

2. Assignment: Modeller

Markus Höferlin

Gregor Mückl

Institut für Visualisierung und Interaktive Systeme

1 Topics

- Basics of a C++-Application with GUI
- Multi-View Application
- Basics of Scene Graph Design
- Hierarchical Modelling
- OpenGL Picking
- OpenGL Lighting
- Per-Pixel Lighting with GLSL

2 Introduction

In this assignment you have to extend your GUI application from the previous assignment to a simple modeller, which has to be able to assemble 3D scenes from simple geometric primitives. The scene itself is described by a *scene graph*. According menu items allow the insertion of new objects into the initially empty scene. Using mouse *picking* you have to be able to select individual objects to reposition them in the scene or to alter their shapes. Further, deleting of objects has to be implemented and you have to write a per pixel lighting GLSL shader as well. Additionally the mouse has to be used to modify camera parameters.

The modeller has to support different modes of operation. Next to the window-filling view of the scene, the drawing area can be divided into two or four child windows to provide multiple views of the same scene. Toggling between the different modes of operation is done with command line switches on program start. Without any parameter supplied the modeller has to start in the single-view operation mode. The command line switches `-2` or `-4` toggle a double-view or a quad-view operation mode, respectively. If you like to test the final program, change into the directory `/proj/fapra/examples` and execute the program `Modeller` on 32 bit or `Modeller-x64` on 64 bit machines.

As we will use GLSL shaders in the last exercise, you need a computer with OpenGL 2.0 support. Executing `glxinfo | grep -i "opengl version"` on your shell will show you the OpenGL version the driver supports.

Before you start with this assignment you have to set the QT environment variable once again. This is done by typing `setenv QTDIR /usr/lib/qt3 (tsh)`, or `export QTDIR=/usr/lib/qt3 (bash)`, respectively.

Introducing with this assignment, you have to use the documentation tool *Doxygen*. For this purpose comments have to be headed by `//!` or have to be enclosed by `/*! ... */`, otherwise they are not included in the final documentation. Additional information can be found in the Doxygen documentation (link provided on the Fapra Web site).

3 First Steps

This document leads you through the individual tasks to complete the assignment step-by-step. The source skeleton can be obtained from our SVN-Server like the previous assignment. The full path is <https://svn.vis.uni-stuttgart.de/FaPras/Graphik/WS2009/<username>/2>.

The following table gives an overview on the new files and classes:

File	Class	Description
View.C csg.C	View csg*	view of the scene scene graph node (data structure for scene representation)
MaterialEd.qt.C	material editor	dialog for object properties
m4x4.H	vector4, matrix4	mathematical auxiliary classes
createJeep.txt	-	implementation of the function <i>createJeep</i>

The version you just checked-out from the SVN should contain a working program which behaves almost like HelloCube³. It doesn't matter if you didn't complete the previous assignment. All files mandatory for solving this assignment are present. However, if you like, you can use your HelloCube sources as starting point. Just copy `AppWin.qt`, `OpenGLFrame.qt`, `OptionsDlg.qt`, and the two shaders into the modeller directory. As first step you have to implement some changes and extensions to your code.

1st Task: Extending the Viewer

For starters you have to modify the "Edit" menu. Combine the menu items `Translate/Rotate/Scale Cube` to a common sub-item `Adjust Objects`. Rename the menu item `Reset Cube` to `Reset Object` and add further menu items to your "Edit" menu to have the items listed in the table below. Please adhere to the layout (order of items and separators) as shown.

Adjust Camera
Adjust Objects
Create Cube
Create Sphere
Create Cylinder
Create Jeep
Object Properties
Delete Object
Reset Object
Reset Camera

Table 1: "Edit" menu

To each menu item assign an appropriate keyboard shortcut. Don't forget to modify the according slots or to connect new slots with newly created menu items, respectively. The body for the new slots does not have to be implemented yet.

Assign an icon to each `Create . . .` menu item. The necessary icons can be found in the subdirectory `icon`.

4 Multi-View Application

The modeller being implemented in this assignment has to support different modes of operation. Next to the single view a double and quad view split operation mode provide multiple views of the same scene by essentially splitting the drawing area of your application's window.

For the implementation of this functionality you *have to* use the *Model-View-Controller (MVC)* paradigm. The core of this framework is the total separation of user interaction (controller), data (model) and its graphical representation (view). The view manages the graphical or textual output, in our case the OpenGL rendering. The controller interprets the mouse and keyboard inputs and issues the according changes of the graphical representation and the data. Finally, the model describes the data, in our case the rendered scene with objects and their properties, namely position, size, orientation, and color.

This separation allows programming of complex and flexible applications. A lot of applications with graphical user interfaces make use of this paradigm, e.g., Emacs. It is able to display the content of a single file in multiple windows simultaneously to alleviate the tedium of editing different parts of a large single file multiple strides apart. Certain CAD applications support two screens, whereas on the one screen the graphical representation of the model is rendered while the other screen provides textual input/output.

In our Modeller the MVC paradigm is used to allow multiple views of the same scene. In that way the scene can be rendered in three orthogonal views and additionally be viewed by a freely configurable camera. The class `OpenGLFrame`, allocating the OpenGL window and accepting user input, serves as Controller. In order to store the scene we use a so called scene graph which will be introduced later. The `View` object (`View.C`) is responsible for the graphical representation.

2nd Task: Multi View Application

Now you have to extend your viewer to a multi view application. Modify the constructor of the class `OpenGLFrame` for it to allocate a certain amount of `View` objects. The amount of view objects has to be defined on the command line, which has to be processed by the constructor of the class `AppWin`. For default only one view is created. The command line switch `-2` triggers the creation of two views next to each other and `-4` forces the creation of four views arranged quadratically. Needless to say, this information has to be made available displaying a *help screen* using the command line switch `-h`.

The size and position of the views have to be specified on their creation. Choose appropriate values and keep in mind to adjust their size and position after (re-)scaling the application window.

Hint: The X-Window system and OpenGL use different coordinate systems! A `View` operates in OpenGL viewport coordinates exclusively.

To support multiple views you have to adapt the method `OpenGLFrame::paintGL`. The function should only deal with display settings like filled or wire-frame mode from now on. Draw calls have to be routed to the according views. The method `View::paintGL` has to implement the settings specific to the view like camera set up. Furthermore a view has to take care to only draw into the area of the OpenGL window it was assigned on creation (`glViewport`). If you've implemented this task correctly you should now see a coordinate plane in each view with a red x-axis and a green y-axis.

Hint: Stay as flexible as possible with your multi-view implementation in order to support 3, 6 or 8 views with only a few modifications to your code.

3rd Task: View-Interaction

So far the same camera setting is used in each view. This has to be changed now. Each view has to implement two display modes. One mode has to support arbitrary change of camera parameters. For the other mode the camera orientation is defined beforehand and afterwards only camera distance and field-of-view (fovy) are to be modified. To implement this behavior use a flag `fixCamera` and block "illegal" operations. Implement the following settings for the modes of operation with this mechanism:

Single View Default scene view: Camera settings can be modified arbitrarily

Double View The left view positions and orientates the camera to look along the negative z-axis. Consequently the x-axis points to the right and the y-axis points upwards. The right view has to use the same settings initially. The camera orientation has to be alterable though.

Quad View The view in the lower right corner has to use the alterable camera initially using the settings already known from the double view. The remaining views feature a fixed camera orientation. In the upper left view the camera looks along the negative z-axis, along the the positive y-axis in the upper right view and along the negative x-axis in the lower left view.

Like in the previous assignment, you have to be able to modify camera parameters. The mouse movements are mapped to the camera movement as in `HelloCube3` with one exception. That is, the middle mouse button now controls the fovy and the right mouse button modifies the camera distance. The camera distance should be a view-dependent state whereas the fovy should be a global state. Use the according methods of the `View` class to set camera properties. Keep `resetCamera` in mind, the slot to set the parameters to their default values. Compare your implementation with the example program to make sure your settings behave correctly.

As in the first assignment was some confusion about the difference between camera distance and fovy, here is a short explanation. If you take a photo of an object, you have two choices to get a close-up picture. Either you walk closer to the object (smaller camera distance) or you use your zoom lens selecting only a part of the big picture (smaller field of view) yielding in a different result.

Furthermore the modeller has to implement an "active" view. The active view has to be highlighted displaying a yellow frame (`View::highlight`). For each mouse click you have to check in which view the click was issued and to update the active view accordingly.

5 Scene Graph - Graphical Objects

So far the rendered scene is rather unimpressive. It's time to add some graphical objects. Unlike the last assignment you have to use a data structure to model the scene. To that end a so called scene graph is used in the field of computer graphics. The module `csG` defines such a simple scene graph.

Scene graphs are a concept for creating complex hierarchical 3D scenes and consist of so called nodes. These nodes can be hierarchically combined to a directed acyclic graph (DAG).

A node can draw the geometry or it can define certain properties like transformation or color applied to all nodes arranged below. Figure 1 shows a simple scene graph.

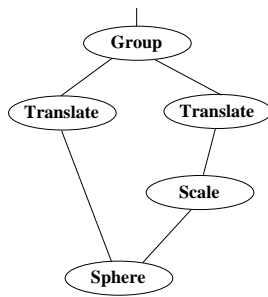


Figure 1: Example of a scene graph.

Consider that a node may have multiple parents. A scene described by a scene graph is rendered traversing the scene graph using a depth first strategy. Thus the scene graph in figure 1 models two spheres. The spheres are displaced by a certain amount relative to the world coordinate system. Additionally the second sphere is scaled. Think about how moving the scaling node in the right subtree in front of the translation node effects the final scene.

5.1 Node Types

The module `csg` provides a set of different node types. They are all derived from the base class `csgNode`, which supplies the basic functionality like a reference mechanism (see next paragraph) and naming. Basically you can differentiate between two classes of nodes.

Shape-Nodes define the different types of geometry. They are all derived from the base class `csgShape`. In detail, the module provides a sphere, a cube and a cylinder for basic primitives. Shape nodes cannot own any child nodes, thus are leaf nodes of the scene graph by definition.

Group-Nodes: All other nodes are derived from the base class `csgGroup` and allow adding of child nodes and thus are the basis for a hierarchically modeling of the scene graph. Next to grouping functionality, the nodes derived from `csgGroup` additionally implement functionality for material properties and transformations applied to all child nodes.

5.2 Reference Mechanism

Usually objects in C++ are allocated using the `new` operator and deleted by the `delete` operator. Scene graphs though may introduce a high degree of node-interdependencies. A node itself may create new nodes. A reference to a node may be stored in multiple different places. Once a reference to a node is no longer needed in some part of your program or your data structure, the question arises how to detect whether the referenced node is still referenced by other nodes. The solution is a *reference mechanism*.

The basic idea is as follows: Nodes are still allocated using the `new` operator. However, the nodes are never explicitly deleted. Instead each node stores a counter on how often it is referenced. Once the reference counter drops to 0, the node deletes itself. This behavior impacts the way working with pointers. Each time a pointer to the instance of the object in question is created, stored or copied, the reference counter of the node has to be incremented calling the method `ref()`. Each time a reference is no longer needed, the reference counter has to be decremented calling the method `unref()`.

Keep the following important behavior in mind: By default the reference counter of a node created with `new` is initialized with 0. If you like to ensure the validity of the reference counter you have

to increment it calling `ref()`. Otherwise a situation may occur in which, e.g., a reference to the node is obtained in the context of traversing the scene. Thus the reference counter is incremented and later on discarding the reference is decremented. Consequently the node detects a reference counter of 0 and deletes itself. Thus, the reference gets invalid. If implemented correctly, the invalidated reference should not be used anymore by any other parts of the scene graph¹.

After you have understood the basics of a scene graph, you can attend to the next task.

4th Task: Creating the Scene Graph

First create an empty scene graph in `OpenGLFrame`, which consists of a single group node `csgGroup` only. This `rootNode` has to be registered at each view. (`setScene`). Afterwards implement the three methods `createSphere`, `createCube` and `createCylinder`. These methods have to create a new subtree of the scene graph, which explicitly has to be added as child node to the existing root-node. The subtree has to contain a translation, a rotation and a scaling transformation allowing interaction with the objects. Each node has to be rotated about its mid point, it has to be resized, and placed in an arbitrary positions relative to the origin of the 3D scene. Think of the order in which to apply the transformations `csgTranslate`, `csgScale` and `csgRotate` to achieve this behavior.

During the entire run time of the program solely one object of `csgCube`, `csgCylinder` or `csgSphere` has to exist. Use multiple references of the same object. Connect the methods with the according menu items of the Edit menu.

Additionally, store a reference to the node allocated last in the pointer variable `selected`. Implement a method `selectObject(csgNode *)` to set the variable. Consider assigning the argument value to the pointer variable `selected` invalidates the previous reference! Mark the currently selected node by calling the method `csgNode::highlight(bool)` – this method should draw a red bounding box around the object. Take care that your scene graph is drawn in each view (`csgNode::render`). Don't be surprised in case a cube you create is not drawn. You have to implement the method `csgCube::render`.

To obtain a better understanding of the structure of a scene graph, you should have a look at the *Jeep* implementation contained in the skeleton of the program. The function `createJeep` is found in the file `createJeep.txt`. You just have to merge the content into your `OpenGLFrame.qt.C`. Possibly you have to rename some variables. The jeep is drawn correctly after you have implemented your transformation nodes and the drawing routine for the cube. If you like to earn an extra point you may model an additional hierarchical object. The scene graph has to have a minimal depth of three and at least one node has to be referenced more than once.

6 3D Graphics with OpenGL

In the last assignment you have been introduced to the basic functionality of OpenGL. Now you'll learn some new possibilities we like to consider in the context of scene graphs.

6.1 Coordinate Systems and Transformations

In the last task you've already implemented camera interaction. Besides you're now familiar with coordinate systems and transformations in OpenGL. In this task you have to implement a more user friendly interaction to transform objects.

¹Otherwise, your program will obviously crash and burn, or behave in some other ways not intended

The transformations have to affect only the selected object. Thus you cannot define the transformations in the `paintGL()` method, but you have to define them in the according transformation nodes of the scene graph.

5th Task: Object Transformation

Implement object transformation for the single view operation mode first. Use the event handling mechanism known from the previous assignment to implement a mouse handler triggering the following events:

Left Mouse Button: triggers the rotation of the selected object. Rotations have to be applied incrementally. Determine the axis of rotation from the relative mouse movements. The axis of rotation has to be perpendicular to the the vector $(x, y, 0)$ and $(0, 0, -1)$.

Middle Mouse Button: translates the object. The X-, Y-movement of the mouse has to be mapped onto X-, Y-translation of the object.

Right Mouse Button: translates the object in Z-direction. You have to map the Y-movement of the mouse onto the Z-axis by doing so.

Object scaling has to happen when the CTRL/STRG key is used together with a pressed mouse button. Implement scaling in the X-axis while the left mouse button is pressed, scaling in the Y-axis while the middle mouse button is pressed and scaling in the Z-axis while the right mouse button is pressed, respectively. Make sure the user cannot define illegal values. Compare your implementation with the example program provided and mimic its behavior!

Reset the object transformations of the selected object whenever the menu item `Reset Object` is selected.

You'll notice transformations of the selected object effecting all other objects defined. This is because the OpenGL transformation matrix is not set absolutely, but multiplied with the current transformation matrix. While traversing the scene graph, each transformation node consequently has to revoke the modification of the current transformation matrix. Use the OpenGL matrix stack for this purpose.

6.2 View Dependent Transformation

In case you started your program in the quad view operation mode, you'll realize the translation and rotation behaving as specified only in the upper left view. In the other views, the mouse movement does not match the corresponding object transformations. This is due to the different coordinate systems.

6th Task: View Dependent Transformation

Implement the method `viewTransform` in the class `View`. This method has to correct the relative mouse movement according to the camera rotation valid in this view. Think about how object translation has to be implemented for the object to move in the X/Y plane *after* the world to view transformation has been applied. Of course the correction not only has to work correctly for camera rotations about 90° . By means of this method you can implement object translation and object rotation for handling mouse events correctly.

Hint: The file `m4x4.H` contains a vector and a matrix class which may be helpful. The classes themselves are template classes which can be declared by `m4x4<double>` or `vector4<float>` for instance.

To make sure your program works correctly you should compare its behavior to the reference implementation once again.

6.3 OpenGL Picking

So far, always the last object created is selected. Now the possibility of selecting one of the rendered objects has to be implemented. You have to use the OpenGL picking mechanism for it.

For picking OpenGL makes use of a special render mode (`glRenderMode(GL_SELECT)`). Objects are not rendered into the frame buffer but checked for an intersection with the viewing frustum instead. Thus picking is done switching to the render mode `GL_SELECT` and drawing the scene "as usual". On switching back to the default render mode (`glRenderMode(GL_RENDER)`), the amount of objects inside the frustum is returned.

In case the scene is rendered with the "default" view frustum, the result is not really meaningful. Eventually, most of the rendered objects fall into the field of view of the viewer. Thus, the world to view transformation has to be adapted by multiplication with a special *pick matrix* (`gluPickMatrix`) restricting the field of view to a small area around the the mouse position. This is illustrated by figure 2.

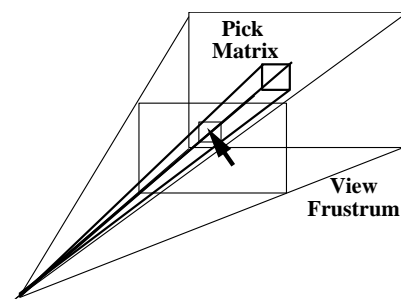


Figure 2: Pick Matrix

The identification of rendered individual objects is done by the means of the OpenGL *Name Stack*. It allows to define integer values as names using `glLoadName(<name>)`. Names can be hierarchically defined using `glPushName(<name>)` instead.

The content of the *Name Stack* at the point in time at which an object has been drawn inside the pick frustum can be determined by a buffer (`glSelectBuffer()`). After rendering while `GL_SELECT` was active, the buffer contains the content of the name stack and the minimal and maximal Z value for each object.

A more detailed description and examples can be found in the *OpenGL Programming Guide* (chapter 13).

7th Task: Picking

Implement the method `OpenGLFrame::pick`. It has to be called for each `mousePressEvent` if the `Adjust Object` mode is active.

The method `pick` has to initialize the OpenGL *Name Stack* and to allocate a buffer in which to retrieve the picking result. Further, the method has to draw the scene just like `paintGL()`, however, using a pick matrix and an adapted view matrix. To that end you have to modify the `View` class accordingly. Adapt the `csgGroup::render()` to manage the *Name Stack* correctly. A clever choice of names allows the content of the *Name Stack* to traverse the scene graph after picking took place.

Implement a method `OpenGLFrame::processHits` to analyze the picking result stored in the select buffer. Comparing the minimal z values of the individual hits find the closest hit and determine the subtree of the scene graph referencing the closest object by examining the contents of the *Name Stack*. Make this object the currently selected.

A name should be assigned to each object to make their distinction easier for the user, because objects can look quite identical. The class `csgNode` already contains methods to set and obtain

node names (`setName`, `getName`). Think about which node logically has to store the name of the object. Modify the methods `createCube`, `createCylinder` and `createSphere` to assign each object a default name. This name has to be determined for the object currently selected and, utilizing the signal-slot mechanism, displayed in a text field of the application's toolbar (*optional*: Implement the inverse operation, i.e., allow changing the name by accepting user input via the text field)

Finally, implement the slot to delete the currently selected object.

6.4 OpenGL Lighting

So far the render modes *wireframe* and *solid* don't resemble any kind of realistic appearance of the scene. Particularly with regard to the three dimensional character of the displayed objects it is mandatory to consider lighting effects. To that end, the color of objects is calculated depending on the orientation of their surfaces to a virtual light source. OpenGL allows to define up to 8 such light sources, whereas the first light source is already initialized with reasonable values by default. Thus it is sufficient to just activate the lighting calculation.

8th Task: Lighting

Add the additional render mode `Lighting` to the options dialog. For this mode OpenGL lighting has to be activated. Expand the according classes to readily make the information whether lighting is desired available to the effected parts of your program.

Although the scene is rendered with OpenGL lighting, the boxes to mark an object have to be rendered as before, i.e., without lighting. The same has to be ensured for the coordinate plane and the frame highlighting the current view.

Possibly your cubes are not drawn correctly. It might be the case that you have forgotten to define a normal or you may have defined a wrong normal. Correct this issue if apparent.

6.5 Material Properties

So far all objects give a rather monotonous impression. Now it'll get colorful. In order to set colors or other various material properties, our scene graph API contains a material node `csgMaterial`. It allows to set a color for diffuse and ambient lighting and to define further object parameters.

9th Task: Material Properties

Use `csgMaterial` nodes to dye your objects. Each object has to have the ability of storing its own material properties. Insert according nodes at the appropriate positions in the scene graph.

Implement the method `csgMaterial::render` to map the defined material properties to OpenGL commands by merely mapping the three parameters, ambient color, diffuse color and opacity to the corresponding OpenGL states. The wire-frame mode now has to use the diffuse color defined to draw the object. To see the effect of the opacity blending has to be enabled and the correct blend function has to be set. Make sure material properties only affect scene graph nodes below a `csgMaterial` node.

To define material properties use the editor `MaterialEditor` found in the program skeleton. It has to be activated by the menu item `ObjectProperties`. Use the signal-slot mechanism to connect the editor with the material node of the object currently edited. When the *Apply* button is pressed, the values defined in the dialog have to be copied into the material node and the editor further emits a `materialChanged` signal to initiate redrawing the scene.

6.6 GLSL Shading Language, Part 2

In task 7 of the last assignment you were introduced to the *fixed function pipeline* which then was replaced by the programmable pipeline. Then we restricted ourselves to transforming the vertices via the model-view and projection matrix only. Now its time to introduce lighting calculations. To that end you'll implement *per-pixel lighting* by the means of a vertex and a fragment shader.

The Blinn-Phong lighting model composes the final color of a point from three components: the ambient term, the diffuse term and the specular term:

$$color = comp_{ambient} + comp_{diffuse} + comp_{specular}$$

The individual terms are defined as follows:

$comp_{ambient} = M_A(G_A + L_A)$, whereas M_A is the ambient material property, L_A the intensity of the ambient light and G_A the intensity of a global ambient light. Heed the fact that material properties as well as light sources are RGBA vectors.

$comp_{diffuse} = M_D L_D (N \cdot L)$, whereas M_D is the diffuse material property, L_D the intensity of the diffuse light, N the vertex normal and L the direction vector pointing from the vertex to the light source.

$comp_{specular} = M_S L_S (N \cdot H)^n$, whereas M_S is the specular material property, L_S the intensity of the specular light, N the vertex normal and H the so called *halfway vector*. This vector lies in the plane defined by the direction vector from the point towards the viewer and the direction vector from the point towards the light position. It is chosen to half the angle between vectors defining the plane. The extent of the specular light is controlled by the specular exponent n .

Make sure the vectors L , H and N are normalized, otherwise you will obtain wrong results.

10th Task: Per-Pixel Lighting

In the last assignment you made your first steps with GLSL, the OpenGL shading language. In the source skeleton of this assignment the implementation of creating the GLSL shader is already done in `OpenGLFrame::initializeGLExt`. As before, `VertexShader.glsl` contains the code of the vertex program and `FragmentShader.glsl` the code of the fragment program, respectively. Now you should extend both shaders with the computation of the Blinn-Phong illumination for light source $k = 0$. You have access to most of the OpenGL states in a GLSL shader. These states begin with `gl_`. For a list of accessible states see the *GLSL Quick Reference Guide*².

The vertex shader should transform the vertex and the corresponding normal. The normal has to be transformed by the inverse transposed model-view matrix (see lecture notes of "Graphical-Interactive Systems"). This matrix is already stored in `gl_NormalMatrix`. It should also compute the halfway vector and the direction to the light for each vertex. You will need *varying* variables to be able to access the computed vectors in the fragment shader (including the normal).

In the fragment shader you will now have to access the material properties of the light (`gl_LightSource[k]`) and the object (`gl_FrontMaterial`). Compute the ambient, diffuse, and specular components of the lighting and combine them. For testing, start your application, switch to filled or lighting mode and enable the programmable pipeline. If all is done correctly, the specular highlight should now be circular and there should be no distinguishable borders on the sphere.

²<http://www.opengl.org/sdk/libs/OpenSceneGraph/glsL-quickref.pdf>.

7 Criteria of Grading

In total you can achieve 20 points in this assignment, extra points excluded. Grading is based on the Jeep scene graph:

1 Point	Comments are in English, clarify the program and are formatted to allow the automatic generation of a documentation of an HTML or \LaTeX document with <code>doxygen</code> .
2 Points	The three modes of operation (single, double, and quad view) work correctly including resizing. Each view renders the same scene. Different modes can be toggled with an command line switch.
3 Points	Camera parameters and interaction are as stated in this assignment. The active view is highlighted by a yellow frame.
2 Points	Object creation for cube, sphere and cylinder works correctly. The subgraph for each object meets the specifications. The objects are rendered correctly. Objects can be selected and deselected.
3 Points	Object movements work as specified. Scene graph transformation nodes are implemented and the matrix stack is used correctly. Object transformations can be reset and work view dependent. Inverse view transformations are taken into account.
3 Points	Picking of rendered objects works. The picked object is highlighted and its name displayed in the application window. Mouse interaction only affects the selected object.
1 Point	The currently selected object can be deleted from the scene graph. The reference mechanism works correctly.
1 Point	Lighting works and has its own menu item in the options dialog. The activation turns on lighting calculation. All objects, even the cube, are correctly lit. Highlight boxes are not lit.
1 Point	Objects can be rendered with different material properties. The properties are rendered as specified. Material properties of the currently selected object can be edited using the material editor.
3 Points	Correct implementation of the Blinn Phong lighting model using per pixel lighting.

Precondition for grading is the error-free compilation of the program on the computers of the VISGS pool. There will be deduction of points if the program compilation leads to warnings (Warnings of QT in the style of “...has virtual functions but non-virtual destructor” are excluded therefrom)! The annotation of the source code should be that extensive to provide enough information to the author to explain the functionality of the code to the supervisors.