Einführung in C++ – Übung 8
Testatgruppe A (Isaak)

Rasmus Diederichsen

4. Dezember 2014

Aufgabe 8.1 Auslagern von Funktionalität in Oberklassen

Die `Renderable`-Klasse kann abstrakt sein, da sie nur eine Methode enthält, die Subklassen überschreiben müssen, damit etwas passiert.

Listing 1: Renderable.hpp

```cpp
/**
 * @file Renderable.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 04.12.2014
 */

#ifndef RENDERABLE_KG6LA00J

#define RENDERABLE_KG6LA00J

namespace asteroids {

   /**
    * @class Renderable
    * @brief Abstract class to represent something which can be
       rendered on screen.
    */
   class Renderable {

      /**
       * @brief Render this object (pure virtual)
       */
      virtual void render () = 0;
   };
}

#endif /* end of include guard: RENDERABLE_KG6LA00J */
```

Das gleiche gilt für `Transformable` , jede Subklasse ist für ihre Bewegung selbst verantwortlich. *Hinweis:* Die Methoden dieser Klasse sind in `TriangleMesh.cpp` implementiert, diese fehlten initial. Auch wurde in allen Klassen die Acquise des Singletons auf Pointer umgestellt.

Listing 2: Transformable.hpp

```
#ifndef TRANSFORMABLE_DILCYMZ4

#define TRANSFORMABLE_DILCYMZ4

#include "Renderable.hpp"
#ifdef __APPLE__
#include <OpenGL/gl.h>
#else
#include <GL/gl.h>
#endif

namespace asteroids

{
   /**
    * @class Transformable
    * @brief Abstract base class for <tt>Mesh</tt>es which can be
      moved &
    * rotated.
    */
   class Transformable : public Renderable
   {
      public:
         /**
          * @brief   Rotate the Transformable
          * @param axis axis of rotation
          * @param speed speed of rotation
          */
         virtual void rotate(int axis, float speed) = 0;

         /**
          * @brief   Move the Transformable along acis
          * @param axis axis to move along
          * @param speed speed of movement
          */
         virtual void move(int axis, float speed) = 0;

      protected:
         /**
          * @brief Array containing the transformation matrix.
          */
         float m_transformation[16];

         /**
          * @brief Method to compute the transformation matrix.
          */
         virtual void computeMatrix() = 0;
   };
}

#endif /* end of include guard: TRANSFORMABLE_DILCYMZ4 */
```

Das `FixedObject` ist leer, also nur ein Markerinterface.

Listing 3: FixedObject.hpp

```cpp
#ifndef FIXEDOBJECT_4ST6ZIUC

#define FIXEDOBJECT_4ST6ZIUC

#include "Renderable.hpp"

namespace asteroids {

   /**
    * @class FixedObject
    * @brief Represents something which cannot be moved or rotated.
    */
   class FixedObject : public Renderable { };
}

#endif /* end of include guard: FIXEDOBJECT_4ST6ZIUC */
```

Aufgabe 8.2 Erweiterungen für neue Dateiformate

Listing 4: TriangleMeshFactory.hpp

```cpp
/**
 * @brief Contains Factory for mesh generation.
 *
 * @file TriangleMeshFactory.hpp
 * @author rdiederichse@uos.de
 */
#ifndef TRIANGLEMESHFACTORY_H

#define TRIANGLEMESHFACTORY_H

#include "rendering/TriangleMesh.hpp"

namespace asteroids
{
   /**
    * @class TriangleMeshFactory
    * @brief Singleton Factory class to encapsulate parsing meshes
        from different file
    * types.
    */
   class TriangleMeshFactory
   {
      public:
         /**
          * @brief Returns a pointer to a mesh parsed from a given
              file. Format
          * is recognized by extension.
          *
          * @param filename The file containing the mesh definition
              .
          * @return A pointer to the parsed mesh
          */
```

```cpp
            TriangleMesh* getMesh(const std::string &filename) const;

            /**
             * @brief Method to acquire the singleton instance
             * @return The singleton.
             */
            static TriangleMeshFactory* instance();

        private:
            /**
             * Empty default constructor. Does nothing.
             */
            TriangleMeshFactory();

            /**
             * The singleton instance.
             */
            static TriangleMeshFactory* instance_ptr;

            /**
             * Copy constructor. Does nothing.
             */
            TriangleMeshFactory(const TriangleMeshFactory& f) {};

            /**
             * Assignment operator. Does nothing.
             */
            TriangleMeshFactory& operator=(const TriangleMesh& f) {};
    };
} /* namespace asteroids */
#endif /* end of include guard: TRIANGLEMESHFACTORY_H */
```

Listing 5: TriangleMeshFactory.cpp

```cpp
/**
 * @brief TriangleMeshFactory implementation.
 * @file TriangleMeshFactory.cpp
 * @author rdiederichse@uos.de
 */
#include "io/TriangleMeshFactory.hpp"
#include "ReadPLY.hpp"
#include "Read3DS.hpp"
#include "MeshReader.hpp"

using std::string;

namespace asteroids
{

    TriangleMeshFactory::TriangleMeshFactory() {}
    TriangleMeshFactory* TriangleMeshFactory::instance_ptr = NULL;

    TriangleMeshFactory* TriangleMeshFactory::instance()
    {
        if (instance_ptr != NULL) return instance_ptr;
        else return (instance_ptr = new TriangleMeshFactory);
    }

```

```
   TriangleMesh* TriangleMeshFactory::getMesh(const string & filename) const
   {
      unsigned found = filename.find_last_of("/\\");
      string basePath = filename.substr(0, found+1);
      TextureFactory::setBasePath(basePath);

      int pos;
      MeshReader *reader;
      if ((pos = filename.find(".ply")) == (filename.length() - 4)) // .ply extension
      {
         reader = new ReadPLY(filename);
         return reader->getMesh();
      } else if ((pos = filename.find(".3ds")) == (filename.length() - 4)) // .3ds extension
      {
         reader = new Read3DS(filename);
         return reader->getMesh();
      } else  // error
      {
         int i = filename.find_last_of(".");
         std::cerr << "TriangleMeshFactory␣Error:␣File␣" << filename << (i == string::npos ? "␣without␣extension" : "␣of␣type␣" + filename.substr(i))
            << "␣not␣readable.␣" << std::endl;
         return NULL;
      }
   }

} /* namespace asteroids */
```

Listing 6: TextureFactory.hpp

```
#ifndef TEXTUREFACTORY_H

#define TEXTUREFACTORY_H

#include "rendering/Texture.hpp"

namespace asteroids
{
   class TextureFactory
   {
      public:
         /**
          * @brief Returns a pointer to a textured mesh parsed from a given file. Format
          * is recognized by extension.
          *
          * @param filename The file containing the mesh definition.
          * @return A pointer to the parsed mesh
          */
         Texture* getTexture(const std::string &filename) const;

         /**
          * @brief Method to acquire the singleton instance
```

```
             * @return The singleton.
             */
            static TextureFactory* instance();

            /**
             * Set the base path relative to which textures will be
                loaded.
             * @param basepath The path relative to which textures are
                 loaded.
             */
            static void setBasePath(std::string basepath);

        private:
            /**
             * Empty default constructor. Does nothing.
             */
            TextureFactory();

            /**
             * The singleton instance.
             */
            static TextureFactory* instance_ptr;

            /**
             * Copy constructor. Does nothing.
             */
            TextureFactory(const TextureFactory& f) {};

            /**
             * Assignment operator. Does nothing.
             */
            TextureFactory& operator=(const TextureFactory& f) {};

            /**
             * @brief The base path.
             */
            static std::string basepath;
    };
} /* namespace asteroids */

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

Listing 7: TextureFactory.cpp

```
/**
 * @file TextureFactory.cpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 03.12.2014
 */

#include "TextureFactory.hpp"
#include "BitmapReader.hpp"
#include "ReadPPM.hpp"
#include "ReadJPG.hpp"
#include "ReadTGA.hpp"
#include <iostream>
#include <cstddef>

```

```cpp
namespace asteroids
{

   TextureFactory::TextureFactory() {}
   TextureFactory* TextureFactory::instance_ptr = NULL;
   std::string TextureFactory::basepath = "";

   TextureFactory* TextureFactory::instance()
   {
      if (instance_ptr != NULL) return instance_ptr;
      else return (instance_ptr = new TextureFactory);
   }

   Texture* TextureFactory::getTexture(const string &filename) const
   {
      int pos;
      BitmapReader *reader;
      if ((pos = filename.find(".ppm")) == (filename.length() - 4)) // .ppm extension
      {
         reader = new ReadPPM(basepath + filename);
         return new Texture(reader->getPixels(), reader->getWidth(), reader->getHeight());
      } else if ((pos = filename.find(".jpg")) == (filename.length() - 4)) // .jpg extension
      {
         reader = new ReadJPG(basepath + filename);
         return new Texture(reader->getPixels(), reader->getWidth(), reader->getHeight());
      } else if ((pos = filename.find(".tga")) == (filename.length() - 4))
      {
         reader = new ReadTGA(basepath + filename);
         return new Texture(reader->getPixels(), reader->getWidth(), reader->getHeight());
      } else  // error
      {
         int i = filename.find_last_of(".");
         std::cerr << "TextureFactory␣Error:␣File␣" << filename
            << (i == string::npos ? "␣without␣extension" : "␣of␣type␣" + filename.substr(i))
            << "␣not␣readable.␣" << std::endl;
         return NULL;
      }
   }

   void TextureFactory::setBasePath(string basepath)
   {
      TextureFactory::basepath = basepath;
   }


} /* namespace asteroids */
```

Listing 8: ReadPPM.hpp

```cpp
/**
```

```
 * @file ReadPPM.hpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 02.12.2014
 */

#ifndef READPPM_H

#define READPPM_H

#include "BitmapReader.hpp"
#include <fstream>

using std::ifstream;

namespace asteroids
{
   /**
    * @enum PPMTYPE
    * @brief Constants for PPM types.
    */
   enum PPMTYPE {
      P3, ///< Ascii file
      P6,  ///< Binary file
      UNDEF ///< Unknown type (should lead to error)
   };

   /**
    * @class ReadPPM
    * @brief A reader for ppm images.
    */
   class ReadPPM : public BitmapReader
   {
      private:
         /**
          * @brief Reads linewise from a ppm file as long as the
             current line starts
          * with a '#'.
          *
          * After the method has run, the stream's current line
             will
          * be the first non-comment line after the position for
             which the
          * method was called.
          * @param stream The input file stream to read from.
          * @param buffer Char buffer to place the bytes or chars
             in.
          * @param bufsize The length of the buffer.
          */
         void readFirstNonCommentLine(ifstream& stream, char*
            buffer, const int bufsize);

         /**
          * @brief Parses a ppm header. Sets the members <tt>
             m_width</tt>,
          * <tt>m_height</tt> and reads the magic number.
          * @param filename The name of the file.
          * @return The type of the file (binary or ascii). One of
```

```
            * <tt>PPMTYPE</tt>
            * @see ReadPPM::PPMTYPE
            */
           PPMTYPE readHeader(const std::string& filename);

           /**
            * The position right after the header. Seeking to this
                position and
            * then reading will read everything except the header.
            */
           ifstream::pos_type end_header;

           /**
            * @brief Read an ascii ppm file.
            * @param filename The name of the file.
            */
           void readP3(const std::string& filename);

           /**
            * @brief Read a binary ppm file.
            * @param filename The name of the file.
            */
           void readP6(const std::string& filename);
        public:

           /**
            * @brief Construct a PPM reader to read from a given file
                .
            * @param filename The name of the file.
            */
           ReadPPM(const std::string& filename);

           /**
            * @brief Empty destructor. Deallocation of the image
                buffer must be
            * taken are of by the client.
            */
           ~ReadPPM();
    };
} /* namespace asteroids */

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

Listing 9: ReadPPM.cpp

```
/**
 * @file ReadPPM.cpp
 * @author Rasmus Diederichsen (rdiederichse@uos.de)
 * @version 02.12.2014
 */

#include "ReadPPM.hpp"
#include <sstream>
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;
using std::string;
using std::stringstream;
using std::ifstream;
using std::ios;

namespace asteroids
{
   ReadPPM::ReadPPM(const std::string& filename)
   {
      cout << "ReadPPM:␣reading␣ppm␣file␣" << filename << endl;
      m_pixels = NULL; // in case shit hits the fan.
      m_height = m_width = 0;
      PPMTYPE type = readHeader(filename); // parse header and recognise file type
      // allocate only if needed.
      if (type != UNDEF) m_pixels = new unsigned char[m_width * m_height * 3];

      switch (type)
      {
         case P3:
            readP3(filename);
            break;
         case P6:
            readP6(filename);
            break;
         default:
            cerr << "ReadPPM:␣Error.␣Unknown␣PPM␣format." << endl;
            break;
      }
   }

   void ReadPPM::readFirstNonCommentLine(ifstream& stream, char* buffer, const int bufsize)
   {
      do { // read at least one line
         stream.getline(buffer, bufsize);
      } while(stream.good() && (buffer[0] == '#')); // continue while comment
      // now we have read the first non-comment line into the buffer.
   }

   PPMTYPE ReadPPM::readHeader(const string& filename)
   {
      ifstream file(filename.c_str());
      if (file.good())
      {
         /* ATTENTION. THIS ERRONEOUSLY ASSUMES THAT EVERYTHING IS SEPARATED BY
            NEWLINES, EXCEPT THE DIMENSIONS. NEED TO FIX THIS. */
         const int bufsize = 70; // Acc. to spec, lines should not be longer
         char buffer[bufsize]; // buffer to hold the line
         readFirstNonCommentLine(file, buffer, bufsize); // fill it
         string magic_number(buffer); // first line must contain magic number
         readFirstNonCommentLine(file, buffer, bufsize);
         string line(buffer); // next one must contain dimensions
         stringstream ss(line); ss >> m_width >> m_height; // get them out
         readFirstNonCommentLine(file, buffer, bufsize);
         file.getline(buffer, bufsize); // skip the max color value. Assume it's 255..
         end_header = file.tellg();
         file.close();

         cout << "ReadPPM:␣Parsed␣header␣of␣" << filename << ".␣width=" << m_width
            <<"␣height=" << m_height << "␣magic␣number␣is␣"
            << magic_number  << endl;

         if (magic_number == "P3") return P3;
         else if (magic_number == "P6") return P6;
      }  return UNDEF;
   }

   void ReadPPM::readP3(const string& filename)
   {
      cout << "ReadPPM:␣Reading␣ascii␣file␣" << filename << endl;
      ifstream file(filename.c_str());
      if (file.good())
      {
         file.seekg(end_header); // go back to where we left off
         // the stream splits on whitespace so the chars can be read one after
         // the other without hassle.
         for(int i = 0; i < m_width * m_height * 3; i++)
         {
            int c;
            file >> c; // we need to map an ascii number symbol (or more) to the
                       // numerical value, not just use the symbols itself
            m_pixels[i] = (char)c;
         }
         file.close();
      } else { cerr << "ReadPPM:␣Error,␣P3␣file␣not␣good." << endl; } // TODO: Detailed error
   }

   void ReadPPM::readP6(const string& filename)
   {
      cout << "ReadPPM:␣Reading␣binary␣file␣" << filename << endl;
      ifstream file(filename.c_str(), ios::binary); // open in binary mode
      if (file.good())
      {
         file.seekg(end_header); // go back to where we left off
         file.read((char *)m_pixels,m_width * m_height * 3); // not sure if good. dump whole binary blob into array.
         file.close();
      } else { cerr << "ReadPPM:␣Error,␣P6␣file␣not␣good." << endl; } // TODO: Detailed error
```

```
    }

    ReadPPM::~ReadPPM() { }

} /* namespace asteroids */
```