Aufgabe 1: Ui, OpenGL!

Martin Aumüller *
Stefan Zellmann †
Universität zu Köln

18. Oktober 2016

1 Themen

In dieser Aufgabe werden folgende Themengebiete bearbeitet:

- 3D-Grafik mit OpenGL
- einfache geometrische Objekte
- OpenGL Vertex Arrays
- OpenGL Vertex Buffer Objects
- Beleuchtung und Materialeigenschaften mit OpenGL
- einfache Vertex- und Fragment-Shader

2 Einführung

In dieser ersten Aufgabe des Praktikums zur Computergrafikprogrammierung wirst Du lernen, eine kleine Anwendung zu schreiben, die hauptsächlich 3D-Grafik zur Ausgabe verwendet. Die Anwendung ist in objektorientiertem C++ geschrieben und benutzt die Qt-Bibliothek für eine plattformübergreifende grafische Benutzerschnittstelle (GUI). Du wirst lernen, wie umfangreiche 3D-Modelle effizient und ansehnlich dargestellt werden können.

Unsere Anwendung erlaubt das Laden von Modellen im `.ply`-Format. In diesem Format liegen umfangreiche, durch 3D-Scans gewonnene Daten vor [5, 3]. Diese Objekte werden dann mit verschiedenen Techniken durch OpenGL dargestellt, so dass diese Objekte interaktiv bewegt werden können.

3 Erste Schritte

Dieses Dokument wird Euch während des gesamten Entwicklungsprozesses begleiten und Euch schrittweise anleiten, das vorgeschlagene Programm zu implementieren. Ihr fangt mit einem Code-Gerüst an, d. h. Ihr bekommt den Quelltext zu einem funktionierenden Programm, dem aber viele Funktionen fehlen. Das Gerüstprogramm und Zusatzmaterial steht auf unserer Webseite unter `http://vis.uni-koeln.de/ex10.html` zur Verfügung. Packe es auf einem UNIX-System mit dem Befehl `tar xzvf a1-base.tar.gz` aus. Wechsle in das neu entstandene Verzeichnis `a1` und starte den Übersetzungsvorgang mit `qmake` und `make`. Dann solltest Du ein ausführbares Programm vorliegen haben, das Du mit `./gl` starten kannst.

Dieses Programm öffnet ein neues Fenster und verfügt bereits über alle notwendigen GUI-Elemente. In der Mitte dieses Bereichs zeigt es die Koordinatenachsen, einen Platzhalter für die Lichtquelle und das geladene Modell an. Deine Aufgabe ist es, die fehlenden Funktionen einzubauen.

In Tabelle 1 sind die Quelltextdateien und die darin zu findenden Klassen aufgeführt.

Zusätzlich zur Bibliothek Qt [6] benötigt Ihr für alle OpenGL-basierten Aufgaben noch die Bibliothek GLEW [2] zum einfachen Umgang mit OpenGL Extensions.

4 Programmdokumentation mit `doxygen`

Aufgabe 1: Programmdokumentation

Da auch wir (und vielleicht auch Ihr selbst noch in einem Jahr) Euer Programm verstehen wollen, müsst Ihr es dokumentieren. Benutzt dazu `doxygen`, das speziell formatierte Kommentare im Quelltext in eine Dokumentation Eures Programms z. B. im HTML-Format umwandeln kann. Richtet Euch nach den Beispielen im Gerüstprogramm und schlagt in [1] für genauere Information

*aumueller@uni-koeln.de
†zellmann@uni-koeln.de

Datei	Klasse	Beschreibung
`main.cpp`	-	Programmstart
`ply.c`	-	Leseroutinen für PLY-Dateien
`ApplicationWindow.cpp`	*ApplicationWindow*	Hauptfenster des Programms
`ApplicationWindow.ui`	*Ui::ApplicationWindow*	Qt-Designer-Beschreibung dazu
`UserParameterDialog.cpp`	*UserParameterDialog*	Dialogbox für Einstellungen
`UserParameterDialog.ui`	*Ui::UserParameterDialog*	Qt-Designer-Beschreibung dazu
`GLFrame.cpp`	*GLFrame*	OpenGL-Zeichenfenster
`Model.cpp`	*Model*	Laden und Zeichnen von PLY-Modellen
`Shader.cpp`	*Shader*	Klasse zur Verwaltung von GLSL-Shadern

Tabelle 1: Dateien und Klassen

nach. Kommentiert v. a. auch, was Ihr im weiteren Verlauf programmiert!

4.1 Die wichtigsten Klassen

Hier sind kurz die Aufgaben der wichtigsten Klassen des Programms dargestellt.

4.1.1 Anwendungsfenster: *ApplicationWindow*

Die zentrale Klasse in unserem Programm ist *ApplicationWindow*. Sie ist von *QMainWindow* abgeleitet und enthält ein *Ui::ApplicationWindow* zum Erzeugen des Hauptfensters. Letztere Klasse wurde aus `ApplicationWindow.ui` erzeugt. In ihrem Konstruktor wird sein Inhalt, z. B. die Menüleiste festgelegt. In der zum Programmgerüst gehörenden Datei `ApplicationWindow.ui` werden bereits alle erforderlichen Menüs angelegt.

4.1.2 Zeichenbereich: *GLFrame*

GLFrame verwaltet den Zeichenbereich und regelt das Zeichnen aller 3D-Elemente außer dem eigentlichen Datensatz. Hierzu wurde *GLFrame* von *QGLWidget* abgeleitet. Diese Klasse stellt ein Fenster zur Verfügung, in das mit OpenGL-Aufrufen gezeichnet werden kann. Drei Methoden sind hier von besonderer Bedeutung:

initializeGL() Qt führt diese Routine einmal, wenn das Widget angelegt wird, aus. Dort sollte die „OpenGL State Machine“ initialisiert werden.

resizeGL(int width, int height) Diese Methode wird aufgerufen, wenn sich die Größe des OpenGL-Fensters ändert. Die Parameter sind die neue Breite bzw. Höhe.

paintGL() Hier wird gezeichnet: wann immer ein Aktualisieren des Fensterinhalts nötig ist, ruft Qt diese Methode zum Neuzeichnen auf. Haben sich die Parameter geändert, sollte man *paintGL()* nicht direkt aufrufen, sondern durch den Aufruf von *updateGL()* mitteilen, dass ein Neuzeichnen notwendig ist. Indirekt werden von hier aus auch die Zeichenroutinen in *Model* zum Darstellen des 3D-Datensatzes aufgerufen.

4.1.3 Laden und Zeichnen der 3D-Daten: *Model*

In der Klasse *Model* wird die eigentliche Arbeit erledigt: dort finden die wichtigen OpenGL-Aufrufe statt.

Das 3D-Modell soll auf drei verschiedene Arten, aber mit dem selben Resultat, gezeichnet werden können. Die erste, basierend auf dem *immediate mode* und *glBegin(...)* und *glEnd()* ist bereits in *Model::drawImmediate(...)* vollständig implementiert. Im weiteren Verlauf wirst Du zwei leistungsfähigere Methoden implementieren.

4.1.4 Shader-Management: *Shader*

Die Klasse *Shader* unterstützt bei der Verwendung von in der *OpenGL Shading Language (GLSL)* implementierten Programmen, die auf der Grafikkarte ablaufen. Sie lädt (*loadVertexSource*, *loadFragmentSource*) und kompiliert (*link*) diese Programme, aktiviert (*enable*) und deaktiviert (*disable*) sie und setzt Parameter (*setUniform1f*).

4.1.5 Einstellungsdialog: *UserParameterDialog*

Die Klasse *UserParameterDialog* zeichnet eine nichtmodale Dialog-Box zum Verändern des Shaderparameters Üser". Dazu ist diese *QDialog* abgeleitet. Außerdem enthält sie die Klasse *Ui::UserParameterDialog*, die sich aus `UserParameterDialog.ui` ergibt.

5 3D-Grafik mit OpenGL

Der Teil zur 3D-Grafik-Ausgabe dieses Programms wird mit der OpenGL-Bibliothek realisiert. OpenGL ist ein hersteller- und plattformübergreifender Standard, die entsprechenden Dokumente sind unter [4] verfügbar. Eine gute Einführung, die für unsere Zwecke schon beinahe ausreichend ist, bieten die ersten 5 Kapitel von [10].

OpenGL umfasst Kommandos zum Zeichnen verschiedener Grafikprimitiven in unterschiedlichen Techniken. Zwei Dinge sind charakteristisch für OpenGL:

- OpenGL ist eine Zustandsmaschine („state machine“), d. h. die Auswirkung eines Zeichenbefehls hängt vom Zustand, in den die OpenGL-Maschine durch die vorangegangen Befehle gebracht wurde, ab. Dadurch werden die Funktionsaufrufe einfacher und schneller, da nicht jedes Mal die vollständige Information zum Zeichnen von Primitiven an OpenGL übergeben werden muss.

- OpenGL ist ursprünglich eine Programmierbibliothek zum unmittelbaren Zeichnen („immediate mode“), d. h. ein Zeichenbefehl wirkt sich sofort auf den Bildspeicher der Grafikkarte aus (obwohl mit den „display lists“ schon immer eine Möglichkeit für „retained mode“ gegeben ist). Das schränkt die Möglichkeiten für das Rendering ein, aber wegen seiner Geschwindigkeit ist dieser Ansatz für Echtzeitgrafik gut geeignet.

  Erst die moderneren Techniken wie *Vertex Arrays* und *Vertex Buffer Objects*, welche Ihr später verwenden werdet, weichen dieses Prinzip ein wenig auf.

5.1 Grafikprimitiven

OpenGL kennt nur einfache geometrische Formen wie Punkte, Linien und Polygone. Flächen höherer Ordnung müssen durch diese approximiert werden.

Das Zeichnen von Primitiven wird durch *glBegin(GLEnum mode)* eingeleitet und durch *glEnd()* abgeschlossen. Abhängig von *mode* können beispielsweise Linienzüge („line strips“) oder Dreiecksfächer („triangle fans“) gezeichnet werden. Zwischen diesem Befehlspaar werden die Knoten des Objekts z. B. durch *glVertex3f(...)* angegeben. Ein einfaches Rechteck kann so gezeichnet werden:

```
glBegin(GL_QUADS);
  glVertex3f( 0.0, 0.0, 0.0);
  glVertex3f( 1.0, 0.0, 0.0);
  glVertex3f( 1.0, 1.0, 0.0);
  glVertex3f( 0.0, 1.0, 0.0);
glEnd();
```

Kennt Ihr bereits den OpenGL-Befehl, den Ihr verwenden wollt, z. B. *glVertex3f(...)*, so könnt Ihr mit dem Kommando `man glVertex3f` auf UNIX-Systemen Hilfe dazu aufrufen. Alternativ ist die Referenzdokumentation auch auf der OpenGL-Webseite verfügbar: `http://www.opengl.org/sdk/docs/man/`.

5.2 Beschleunigung durch weniger OpenGL-API-Aufrufe

Das oben beschriebene Verfahren zum Zeichnen erfordert aber sehr viele OpenGL-API-Aufrufe und erzeugt deshalb beträchtlichen Zusatzaufwand. Um diesen zu umgehen, existieren Aufrufe, die Gruppen anderer API-Aufrufe bündeln.

Mittels *glDrawElements(...)* lassen sich die Aufrufe zum Zeichnen einer Primitive, also ein vollständiger *glBegin(...)/glEnd()*-Block, zusammenfassen. *glMultiDrawElements(...)* wiederum kombiniert mehrere *glDrawElements(...)*-Aufrufe.

Beide Aufrufe greifen auf sog. *Vertex Arrays* zu, aus denen die Attribute der zu zeichnenden Vertices stammen. Diese werden über *glColorPointer(...)*, *glNormalPointer(...)* bzw. *glVertexPointer(...)* befüllt und mit *glEnableClientState(...)* aktiviert, so dass sie dann durch *glMultiDrawElements(...)* verwendet werden.

Aufgabe 2: Zeichnen mit *glMultiDrawElements(...)*

Eure Aufgabe ist es, in *Model::drawArray(...)* die Funktionalität von *GLFrame::drawImmediate(...)* unter Verwendung von *glMultiDrawElements(...)* zu implementieren, so dass das Zeichnen des gesamten geladenen Modells mit einer von der Modellgröße unabhängigen Anzahl von OpenGL-Aufrufen geschieht. Dazu müsst Ihr auch in *Model::computeVertexArrayData()* die Datenstrukturen geeignet anlegen.

Solltet Ihr mysteriöse Fehler erleben, so lohnt sich vielleicht das Einstreuen des Makros *CHECKGL* in den Code.

Der Menüpunkt „OpenGL“ wählt zwischen den unterschiedlichen Zeichenmethoden.

5.3 Beschleunigung durch Nutzung von Grafikspeicher

Auch bei Nutzung von *Vertex Arrays* müssen bei jedem Zeichenvorgang, also bei jedem *Frame*, die Daten erneut zur Grafikkarte übertragen werden – obwohl sich in unserem Beispiel diese Daten nicht verändern. Mittels *Vertex Buffer Objects* lässt sich das vermeiden.

Hierzu lädt man die Daten in einen mittels *glGenBuffers(...)* angelegten Puffer, indem man *glBufferData(...)* aufruft, nachdem der Puffer mittels *glBindBuffer(...)* aktiviert wurde.

Aufgabe 3: Zeichnem mit VBOs

Puffert diese Daten in *Model::bufferData()* und ergänzt *Model::computeVertexArrayData*, so dass dort die notwendigen Offset-Listen erstellt werden.

Implementiert *Model::drawVBO(...)* so, dass die Daten für die Aufrufe von *gl...Pointer(...)* und *glMultiDrawElements(...)* aus *Vertex Buffer Objects* stammen. Diese Aufrufe beziehen die Daten aus den VBOs, wenn diese mittels *glBindBuffer(...)* aktiviert wurden. Dann werden die übergebenen Zeigerargumente nicht mehr als Speicheradressen sondern als Byte-Offsets in die gepufferten Daten interpretiert.

5.4 Beleuchtung und Normalen

Damit das Modell ein wenig realistischer wirkt, wird es von einer Punktlichtquelle beleuchtet.

Die Beleuchtungsberechnung erfordert Normalen. Je nach gewünschtem Effekt müssen diese auf verschiedene Art ermittelt werden. Geht es darum, die einzelnen Seiten eines Polyeders zu betonen, müssen alle Vertices eines Polygons unter Verwendung derselben Normalen beleuchtet werden. Es wird also eine Normale pro Seitenfläche verwendet.

Ist das Polygonmodell in Wirklichkeit die Approximation eines Objekts mit gekrümmter Oberfläche, kommen zusammen mit dem sog. Gouraud-Shading an jedem Vertex Normalen zum Einsatz, die sich durch Mittelung der Normalen an den angrenzenden Flächen ergeben. Durch Interpolation der an den Polygonecken ermittelten Farbwerte über das Polygon entstehen keine Kanten mit Farbsprüngen und die Darstellung des Modells wirkt glatter.

Aufgabe 4: Vertexnormalen

Manche der im `.ply`-Format vorliegenden Modelle besitzen keine Vertex-Normalen. Deine Aufgabe ist es, solche zu ermitteln. Beachte dabei, dass nur die Richtung der Normalen, nicht aber ihre Länge von Bedeutung ist. D. h., Du solltest die vorhandenen Flächennormalen normieren, bevor Du ihren Mittelwert bestimmst.

Vervollständige dazu die Methode *Model::computeVertexNormals()*. Du kannst zwischen Face- und Vertex-Normalen im Menü „Rendering“ umschalten.

5.5 Shader-Programme

Anstatt die Vertex- und Fragment-Daten durch die klassische OpenGL-Pipeline bearbeiten zu lassen, kannst Du diese auf Systemen, die OpenGL 2.0 unterstützen, auch mit eigenen Programmen verarbeiten. Diese Programme werden in der OpenGL Shading Language, einer C-ähnlichen Sprache, geschrieben. Im Gerüstprogramm ist bereits eine Klasse *Shader* definiert, mit dessen Hilfe Du solche Programme laden kannst. Hierzu musst Du ein Vertex- oder Fragment-Programm mit *loadVertexSource("Filename.vsh")* bzw. mit *loadFragmentSource("Filename.fsh")* laden und anschließend mit *link()* kompilieren. *enable()* aktiviert diesen Shader und ersetzt die Verarbeitung von Vertices und Fragmenten in der Fixed Function Pipeline durch die angegebenen Shader. D. h. Du musst die gesamte benötigte Funktionalität aus der klassischen OpenGL-Verarbeitung in Deinem Vertex- bzw. Fragment-Programm nachbilden. Um die Shader-Programme wieder auszuschalten, rufe einfach *disable* in der *Shader*-Klasse auf. aktiv. Ab diesem Zeitpunkt wird dann (durch Laden des Programms mit der Kennung 0) wieder die klassische Verarbeitung aktiviert.

Vertex-Programme können auf die gesamten Vertex-Attribute wie Position, Farben und Normalen sowie auf uniforme Daten wie die Model-View-Projection-Matrix zugreifen. Man kann Daten von Vertex- an Fragment-Programme weitergeben, indem man *varying*-Parameter einführt – deren Werte werden dann, genauso wie einige Ausgabewerte der Vertex-Verarbeitung – mittels baryzentrischer Koordinaten von den benachbarten Vertices auf die Fragmente interpoliert und stehen dann in den Fragment-Programmen zur Verfügung. Außerdem kann man mittels *glUniform...* auch eigene uniforme Parameter an Vertex- und Fragment-Programme weitergeben. Die Details dazu stehen in Kapitel 7 von [9].

Im Gerüstprogramm wird dieser Mechanismus bereits genutzt, um die beiden uniformen skalaren Parameter *Time* und *UserParameter* an die Shader weiterzugeben.

Man hat wenig Möglichkeiten, Shader-Programme zu debuggen. Aber oft ist es hilfreich, die erzeugten Daten bereits in einem früheren Schritt als Farben auszugeben. Dazu müssen diese aber auf das Intervall $[0., 1.]$ normiert werden.

5.6 Emulation der Fixed Function Pipeline durch Shader

Das Zeichnen des Modells erfolgt in der Grundeinstellung durch die *Fixed Function Pipeline*. Deine Aufgabe ist es, einen jeweils möglichst einfache Vertex- und Fragment-Shader zu implementieren, der ein identisches Bild erzeugt. Erstelle Deine Shader in den Dateien `simple.vsh` und `simple.fsh`. Inspiration gibt es im Abschnitt „Emulating OpenGL Fixed Functionality“ von [9], aber Eure Shader sollten speziell an den vorliegenden Fall angepasst sein und nichts überflüssiges machen. Lediglich folgendes sollte Euer Shader für den Fall einer einzigen Punktlichtquelle und eines *Local Viewer*s tun:

- Berechnung von *gl_Position*,
- Berechnung von *gl_FrontColor* unter Berücksichtigung von *gl_FrontMaterial* und *gl_LightModel.ambient* sowie *gl_LightSource[0]*.

Aufgabe 5: Fixed Function Pipeline als Shader

5.7 Shader für Phong-Shading

Beim der *Fixed Function Pipeline* zugrundeliegenden Gouraud-Shading wird das Beleuchtungsmodell an den Vertices ausgewertet, die so erhaltenen Farben werden dann über die Polygone interpoliert.

Beim Phong-Shading werden die für das Beleuchtungsmodell notwendigen Normalen über die Polygone interpoliert, und für jeden Pixel wird das Beleuchtungsmodell unter Berücksichtigung der interpolierten Normalen ausgewertet.

Benutze `simple.vsh` als Basis und verlagere dazu einen Teil der Berechnungen, die im Vertexprogramm durchgeführt wurden, in den Fragment-Shader `phong.fsh`, indem Du mithilfe von *varying*-Parametern vom Vertex- zum Fragmentshader kommunizierst. So übertragene und dabei interpolierte Normalen müssen vor der Verwendung im Fragmentprogramm erneut normiert werden, da die Normiertheit bei der Interpolation verlorengeht.

Aufgabe 6: Phong-Shading

Implementiere ein Vertex-/Fragment-Shader-Paar in den Dateien `phong.vsh` bzw. `phong.fsh` fürs Phong-Shading.

5.8 Eigener Shader

Die Shader-Programmierung bietet sehr viele Möglichkeiten. Implementiere Shader `freestyle.{v,f}sh` nach Deinem Geschmack. Hierbei stehen Dir zusätzlich zu den von OpenGL gesetzten Parametern noch folgende uniformen Parameter zur Verfügung:

Time Dieser Parameter ändert sich mit der Zeit und kann für Animationen genutzt werden.

UserParameter Dieser skalare Parameter wird durch den Schieberegler im Einstellungsdialog gesteuert.

Aufgabe 7: Shader nach eigenem Gusto

Du könntest z.B.

- das Modell mit einer Musterung versehen (prozedurale Texturierung),
- Höhenlinien auf dem Modell einzeichnen,
- ein neues Beleuchtungsmodell implementieren,
- die Vertices in Abhängigkeit von der Zeit animieren,
- ...

Deiner Fantasie sind keine Grenzen gesetzt!

Falls Du weitere uniforme Shader-Parameter benötigst, steht es Dir auch frei, diese zu implementieren.

6 Zusammenfassung

Habt Ihr diese Aufgaben gelöst, dann habt Ihr schon ein ganz ordentliches Verständnis von OpenGL zur 3D-Programmierung und der Shading Language entwickelt. In der nächsten Aufgabe werdet Ihr lernen, wie man vieles davon mit weniger Mühe erreichen kann.

7 Bewertungsrichtlinien

Wenn Du die folgenden Bedingungen erfüllst, dann kannst Du die Höchstpunktzahl von 20 Punkten erreichen:

1 Punkt Der Quelltext ist vollständig in objektorientiertem C++ verfasst. Der Code kompiliert auf dem Referenzsystem mit dem GNU C++-Compiler ohne Fehler und erzeugt mit den Optionen `-Wall -O2` keine unnötigen Warnungen. (Lediglich `ply.c` ist davon ausgenommen.) Auch die anderen Programmierrichtlinien wurden befolgt.

2 Punkte Der gesamte Quelltext ist in englischer Sprache kommentiert und die Kommentare erklären die algorithmische Struktur des C++-Codes. Die Kommentare sind so formatiert, dass sie das Erzeugen einer HTML- und LaTeX-Dokumentation mit `doxygen` erlauben.

2 Punkte Aus den Per-Face-Normalen interpolierst Du Per-Vertex-Normalen, die das Modell glatter erscheinen lassen.

3 Punkte Alternativ zur bereits vorhandenen Implementation unter Benutzung *glBegin* und *glEnd* kann das geladene Modell auch mittels *Vertex Arrays* gezeichnet werden.

3 Punkte Die *Vertex Arrays* werden optional aus *Vertex Buffer Objects* mit Daten beschickt.

3 Punkte Ihr habt einen möglichst einfachen Vertex- und Fragment-Shader implementiert, der das Verhalten der *Fixed Function Pipeline* nachbildet.

3 Punkte Ihr habt einen möglichst einfachen Vertex- und Fragment-Shader implementiert, der Per-Pixel-Phong-Lighting macht.

3 Punkte Du hast einen Shader implementiert, der etwas Witziges macht.

Literatur

[1] Doxygen manual. `http://www.stack.nl/~dimitri/doxygen/manual.html`, 2010.

[2] Glew: The opengl extension wrangler library. `http://glew.sourceforge.net/`, 2010.

[3] Large geometric models archive at georgia tech. `http://www.cc.gatech.edu/projects/large_models/`, 2010.

[4] Opengl api documentation overview. `http://www.opengl.org/documentation/`, 2010.

[5] The stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`, 2010.

[6] Qt reference documentation (open source edition). `http://qt.apidoc.info/5.2.0/qtdoc/`, 2013.

[7] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2nd edition, 1990.

[8] Paul Martz. *OpenGL Distilled*. Addison-Wesley, 2006.

[9] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison Wesley, 3rd edition, 2009.

[10] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide, The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison Wesley, 7th edition, 2009.

[11] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real-Time*. Addison Wesley, 2002.

[12] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 3rd edition, 1997.

[13] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2004.

[14] Richard S. Wright, Nicholas Haemel, Graham Sellers, and Benjamin Lipchak. *OpenGL SuperBible, Comprehensive Tutorial and Reference*. Addison Wesley, 5th edition, 2010.