstream>

namespace asteroids
{

   Timestamp::Timestamp()
   {
      resetTimer();
   }
   unsigned long Timestamp::getCurrentTimeInMs() const
   {
      struct timeval tv;
      struct timezone tz;
      if (gettimeofday(&tv, &tz) == -1)
         throw std::runtime_error("Error␣while␣attempting␣to␣get␣system␣time.");
      return 1000 * tv.tv_sec + (unsigned long) (tv.tv_usec / 1000);
   }
   unsigned long Timestamp::getElapsedTimeInMs() const
   {
      return getCurrentTimeInMs() - m_startTime;
   }
   unsigned long Timestamp::getCurrentTimeInS() const
   {
      return getCurrentTimeInMs() / 1000;
   }
   unsigned long Timestamp::getElapsedTimeInS() const
   {
      return getElapsedTimeInMs() / 1000;
   }
   void Timestamp::resetTimer()
   {
      m_startTime = getCurrentTimeInMs();
   }
   std::string Timestamp::getElapsedTime() const
   {
      unsigned long elapsed = getElapsedTimeInMs();
      unsigned long hours = elapsed / (1000 * 60 * 60);
      elapsed -= hours * 1000 * 60 * 60;
      unsigned long minutes = elapsed / (1000 * 60);
      elapsed -= minutes * 1000 * 60;
      unsigned long seconds = elapsed / 1000;
      elapsed -= seconds;
      unsigned long milliseconds = elapsed;
      char buffer[17];
      buffer[16] = '\0'; // necessary?
      sprintf(buffer, "[%02lu:%02lu:%02lu␣-␣%03lu]", hours, minutes
          , seconds, milliseconds);
      return std::string(buffer);
   }
   std::ostream& operator<<(std::ostream& os, const Timestamp& ts)
   {
      os << ts.getElapsedTime();
      return os;
   }
} /* namespace asteroids */
```

Logger

src/logging/Logger.hpp

```
/**
 * @file Logger.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 08.12.2014
 */

#ifndef LOGGER_H

#define LOGGER_H

#include "time/Timestamp.hpp"
#include <ostream>

namespace asteroids
```

```
{
   /**
    * @class Logger
    * @brief Singleton Class to log program events
    */
   class Logger
   {
      public:
         /**
          * @brief Get the singleton instance
          * @return the singleton
          */
         static Logger& instance();

         /**
          * @brief specify destination
          * @param filename File to which log should go.
          */
         void setOutputFile(std::string filename);

         /**
          * @brief reset loggin to stdout
          */
         void setOutputToStdout();

         /**
          * @brief Print log message.
          * @param s Message to log.
          */
         Logger& operator<<(const std::string& s);

         /**
          * @brief Desctructor
          */
         ~Logger();
      private:
         static Logger* logger;
         static Timestamp stamp;
         std::ostream* out;
         Logger();
         Logger(const Logger& l);
         void operator=(const Logger& l);

   };
} /* namespace asteroids */

#endif /* end of include guard: LOGGER_H */
```

src/logging/Logger.cpp

```
#include "Logger.hpp"
#include <iostream>
#include <fstream>
#include <cstddef>
#include <typeinfo>

namespace asteroids
{
   Timestamp Logger::stamp;
   Logger* Logger::logger = NULL;
   Logger::Logger()
   {
      setOutputToStdout();
   }

   Logger& Logger::operator<<(const std::string& s)
   {
      *out << stamp;
      *out << "␣-␣";
      *out << s << std::endl;
      return *this;
   }

   void Logger::setOutputFile(std::string filename)
   {
      out = new std::ofstream(filename.c_str());
   }

   void Logger::setOutputToStdout()
   {
      out = &std::cout;
   }

   Logger& Logger::instance()
   {
      if (logger == NULL)
         logger = new Logger;
      return *logger;
   }

   Logger::~Logger()
   {
      if (typeid(*out) == typeid(std::ofstream))
      {
         ((std::ofstream*)out)->close();
         delete out;
      }
   }
} /* namespace asteroids */
```

Geloggt wurde in der `MainWindow`-Klasse, um die Zeit für das Rendering zu prüfen.

src/view/MainWindow.cpp

```
#include "MainWindow.hpp"
#include "io/TriangleMeshFactory.hpp"
#include "io/Read3DS.hpp"
#include "rendering/TexturedMesh.hpp"
#include "logging/Logger.hpp"

using std::cout;
using std::endl;

namespace asteroids
{

```

```
MainWindow* MainWindow::master = 0;

Camera MainWindow::m_cam;

MainWindow::MainWindow(string filename)
{

        // Save pointer to current instance. Later on we
        // will create a proper singleton here...
        MainWindow::master = this;

    // Init glut main window
    initGlut();
    initGL();

        for(int i = 0; i < 256; i++)
        {
                m_keyStates[i] = false;
                m_specialkeyStates[i] = false;
        }


   /* // Timestamp this point */
   /* Timestamp t; */
   Logger::instance() << "Start␣loading␣mesh.";

        // Create a triangle mesh instance
        m_mesh = TriangleMeshFactory::instance().getMesh(filename);

        // Create a sky box. We assume that a model was loaded
            beforehand
        // to ensure that the base path in texture factory was set
            correctly.
        // If not set it manually before creating the sky box!!
        string names[6];
        names[0] = "box1.ppm";
        names[1] = "box2.jpg";
        names[2] = "box3.ppm";
        names[3] = "box4.jpg";
        names[4] = "box5.ppm";
        names[5] = "box6.jpg";

   Logger::instance() << "Start␣rendering.";

   m_skybox = new Skybox(2048, names);

   Logger::instance() << "Finished␣rendering.";

   /* // print elapsed time */
   /* std::cout << t << std::endl; */

   // Call main loop
   glutMainLoop();
}

MainWindow::~MainWindow()
{
   // Check if objects exists an delete if necessary
   if(m_mesh)
   {
      delete m_mesh;
      m_mesh = 0;
   }

   if(m_skybox)
   {
      delete m_skybox;
      m_skybox = 0;
   }
}


void MainWindow::initGlut()
{
   // Create dummy arguments for command line options
   int dummy_argc = 1;
   char *dummy_argv[1];
   dummy_argv[0] = new char[255];
   dummy_argv[0] = (char*)"Main␣Window";

   // Initialize glut toolkit
   glutInit(&dummy_argc, dummy_argv);

   glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
   glutInitWindowPosition(100,100);
   glutInitWindowSize(762, 576);
   glutCreateWindow("3D-Viewer");
   glutSetKeyRepeat(1);

   //Register callback functions
   glutDisplayFunc(MainWindow::callback_render);
   glutReshapeFunc(MainWindow::callback_reshape);
   glutMouseFunc(MainWindow::callback_mouse);
   glutMotionFunc(MainWindow::callback_motion);
   glutKeyboardFunc(MainWindow::callback_key);
   glutKeyboardUpFunc(MainWindow::callback_keyUp);
   glutSpecialFunc(MainWindow::callback_specialkey);
   glutSpecialUpFunc(MainWindow::callback_specialkeyUp);
   glutTimerFunc(15, MainWindow::callback_timer, 0);
   // glutIdleFunc(MainWindow::callback_render);

   // Init OpenGL stuff
   initGL();
}

void MainWindow::initGL()
{
   glMatrixMode(GL_MODELVIEW);
   glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);

   // Setup two light sources
   float light0_position[4];
   float light0_ambient[4];
   float light0_diffuse[4];

   float light1_position[4];
   float light1_ambient[4];
   float light1_diffuse[4];

   light0_position[0] =  1.0f; light0_ambient[0] = 0.8f;
       light0_diffuse[0] = 0.8f;
   light0_position[1] =  1.0f; light0_ambient[1] = 0.8f;
       light0_diffuse[1] = 0.8f;
   light0_position[2] =  0.0f; light0_ambient[2] = 0.8f;
       light0_diffuse[2] = 0.8f;
   light0_position[3] =  1.0f; light0_ambient[3] = 0.1f;
       light0_diffuse[3] = 1.0f;

   light1_position[0] =  0.0f; light1_ambient[0] = 0.1f;
       light1_diffuse[0] = 0.5f;
   light1_position[1] = -1.0f; light1_ambient[1] = 0.1f;
       light1_diffuse[1] = 0.5f;
   light1_position[2] =  0.0f; light1_ambient[2] = 0.1f;
       light1_diffuse[2] = 0.5f;
   light1_position[3] =  1.0f; light1_ambient[3] = 1.0f;
       light1_diffuse[3] = 1.0f;

   // Light 1
   glLightfv(GL_LIGHT0, GL_AMBIENT,  light0_ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE,  light0_diffuse);
   glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
   glEnable(GL_LIGHT0);

   // Light 2
   glLightfv(GL_LIGHT1, GL_AMBIENT,  light1_ambient);
   glLightfv(GL_LIGHT1, GL_DIFFUSE,  light1_diffuse);
   glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
   glEnable(GL_LIGHT1);

   // Enable lighting
   glEnable(GL_LIGHTING);

   // Enable z buffer and gouroud shading
   glEnable(GL_DEPTH_TEST);
   glDepthFunc(GL_LESS);
   glShadeModel (GL_SMOOTH);
}


void MainWindow::callback_key(unsigned char key, int x, int y)
{
   // Re-route key callback
   MainWindow::master->keyPressed(key, x, y);
   glutPostRedisplay();
}

void MainWindow::callback_keyUp(unsigned char key, int x, int y)
{
   // Re-route key callback
   MainWindow::master->keyUp(key, x, y);
   glutPostRedisplay();
```

```cpp
}

void MainWindow::callback_specialkey(int key, int x, int y)
{
   // Re-route special key callback
   MainWindow::master->specialkeyPressed(key, x, y);
   glutPostRedisplay();
}

void MainWindow::callback_specialkeyUp(int key, int x, int y)
{
   // Re-route special key callback
   MainWindow::master->specialkeyUp(key, x, y);
   glutPostRedisplay();
}

void MainWindow::callback_reshape(int w, int h)
{
   // Re-route reshape callback
   MainWindow::master->reshape(w, h);
}

void MainWindow::callback_render()
{
   // Re-route render callback
   MainWindow::master->render();
}

void MainWindow::callback_mouse(int button, int state, int x, int y
    )
{
   // Re-route mouse callback
   MainWindow::master->mousePressed(button, state, x, y);
   glutPostRedisplay();
}

void MainWindow::callback_motion(int x, int y)
{
   // Re-route motion callback
   MainWindow::master->mouseMoved(x, y);
   glutPostRedisplay();
}

void MainWindow::callback_timer (int value)
{
   glutPostRedisplay();
   glutTimerFunc(15, callback_timer, 0);
}

void MainWindow::keyPressed(unsigned char key, int x, int y)
{
   // State of key is pressed
   m_keyStates[key] = true;
}

void MainWindow::keyUp (unsigned char key, int x, int y)
{
   // State of key is unpressed
   m_keyStates[key] = false;
}

void MainWindow::specialkeyPressed(int key, int x, int y)
{
   // State of key is pressed
   m_specialkeyStates[key] = true;
}

void MainWindow::specialkeyUp(int key, int x, int y)
{
   // State of key is unpressed
   m_specialkeyStates[key] = false;
}

void MainWindow::keyOperations(void)
{
   // Controller for moving and rotation
   if (m_keyStates[’q’])
   {
      m_mesh->rotate(ROLL, 0.05);
   }

   if (m_keyStates[’e’])
   {
      m_mesh->rotate(ROLL, -0.05);
   }

   if (m_keyStates[’a’])
   {
      m_mesh->rotate(YAW, 0.05);
   }

   if (m_keyStates[’d’])
   {
      m_mesh->rotate(YAW, -0.05);
   }

   if (m_keyStates[’w’])
   {
      m_mesh->rotate(PITCH, 0.05);
   }

   if (m_keyStates[’s’])
   {
      m_mesh->rotate(PITCH, -0.05);
   }

   if (m_specialkeyStates[GLUT_KEY_UP])
   {
      m_mesh->move(STRAFE, -10);
   }

   if (m_specialkeyStates[GLUT_KEY_DOWN])
   {
      m_mesh->move(STRAFE, 10);
   }

   if (m_specialkeyStates[GLUT_KEY_LEFT])
   {
      m_mesh->move(LIFT, 5);
   }

   if (m_specialkeyStates[GLUT_KEY_RIGHT])
   {
      m_mesh->move(LIFT, -5);
   }

   if (m_specialkeyStates[GLUT_KEY_PAGE_UP])
   {
      m_mesh->move(ACCEL, 5);
   }

   if (m_specialkeyStates[GLUT_KEY_PAGE_DOWN])
   {
      m_mesh->move(ACCEL, -5);
   }
}

void MainWindow::mousePressed(int button, int state, int x, int y)
{
   // Save state and button id. We need this information
   // when the mouse is moved to call the proper reaction
   // callback of the camera
   m_button      = button;
   m_buttonState = state;

}

void MainWindow::mouseMoved(int x, int y)
{

   // Get number the number of pixel between the last
   // und current mouse position
   int dx = x - m_mouseX;
   int dy = y - m_mouseY;

   // Check which button was pressend and apply action
   if(m_button == GLUT_LEFT_BUTTON)
   {
      moveXY(dx, dy);
   }

   if(m_button == GLUT_RIGHT_BUTTON)
   {
      moveHead(dx, dy);
   }

   if(m_button == GLUT_MIDDLE_BUTTON)
   {
      moveZ(dy);
   }

   // Transform viewport
   m_cam.apply();

   // Save new coodinates
   m_mouseX = x;
   m_mouseY = y;

}

void MainWindow::moveXY(int dx, int dy)
{

   if(fabs(dx) > MOUSE_SENSITY)
   {
      if(dx > 0)
      {
         m_cam.turnRight();
      }

      else
      {
         m_cam.turnLeft();
      }

   }

   if(fabs(dy) > MOUSE_SENSITY)
   {
      if(dy > 0)
      {
         m_cam.moveBackward();
      }

      else
      {
         m_cam.moveForward();
      }
   }
}

void MainWindow::moveZ(int dy)
{

   if(fabs(dy) > MOUSE_SENSITY)
   {
      if(dy > 0)
      {
         m_cam.moveUp();
      }

      else
      {
         m_cam.moveDown();
      }
   }
}

```

```
void MainWindow::moveHead(int dx, int dy)
{

   if(fabs(dy) > MOUSE_SENSITY)
   {
      if(dy > 0)
      {
         m_cam.turnUp();
      }

      else
      {
         m_cam.turnDown();
      }
   }

   if(fabs(dx) > MOUSE_SENSITY)
   {
      if(dx > 0)
      {
         m_cam.turnRight();
      }

      else
      {
         m_cam.turnLeft();
      }
   }
}


void MainWindow::reshape(int w, int h)
{
   if(h == 0)
   {
      h = 1;
   }

   float ratio = 1.0* w / h;

   // Reset the coordinate system before modifying
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   // Set the viewport to be the entire window
   glViewport(0, 0, w, h);

   // Set the correct perspective.
   gluPerspective(45,ratio,1,100000);
   glMatrixMode(GL_MODELVIEW);

   // Set 'LookAt'
   m_cam.apply();
}

void MainWindow::render()
{
   MainWindow::keyOperations();
   // Set black background color
   glClearColor(0.0, 0.0, 0.0, 0.0);

   // Clear bg color and enable depth test (z-Buffer)
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // Render objects
   if(m_skybox)
   {
      m_skybox->render();
   }

   if(m_mesh)
   {
      m_mesh->render();
   }



   glFinish();

   // Call back buffer
   glutSwapBuffers();
}

}
```