

Softwarearchitektur
Programmieren eines 3D-Editors
-- Blacksun --

Mitglieder:	Philipp Gruber Reinhard Jeschull Thomas Kuhndörfer Thomas Tischler Stefan Zeltner	pgruber@fh-landshut.de rjeschu@fh-landshut.de tkuhndo@fh-landshut.de ttischl1@fh-landshut.de szeltne@fh-landshut.de
Betreuer:	Prof. Peter Hartmann	
Homepage:	http://sourceforge.net/projects/blacksun/	
letzte Änderung:	Montag, 18. Juni 2007	

Inhaltsverzeichnis

1 Übersicht.....	8
1.1 Funktionale und nicht funktionale Anforderungen.....	8
1.2 Aufteilung der Arbeit (Kernmodule).....	8
1.3 Architektursicht – Gesamtsicht.....	8
1.4 Architektursicht – Modulsicht.....	8
2 Core/Logger.....	9
2.1 UML.....	9
2.2 Funktionale und nicht funktionale Anforderungen.....	9
2.3 Mengengerüste.....	9
2.4 Architektursicht – Gesamtsicht.....	9
2.5 Architektursicht – Klassensicht.....	10
2.6 Architektursicht – Laufzeit.....	10
2.7 Schnittstellen nach außen.....	10
2.8 Einflussfaktoren und Randbedingungen.....	10
2.9 Globale Entwurfsentscheidungen.....	10
3 Math.....	11
3.1 UML.....	11
3.2 Funktionale und nicht funktionale Anforderungen.....	11
3.3 Mengengerüste.....	11
3.4 Architektursicht – Gesamtsicht.....	12
3.5 Architektursicht – Klassensicht.....	12
3.6 Architektursicht – Laufzeit.....	12
3.7 Schnittstellen nach außen.....	12
3.8 Einflussfaktoren und Randbedingungen.....	13
3.9 Globale Entwurfsentscheidungen.....	13
4 Renderer.....	14
4.1 UML.....	14
4.2 Funktionale und nicht funktionale Anforderungen.....	15
4.3 Mengengerüste.....	15
4.4 Architektursicht – Gesamtsicht.....	15
4.5 Architektursicht – Klassensicht.....	16
4.5.1 Eigentliches Renderer-System.....	16
4.5.2 Textur- und Material-Sytem.....	18
4.5.3 Zentrale Konfigurationsschnittstelle.....	19
4.6 Architektursicht – Laufzeit.....	19
4.7 Schnittstellen nach außen.....	19
5 Pluginsystem.....	20
5.1 UML.....	20
5.2 Funktionale und nicht funktionale Anforderungen.....	20
5.3 Mengengerüste.....	20
5.4 Architektursicht – Gesamtsicht.....	21
5.5 Architektursicht – Klassensicht.....	21
5.6 Architektursicht – Laufzeit.....	22
5.7 Schnittstellen nach außen.....	22
5.8 Einflussfaktoren und Randbedingungen.....	23
5.9 Globale Entwurfsentscheidungen.....	23
6 Scenegraph.....	24
6.1 UML.....	24
6.2 Funktionale und nicht funktionale Anforderungen.....	25
6.2.1 Grundsätzliche Anforderungen.....	25
6.2.2 Anforderungen betreffend Komponente GUI / Plugins.....	25
6.2.3 Anforderungen betreffend Komponente Renderer.....	25
6.3 Mengengerüste.....	25
6.4 Architektursicht – Gesamtsicht.....	25
6.5 Architektursicht – Klassensicht.....	26

6.6 Architektursicht – Laufzeit.....	30
6.7 Schnittstellen nach außen.....	30
6.8 Globale Entwurfsentscheidungen.....	30
7 Benutzeroberfläche/GUI.....	32
7.1 UML.....	32
7.2 Funktionale und nicht funktionale Anforderungen.....	33
7.3 Architektursicht – Gesamtsicht.....	33
7.4 Architektursicht – Klassensicht.....	33
7.5 Architektursicht – Laufzeit.....	35
7.6 Schnittstellen nach außen.....	35
7.7 Einflussfaktoren und Randbedingungen.....	35
7.8 Globale Entwurfsentscheidungen.....	35
8 Pluginspezifikation.....	36
8.1 Aufbau der dynamischen Bibliothek.....	36
9 Plugins.....	39
10 Grundkörper-Plugins.....	40
10.1 Block.....	40
10.1.1 UML.....	40
10.1.2 Screenshots.....	40
10.1.3 Funktionale und nicht funktionale Anforderungen.....	40
10.1.4 Mengengerüste.....	40
10.1.5 Architektursicht – Klassensicht.....	40
10.1.6 Architektursicht – Laufzeit.....	41
10.1.7 Globale Entscheidungen.....	41
10.2 Box.....	42
10.2.1 UML.....	42
10.2.2 Screenshots.....	42
10.2.3 Funktionale und nicht funktionale Anforderungen.....	42
10.2.4 Mengengerüste.....	42
10.2.5 Architektursicht – Klassensicht.....	42
10.2.6 Architektursicht – Laufzeit.....	43
10.2.7 Globale Entwurfsentscheidungen.....	43
10.3 GeoSphere.....	44
10.3.1 UML.....	44
10.3.2 Screenshots.....	44
10.3.3 Funktionale und nicht funktionale Anforderungen.....	44
10.3.4 Mengengerüste.....	44
10.3.5 Architektursicht – Klassensicht.....	45
10.3.6 Architektursicht – Laufzeit.....	45
10.3.7 Einflussfaktoren und Randbedingungen.....	45
10.3.8 Globale Entwurfsentscheidungen.....	45
10.4 Grid (Gitter).....	46
10.4.1 UML.....	46
10.4.2 Screenshots.....	46
10.4.3 Funktionale und nicht funktionale Anforderungen.....	46
10.4.4 Mengengerüste.....	46
10.4.5 Architektursicht – Klassensicht.....	46
10.4.6 Architektursicht – Laufzeit.....	47
10.4.7 Einflussfaktoren und Randbedingungen.....	47
10.4.8 Globale Entwurfsentscheidungen.....	47
10.5 Helix.....	48
10.5.1 UML.....	48
10.5.2 Screenshots.....	48
10.5.3 Funktionale und nicht funktionale Anforderungen.....	48
10.5.4 Mengengerüste.....	48
10.5.5 Architektursicht – Klassensicht.....	49
10.5.6 Architektursicht – Laufzeit.....	49
10.5.7 Einflussfaktoren und Randbedingungen.....	49

10.5.8 Globale Entwurfsentscheidungen.....	49
10.6 nBox (Zylinder).....	50
10.6.1 UML.....	50
10.6.2 Screenshots.....	50
10.6.3 Funktionale und nicht funktionale Anforderungen.....	50
10.6.4 Mengengerüste.....	50
10.6.5 Architektursicht – Klassensicht.....	51
10.6.6 Architektursicht – Laufzeit.....	51
10.6.7 Globale Entwurfsentscheidungen.....	51
10.7 Pyramid.....	52
10.7.1 UML.....	52
10.7.2 Screenshots.....	52
10.7.3 Funktionale und nicht funktionale Anforderungen.....	52
10.7.4 Mengengerüste.....	52
10.7.5 Architektursicht – Klassensicht.....	52
10.7.6 Architektursicht – Laufzeit.....	53
10.7.7 Globale Entwurfsentscheidungen.....	53
10.8 Sphere (Kugel).....	54
10.8.1 UML.....	54
10.8.2 Screenshots.....	54
10.8.3 Funktionale und nicht funktionale Anforderungen.....	54
10.8.4 Mengengerüste.....	54
10.8.5 Architektursicht – Klassensicht.....	55
10.8.6 Architektursicht – Laufzeit.....	55
10.8.7 Einflussfaktoren und Randbedingungen.....	55
10.8.8 Globale Entwurfsentscheidungen.....	55
10.9 Tube (Röhre).....	56
10.9.1 UML.....	56
10.9.2 Screenshots.....	56
10.9.3 Funktionale und nicht funktionale Anforderungen.....	56
10.9.4 Mengengerüste.....	56
10.9.5 Architektursicht – Klassensicht.....	57
10.9.6 Architektursicht – Laufzeit.....	57
10.9.7 Globale Entwurfsentscheidungen.....	57
10.10 Torus.....	58
10.10.1 UML.....	58
10.10.2 Screenshots.....	58
10.10.3 Funktionale und nicht funktionale Anforderungen.....	58
10.10.4 Mengengerüste.....	58
10.10.5 Architektursicht – Klassensicht.....	59
10.10.6 Architektursicht – Laufzeit.....	59
10.10.7 Einflussfaktoren und Randbedingungen.....	59
10.10.8 Globale Entwurfsentscheidungen.....	59
10.11 Goblet (Kelch).....	60
10.11.1 UML.....	60
10.11.2 Screenshots.....	60
10.11.3 Funktionale und nicht funktionale Anforderungen.....	60
10.11.4 Mengengerüste.....	60
10.11.5 Architektursicht – Klassensicht.....	60
10.11.6 Architektursicht – Laufzeit.....	60
10.11.7 Globale Entwurfsentscheidungen.....	61
11 Modifikator-Plugins.....	62
11.1 Convex Hull (Konvexe Hülle).....	62
11.1.1 UML.....	62
11.1.2 Screenshots.....	62
11.1.3 Funktionale und nicht funktionale Anforderungen.....	62
11.1.4 Mengengerüste.....	62
11.1.5 Architektursicht – Klassensicht.....	62

11.1.6 Architektursicht – Laufzeit.....	63
11.1.7 Einflussfaktoren und Randbedingungen.....	63
11.1.8 Globale Entwurfsentscheidungen.....	63
11.2 Explosion.....	64
11.2.1 UML.....	64
11.2.2 Screenshots.....	64
11.2.3 Funktionale und nicht funktionale Anforderungen.....	64
11.2.4 Mengengerüste.....	64
11.2.5 Architektursicht – Klassensicht.....	64
11.2.6 Architektursicht – Laufzeit.....	65
11.2.7 Globale Entwurfsentscheidungen.....	65
11.3 Extrude.....	66
11.3.1 UML.....	66
11.3.2 Screenshots.....	66
11.3.3 Funktionale und nicht funktionale Anforderungen.....	66
11.3.4 Mengengerüste.....	66
11.3.5 Architektursicht – Klassensicht.....	66
11.3.6 Architektursicht – Laufzeit.....	67
11.3.7 Einflussfaktoren und Randbedingungen.....	67
11.3.8 Globale Entwurfsentscheidungen.....	67
11.4 Insert vertex (Vertex einfügen).....	68
11.4.1 UML.....	68
11.4.2 Screenshots.....	68
11.4.3 Funktionale und nicht funktionale Anforderungen.....	68
11.4.4 Mengengerüste.....	68
11.4.5 Architektursicht – Klassensicht.....	68
11.4.6 Architektursicht – Laufzeit.....	69
11.4.7 Einflussfaktoren und Randbedingungen.....	69
11.4.8 Globale Entwurfsentscheidungen.....	69
11.5 Make Hole.....	70
11.5.1 UML.....	70
11.5.2 Screenshots.....	70
11.5.3 Funktionale und nicht funktionale Anforderungen.....	70
11.5.4 Mengengerüste.....	70
11.5.5 Architektursicht – Klassensicht.....	70
11.5.6 Architektursicht – Laufzeit.....	71
11.6 Revolving (Rotationskörper).....	72
11.6.1 UML.....	72
11.6.2 Screenshots.....	72
11.6.3 Funktionale und nicht funktionale Anforderungen.....	72
11.6.4 Mengengerüste.....	72
11.6.5 Architektursicht – Klassensicht.....	72
11.6.6 Architektursicht – Laufzeit.....	73
11.6.7 Globale Entwurfsentscheidungen.....	73
11.7 Smooth Corners (Weiche Ecken).....	74
11.7.1 UML.....	74
11.7.2 Screenshots.....	74
11.7.3 Funktionale und nicht funktionale Anforderungen.....	74
11.7.4 Mengengerüste.....	74
11.7.5 Architektursicht – Klassensicht.....	74
11.7.6 Architektursicht – Laufzeit.....	74
11.8 Spike (Stacheln).....	75
11.8.1 UML.....	75
11.8.2 Screenshots.....	75
11.8.3 Funktionale und nicht funktionale Anforderungen.....	75
11.8.4 Mengengerüste.....	75
11.8.5 Architektursicht – Klassensicht.....	75
11.8.6 Architektursicht – Laufzeit.....	76

11.8.7 Einflussfaktoren und Randbedingungen.....	76
11.8.8 Globale Entwurfsentscheidungen.....	76
11.9 Subdivision.....	77
11.9.1 UML.....	77
11.9.2 Screenshots.....	77
11.9.3 Funktionale und nicht funktionale Anforderungen.....	77
11.9.4 Mengengerüste.....	77
11.9.5 Architektursicht – Klassensicht.....	78
11.9.6 Architektursicht – Laufzeit.....	78
11.9.7 Einflussfaktoren und Randbedingungen.....	78
11.9.8 Globale Entwurfsentscheidungen.....	78
11.10 Split face (Face aufteilen).....	79
11.10.1 UML.....	79
11.10.2 Screenshots.....	79
11.10.3 Funktionale und nicht funktionale Anforderungen.....	79
11.10.4 Mengengerüste.....	79
11.10.5 Architektursicht – Klassensicht.....	79
11.10.6 Architektursicht – Laufzeit.....	80
11.10.7 Einflussfaktoren und Randbedingungen.....	80
12 Sonstige Plugins.....	81
12.1 Manual edit (Manuelle Bearbeitung).....	81
12.1.1 UML.....	81
12.1.2 Screenshots.....	81
12.1.3 Funktionale und nicht funktionale Anforderungen.....	81
12.1.4 Mengengerüste.....	81
12.1.5 Architektursicht – Klassensicht.....	82
12.1.6 Architektursicht – Laufzeit.....	82
12.1.7 Einflussfaktoren und Randbedingungen.....	82
12.1.8 Globale Entwurfsentscheidungen.....	82
12.2 Materialeditor.....	83
12.2.1 UML.....	83
12.2.2 Screenshots.....	84
12.2.3 Funktionale und nicht funktionale Anforderungen.....	84
12.2.4 Mengengerüste.....	84
12.2.5 Architektursicht – Klassensicht.....	84
12.2.6 Architektursicht – Laufzeit.....	85
12.2.7 Einflussfaktoren und Randbedingungen.....	85
12.2.8 Globale Entwurfsentscheidungen.....	85
12.3 Texturcoordinate-Editor.....	86
12.3.1 UML.....	86
12.3.2 Screenshots.....	87
12.3.3 Funktionale und nicht funktionale Anforderungen.....	87
12.3.4 Mengengerüste.....	87
12.3.5 Architektursicht – Klassensicht.....	87
12.3.6 Architektursicht – Laufzeit.....	88
12.3.7 Einflussfaktoren und Randbedingungen.....	88
12.3.8 Globale Entwurfsentscheidungen.....	88
12.4 Im- und Exporter-Pack.....	89
12.4.1 UML.....	89
12.4.2 Screenshots.....	89
12.4.3 Funktionale und nicht funktionale Anforderungen.....	90
12.4.4 Mengengerüste.....	90
12.4.5 Architektursicht – Klassensicht.....	90
12.4.6 Architektursicht – Laufzeit.....	91
12.4.7 Einflussfaktoren und Randbedingungen.....	91
12.4.8 Globale Entwurfsentscheidungen.....	91
12.5 TextureLoaderPack.....	92
12.5.1 UML.....	92

12.5.2 Funktionale und nicht funktionale Anforderungen.....	92
12.5.3 Mengengerüste.....	92
12.5.4 Architektursicht – Klassensicht.....	93
12.5.5 Architektursicht – Laufzeit.....	93
12.5.6 Einflussfaktoren und Randbedingungen.....	93
12.5.7 Globale Entwurfsentscheidungen.....	93
12.6 Updater.....	94
12.6.1 UML.....	94
12.6.2 Funktionale und nicht funktionale Anforderungen.....	94
12.6.3 Mengengerüste.....	94
12.6.4 Architektursicht – Gesamtsicht.....	94
12.6.5 Architektursicht – Klassensicht.....	95
12.6.6 Architektursicht – Laufzeit.....	95
12.6.7 Einflussfaktoren und Randbedingungen.....	96
12.6.8 Globale Entwurfsentscheidungen.....	96
13 Ergänzungen.....	97
13.1 Das Makefile-Konzept.....	97
13.2 Modul-Konzept.....	97
13.3 Singletons.....	97
13.4 Namespace-Konzept.....	98

1 Übersicht

1.1 Funktionale und nicht funktionale Anforderungen

Alle funktionalen Anforderungen sind im Pflichten-/Lastenheft aufgelistet.

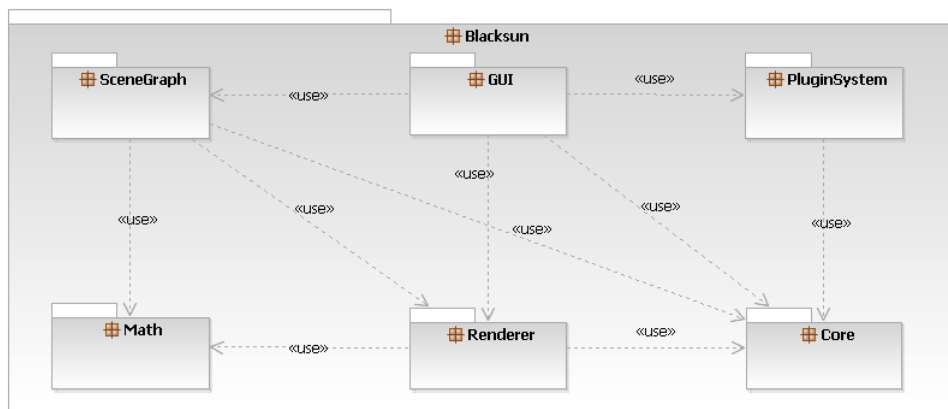
1.2 Aufteilung der Arbeit (Kernmodule)

- **Philipp Gruber:** Core/Logger
- **Reinhard Jeschull:** Mathebibliothek, Renderer
- **Thomas Kuhndörfer (Projektleiter):** SceneGraph
- **Thomas Tischler:** GUI (+ Qt-spezifische Elemente in vielen Modulen)
- **Stefan Zeltner:** Pluginsystem, Make- bzw. Built-System

Nähere Infos zur Einteilung (besonders der Plugins) befinden sich in der Eigenleistungs-Dokumentation.

1.3 Architektursicht – Gesamtsicht

Das folgende Paketdiagramm zeigt die Integration aller Blacksun-Module:



1.4 Architektursicht – Modulsicht

Der Blacksun-Editor besteht aus folgenden Modulen:

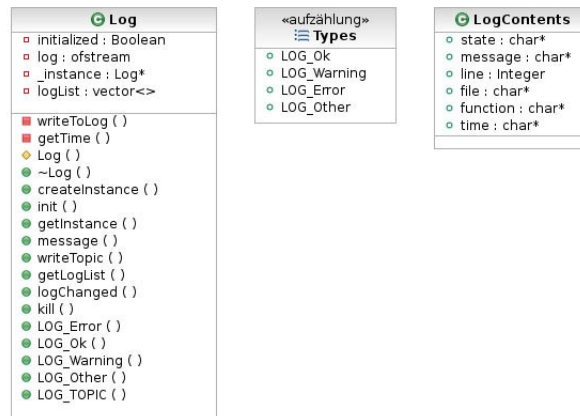
- **Core:** Beinhaltet (bisher nur) den Logger.
- **Math:** Enthält alle wichtigen Elemente der linearen Algebra sowie Hilfskonstrukte für die 3D-Programmierung um sie für die Benutzer komfortabel zur Verfügung zu stellen.
- **Renderer:** Die Vermittlungsstelle zwischen OpenGL und den anderen Blacksun-Modulen. Die Hauptaufgabe ist es die Rendereaufträge an OpenGL weiter zu leiten und wichtige Daten wie Rendereaufträge, Texturen und Materialien zu verwalten.
- **PluginSystem:** Verantwortlich für die Installation, Verwaltung und Kommunikation der Plugins.
- **SceneGraph:** Verwaltung der 3D-Modellierungsdaten zur internen Speicherung
- **GUI:** Die Benutzeroberfläche des Editors

Die Module werden in den folgenden Kapiteln im Detail erklärt.

2 Core/Logger

Der Editor enthält einen Logger um alle wichtigen Programmabläufe zu dokumentieren, die im Fehlerfall aufschluss auf die Fehlerquelle geben können. Die Einträge werden in einer Datei festgehalten und können über die GUI ausgegeben werden.

2.1 UML



2.2 Funktionale und nicht funktionale Anforderungen

Der Logger muss folgende Anforderungen erfüllen:

- Um eine umständliche Verwendung zu vermeiden, sind die Aufrufe für Logdatei-Einträge einfach gehalten.
- In jedem Log-Eintrag ist die Zeilennummer, die Datei sowie die Funktion in welcher der Eintrag vorgenommen wurde miteinzufügen.
- Jedem Eintrag kann eine individuell zur Situation passende Bemerkung hinzugefügt werden.
- Das Logging erfolgt sowohl in eine Html-Datei als auch in eine interne Liste, die von der GUI ausgelesen wird.
- Die Formatierung erfolgt automatisch.

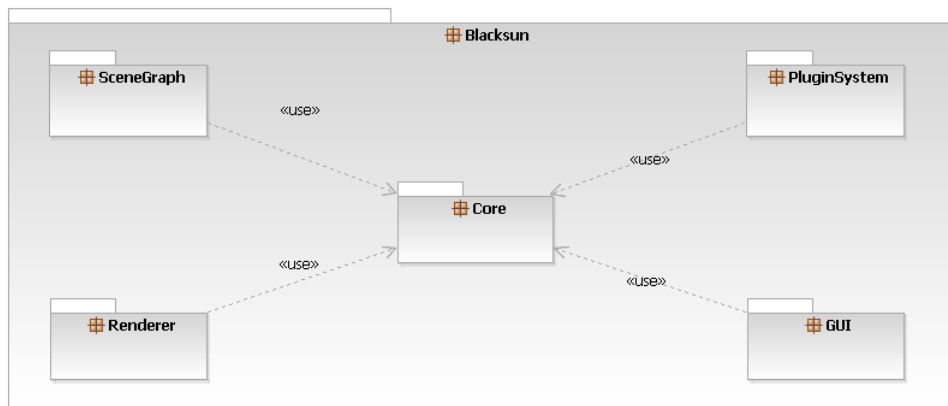
2.3 Mengengerüste

In diesem Module werden Daten in eine Datei geschrieben und in eine Liste gespeichert. Es ist wichtig dass alle vorhandenen Daten in die Datei geschrieben werden. Daher wird der Schreibbuffer immer geleert.

2.4 Architektursicht – Gesamtsicht

Das Log-Modul ist unabhängig von anderen Modulen. Um Log-Einträge machen zu können wird einfach die Log-Klasse in dem entsprechenden Modul/Plugin inkludiert.

Das folgende Paketdiagramm zeigt die Integration des Core-Moduls:



2.5 Architektursicht – Klassensicht

Die Log-Klasse enthält die Struktur „LogContents“. Diese bildet das Abbild eines Logeintrags der in einer Liste abgelegt wird. Die GUI liest die Einträge dann aus dieser Liste aus.

2.6 Architektursicht – Laufzeit

Die Benutzung des Loggers wird am Beispiel eines einzutragenden Fehlers demonstriert:

```
LOG_ERROR("Can't open file");
```

2.7 Schnittstellen nach außen

Der Logger ist die einzige Schnittstelle in diesem Modul (siehe oben).

2.8 Einflussfaktoren und Randbedingungen

Um die Aufrufe für die Logeinträge einfach zu halten wurden folgende Defines verwendet. Dies werden eingesetzt um dem User den Aufruf der „Lokalisierungs“-Funktionen zu ersparen, da diese bei jedem Eintrag die selben sind.

- LOG_Error(details) für message(LOG_Error, details, __LINE__, __FILE__, __FUNCTION__)
- LOG_Ok(details) für message(LOG_Ok, details, __LINE__, __FILE__, __FUNCTION__)
- LOG_Warning(details) für message(LOG_Warning, details, __LINE__, __FILE__, __FUNCTION__)
- LOG_Other(details) für message(LOG_Other, details, __LINE__, __FILE__, __FUNCTION__)

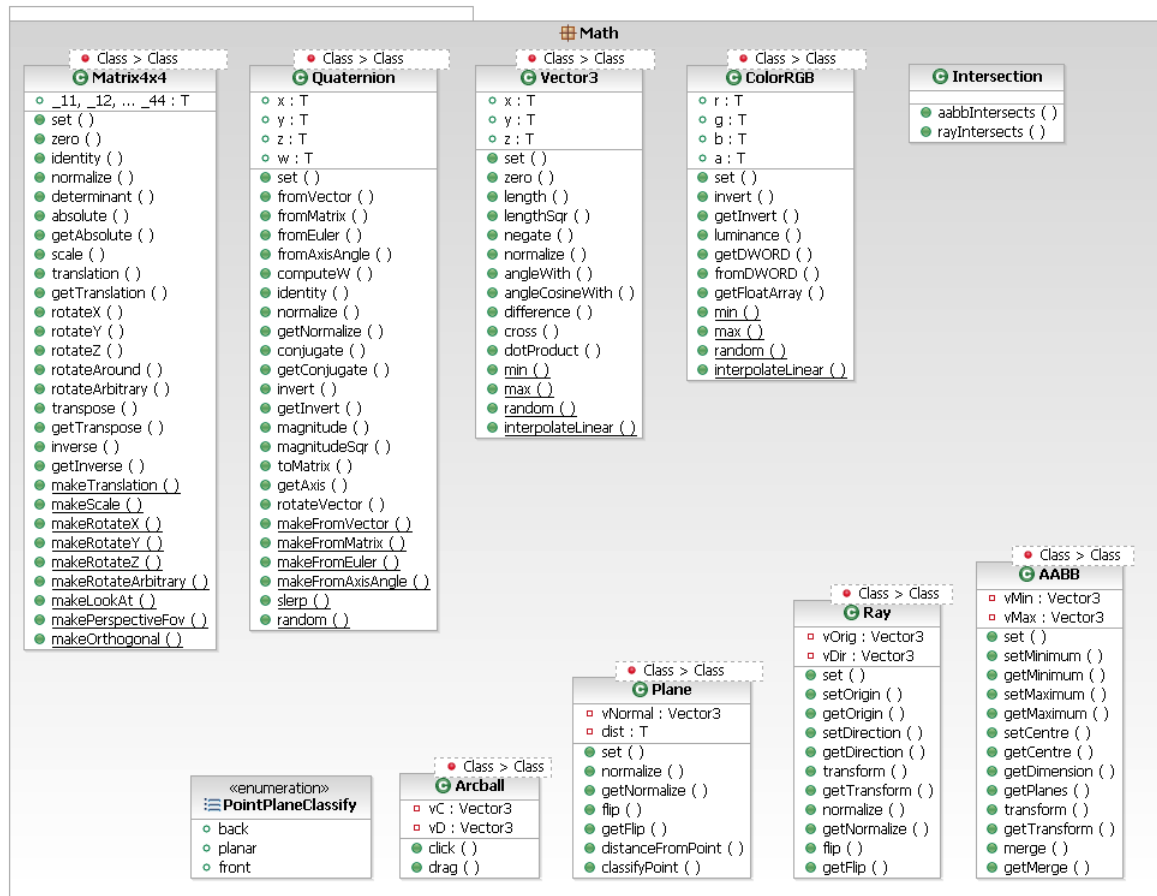
2.9 Globale Entwurfsentscheidungen

Der Logger ist ein eigenständiges Modul des Kerns von Blacksun. Da das Logging während der gesamten Programmlaufzeit ausgeführt wird ist das Modul in das Programm integriert und nicht als Plugin realisiert.

3 Math

Um die für die 3D-Manipulationen notwendigen Operationen für den Benutzer komfortable nutzbar zu machen, enthält der Editor eine eigene Mathe-Bibliothek. Sie enthält alle wichtigen Elemente der linearen Algebra sowie Hilfskonstrukte für die 3D-Programmierung.

3.1 UML



3.2 Funktionale und nicht funktionale Anforderungen

Die Mathe-Bibliothek muss folgende Anforderungen erfüllen:

- Um die Mathebibliothek in allen Blacksun-Modulen verwenden zu können, haben die Klassen eine breit gefächerte Funktionalität.
- Die Klassen sind nicht auf einen festen Typen festgelegt, sondern sind vollkommen typunabhängig.
- Für gängige kleinere mathematische Probleme werden Util-Funktionen angeboten. Darunter fallen z.B. der Vergleich zweier Zahlen auf Gleichheit mit bestimmter Toleranz, die Umwandlung Grad->Bogenmaß oder Bogenmaß->Grad.
- Die verwendeten Algorithmen müssen sehr performant sein, da einige der Klassen elementarer Bestandteil anderer Module sind und manche Operationen sehr häufig benutzt werden (z.B. Vector*Matrix zur Transformierung von 3D-Daten).
- Im Falle von Fehlern (z.B. Division durch 0) wird die betroffene Funktion eine Assertion werfen, die dem Programmierer die Fehlerquelle aufzeigt.

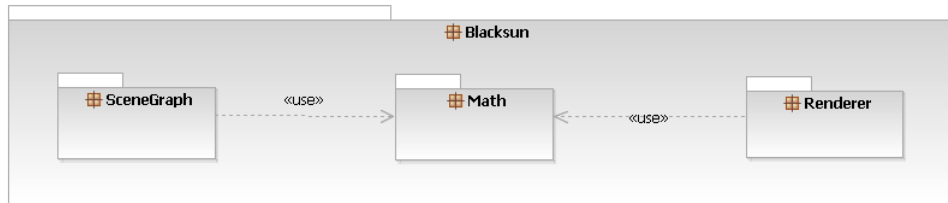
3.3 Mengengerüste

Es werden in diesem Modul keine Daten gespeichert. Die Aufrufhäufigkeit einiger Klassen ist sehr hoch (z.B. Vektoren und Matrizen). Daher ist Wert auf Schnelligkeit gelegt worden.

3.4 Architektursicht – Gesamtsicht

Das Mathe-Modul von Blacksun ist unabhängig von allen anderen Modulen. Jedes Modul/Plugin, das die Mathebibliothek benötigt, inkludiert einfach die gewünschte Klasse (siehe 3.9).

Das folgende Paketdiagramm zeigt die Integration der Mathebibliothek:



3.5 Architektursicht – Klassensicht

Es kann zwischen zwei Kategorien unterschieden werden:

- Klassen für Elemente der linearen Algebra
- Hilfsklassen für die 3D-Programmierung

Zu den Elementen der linearen Algebra gehören:

- **Matrix4x4**: Eine auf OpenGL zugeschnittene Matrix-Klasse. Bietet unter anderem Funktionen für 3D-Projektionen, Rotation und Translation. Da OpenGL mit 4x4-Matrizen rechnet, ist diese Klasse auf Schnelligkeit ausgelegt, in dem alle Schleifen 'abgerollt' sind. Dies vermeidet einen Overhead bei den Schleifen, da im Editor mehrere Tausend Vektoren transformiert werden ($\text{Vector} * \text{Matrix}$). Die Matrix-Klasse ist so ausgelegt, dass sie als Eingabeparameter für OpenGL-Funktionen verwendet werden kann, ohne eine zuvorige Konvertierung.
- **Vector3**: Ein dreidimensionaler Vektor, der beispielsweise für Positionen und Richtungen verwendet werden kann. Wie die Matrix-Klasse kann auch diese Klasse als direkter Eingabeparameter für OpenGL-Funktionen genutzt werden.
- **ColorRGB**: Farbwert im RGB-Raum. Enthält außerdem eine Alpha-Komponente um Transparenz zu ermöglichen. Auch diese Klasse kann direkt in OpenGL-Funktionen verwendet werden.
- **Quaternion**: Eine Quaternion, die eine rechnerisch elegante Beschreibung des dreidimensionalen Raumes erlaubt, insbesondere von Drehungen.

Die zweite Kategorie sind die Hilfskonstrukte. Hierzu werden diese Klassen angeboten:

- **Plane**: Beschreibt mithilfe eines zweidimensionalen Vektorraums eine euklidische Ebene.
- **Ray**: Ein Strahl ohne Länge der sich von einem Anfangspunkt aus in eine bestimmte Richtung erstreckt.
- **AABB**: Eine achsenorientierte Begrenzungsbox (engl. **A**xis-**A**ligned **B**ounding **B**ox). Sie wird beispielsweise verwendet, um nicht sichtbare Objekte vom Rendern auszuschließen.
- **Arcball**: Wandelt eine Bewegung der Maus in eine Rotations-Quaternion um.
- **Intersection**: Berechnet, ob sich zwei 3D-Objekte schneiden (z.B. AABB-AABB-Schnitt)

3.6 Architektursicht – Laufzeit

Die Verwendung des Moduls wird am Beispiel eines zu rotierenden Vektors demonstriert:

```
//Rotiere Vektor um 90 Grad um Y-Achse und normalisiere ihn danach
Vector vRot(0.1, 0.0, 0.0);
Vector vRes = vRot * Matrix::rotateY(degToRad(90.0));
vRes.normalize();
```

3.7 Schnittstellen nach außen

Besitzt keine, die Klassen sind die Schnittstellen zu diesem Modul.

3.8 Einflussfaktoren und Randbedingungen

Da die Matheklassen am häufigsten mit `double` als Templateparameter verwendet werden, enthält das Modul zusätzlich noch Typedefs für die am meist genutzten Klassen. Dies macht das Schreiben des Template-Typs für `double` nicht mehr notwendig. Folgende Typedefs sind definiert:

- `Matrix` für `Matrix4x4<double>`
- `Vector` für `Vector3<double>`
- `Quat` für `Quaternion<double>`
- `Aabb` für `AABB<double>`
- `Color` für `ColorRGB<double>`

3.9 Globale Entwurfsentscheidungen

Da die Mathebibliothek nur aus Template-Klassen und -Funktionen besteht, wird diese nicht als statische oder dynamische Bibliothek kompiliert. Es reicht ein einfaches includieren der benötigten Klassen um den Modulbaustein zu nutzen.

4.2 Funktionale und nicht funktionale Anforderungen

Der Renderer muss folgende Anforderungen erfüllen:

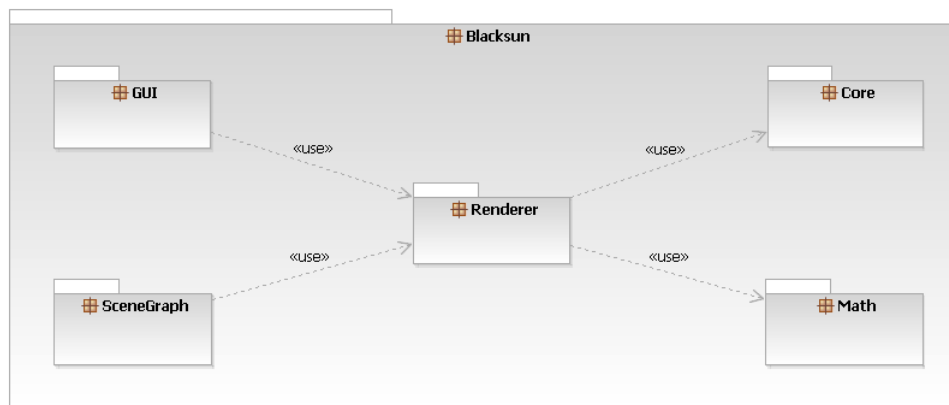
- Es müssen Renderaufträge angenommen werden können. Diese können Triangles (Dreiecke), Linien und Punkte sein.
- Um die Performance beim Rendern zu steigern, werden die Renderaufträge zu möglichst großen Paketen zusammengefasst. Der Grund ist, dass das Senden der 3D-Daten an die Grafikkarte über OpenGL immer über den Grafikkbus geht (z.B. AGB-Bus), der jedoch den Bottleneck bei Grafikkarten darstellt. Der Render muss daher Daten sammeln um nicht zu viele Aufrufe über den Bus zu tätigen.
- Texturen - die Bilddateien die über Modelle gelegt werden - sollen geladen werden können. Um den Ladevorgang zu beschleunigen und Speicher zu sparen, sollen zudem die Texturen verwaltet werden um ein doppeltes Laden einer Datei zu verhindern.
- Um neue Bilddatei-Typen (z.B. PNG) als Textur laden zu können, muss man neue Laderoutinen einfügen können. Diese Laderoutinen können beispielsweise über Plugins dem Editor eingefügt werden (siehe dazu Kapitel über Pluginsystem und Pluginentwicklung)
- Der Benutzer soll Materialien erstellen, löschen und ändern können. Ein Material beschreibt die Oberflächeneigenschaften und vereint materialspezifische Eigenschaften (z.B. Farben), eine Menge von Texturen und die dazugehörigen Kombinationseinstellungen derjenigen Texturen.
- Eine komfortable Technik zum Modellieren soll direkt im Renderer umgesetzt sein, nämlich das Realtime-Mirroring. Es spiegelt die Szene ohne die Generierung neuer 3D-Rohdaten in der Szene. Dies hat den Vorteil, dass die Modellierung von symmetrischen Modellen (z.B. ein Gesicht) einfacher wird, da es nicht mehr bei jeder Änderung manuell von Hand gespiegelt werden muss, sondern automatisch geschieht.

4.3 Mengengerüste

- Im Modul wird immer der aktuelle Zustand gespeichert. Darunter fallen beispielsweise Darstellungseigenschaften. Diese Eigenschaften müssen für jedes im Editor dargestellte Fenster vor dem Rendern neu gesetzt werden, da es nur einen Renderer für beliebig viele Ansicht-Fenster gibt.
- Renderaufträge, also die zu rendernden 3D-Daten, werden so lange gespeichert, bis alle Szenen-Rohdaten (eine Menge an Vertices) zusammengesammelt sind. Danach werden alle diese Daten zur Grafikkarte geschickt.
- Je nach Szene und Einstellungen werden im Worst-Case alle Rohdaten bis zum entgeltigen Senden an die Grafikkarte im Speicher gehalten. Pro zu rendernden Vertex werden 112 Byte Speicher benötigt (ca. 9300 Vertices/MB).
- Texturen/Materialien werden dynamisch zur Laufzeit geladen/erstellt und gelöscht. Je geladene Datei wird genau eine Textur im Speicher geladen (kein doppeltes Laden), es sei denn, sie wurde zwischendurch gelöscht.

4.4 Architektursicht – Gesamtsicht

Das folgende Paketdiagramm zeigt die Integration des Renderer-Moduls:



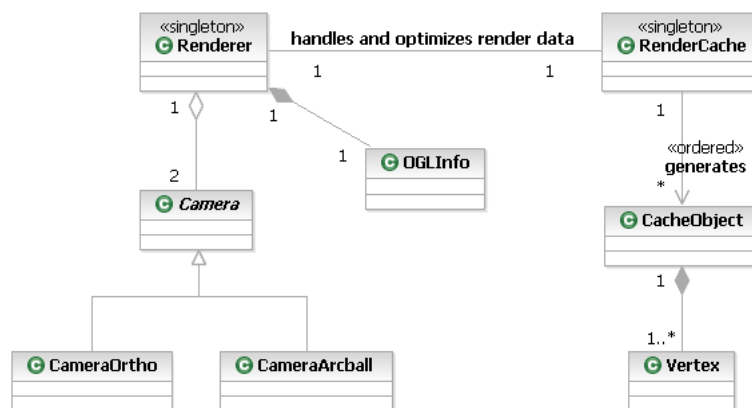
4.5 Architektursicht – Klassensicht

Das Renderer-Modul enthält drei Untersysteme:

- **Der eigentliche Renderer:** Dieser Teil ist zuständig die 3D-Rohdaten zu verwalten und diese performant an die Grafikkarte zu schicken. Zudem enthält er die Kamera-Klassen die für die verschiedenen Ansichten notwendig sind.
- **Textur- und Material-System:** Speichert und verwaltet alle Texturen, Materialien und Laderoutinen, welche zusammen die Oberflächen von Modellen/Szenen beschreiben.
- **Zentrale Konfigurationsschnittstelle:** Bietet zentrale Klasse zum Zugriff auf alle Einstellungen des Renderer-Moduls.

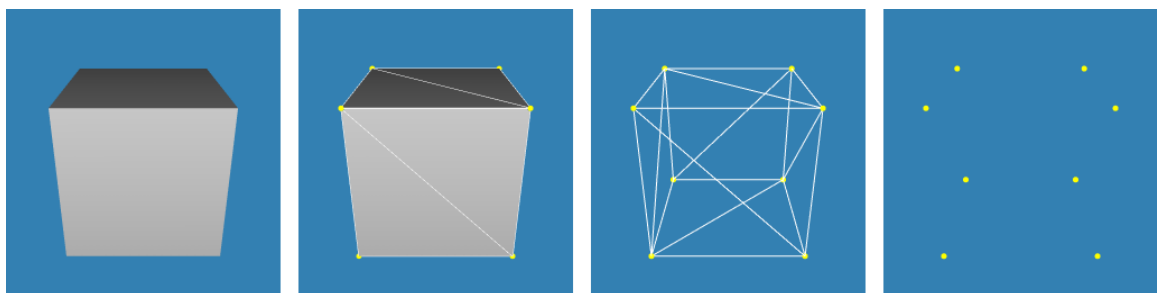
Diese drei Untersysteme sind im Folgenden im Detail beschrieben:

4.5.1 Eigentliches Renderer-System



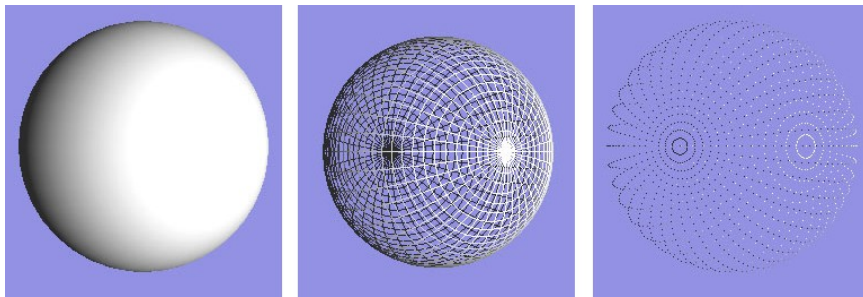
Folgende Klassen /Strukturen sind beteiligt:

- **Renderer:** Stellt die Schnittstelle nach außen dar. Der Renderer kann hier initialisiert werden und Informationen über die Grafikkarte und ähnliches abgerufen werden (siehe OGLInfo). Das wichtigste sind die Funktionen zum Rendern. Da im Editor zwischen selektierten, gesperrten und normalen Daten unterschieden wird, gibt es für jede dieser Arten eine Rendermethode. Gerendert werden können Dreiecke, Linien und einzelne Punkte. Außerdem enthält das Renderer-Singleton Methoden zum Setzen von Renderer-Einstellungen. So gibt es eine Methode zum Einstellen, ob das zu rendernde Fenster eine orthogonale Ansicht ist oder eine perspektivische (je nachdem wird eine entsprechende Kamera gewählt, siehe Camera). Außerdem kann der Füllmodus (Fillmode) eingestellt werden. Zur Verfügung stehen das Rendern als texturierte (Textured) oder untexturierte Szene (Solid), als Drahtgittermodell (Wireframe), als Punktmenge (Point) oder als texturierte Szene mit darüber gelegtem Drahtgittermodell (WireframeOverlay). Die Render-Methoden erstellen aus diesen Einstellungen die Renderaufträge, die an den RenderCache weitergeleitet werden. Dabei erstellen sie auch automatisch eigene Renderaufträge um RealtimeMirror zur Verfügung zu stellen bzw. wenn die Normalen eines Modells visuell dargestellt werden sollen. Zusätzlich neben dem Rendern bietet die Renderer-Klasse Zugriff auf den TextureManager, MaterialManager und die zentrale Konfigurations-Klasse des Renderer-Moduls.



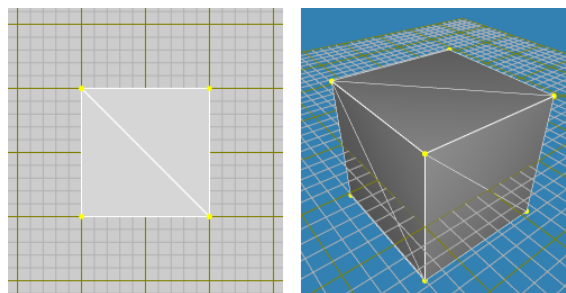
Füllmodi am Beispiel einer Box: Solid, WireframeOverlay, Wireframe, Point

- **OGLInfo:** Enthält Informationen über die OpenGL-Version, Treiber-Name, Name der Grafikkarte und einen String der alle auf der Grafikkarte verfügbare OpenGL-Extensions enthält. Jede Extension ist durch ein Leerzeichen getrennt.
- **RenderCache:** Ist zuständig für das Verarbeiten der Rendereaufträge vom Renderer-Singleton. Hierbei werden alle zu rendernden Daten gesammelt und in Pakete einer bestimmten Größe geteilt. Sind die Daten komplett, werden alle Pakete (CacheObject-Instanzen) zur Grafikkarte zum entgeltigen Rendervorgang geschickt. Die Bildung von Paketen hat den Vorteil, das nicht zu oft auf den Grafikkartenbus Daten übertragen werden, da der Bus den Bottleneck darstellt. Einige große Datenmengen zu senden ist daher performanter als viele kleine Mengen zu senden. Der RenderCache sorgt außerdem dafür, dass transparente so an die Grafikkarte geschickt werden, dass ein bestmögliches Ergebnis erzielt wird (möglichst wenig Artefakte, ...).
- **CacheObject:** Bildet ein Paket des RenderCache mit einer Menge an Vertices (die 3D-Daten) und charakteristischen Eigenschaften dieses Pakets. Dazu zählen die Material-ID, die Transformationsmatrix welche von OpenGL benötigt wird und den Primitive-Typ. Dieser Typ bestimmt, ob die Vertices als Dreiecke, Linien oder als Punktmenge interpretiert und gerendert werden sollen. Zusätzlich wird noch ein sogenannter Polygon-Typ gespeichert, mit dessen Hilfe die Art des Renderns beschrieben wird. So kann z.B. ein Dreieck entweder gefüllt (Filled), als Drahtgittermodell (Wireframe) oder als Punktmenge gerendert werden (Point). Dies kann natürlich auch auf auf die Primitiven Linie und Punkt angewendet werden.



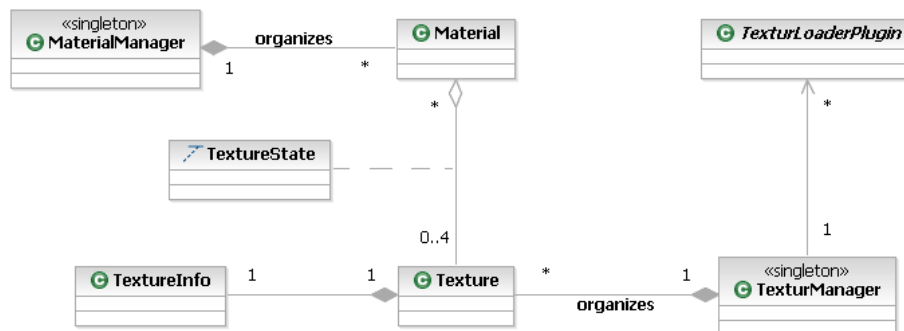
Polygon-Typ am Beispiel einer Kugel: Filled, Wireframe, Point

- **Camera:** Die abstrakte Klasse für die konkreten Kamera-Klassen, die für die Ansichten der Szene verwendet wird. Sie stellt unter anderem Setter und Getter auf zwei für OpenGL wichtige Matrizen zur Verfügung. Eine davon ist die Projektionsmatrix, die definiert, wie die 3D-Daten auf die 2D-Bildschirmenebene projiziert werden sollen. Die zweite Matrix ist die View-Matrix, welche die Kameraposition und -rotation speichert. Zudem bietet die Klasse Methoden zur Mausinteraktion und zu Fenstereinstellungen (Einstellen der Fensterdimension die in die Projektionsmatrix rein gerechnet werden muss).
- **CameraOrtho:** Eine konkrete Kamera-Klasse, die eine Kamera für die orthogonale 2D-Ansicht darstellt. Sie lässt sich auf die 6 verschiedenen Ansichten (Front, Top, Right, Back, Bottom, Right) umstellen. Die Mausinteraktion bewirkt eine Verschiebung der Kamera bzw. einen Zoom.
- **CameraArcball:** Eine konkrete Kamera-Klasse, die eine Kamera für die perspektivische 3D-Ansicht darstellt. Die Mausinteraktion bewirkt ein Bewegen und/oder Rotieren der Kamera um das Szenenzentrum bzw. einen Zoom (Entfernung der Kamera vom Zentrum). Da Objekt kann dabei per Maus 'angefasst' werden, der 'Anfasspunkt' bleibt immer unter der Maus. Die Szene lässt sich damit intuitiv drehen.



Kamera-Typen: Orthogonal und Perspektivisch

4.5.2 Textur- und Material-System

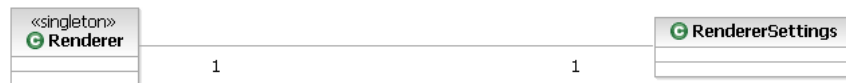


Folgende Klassen /Strukturen sind beteiligt:

- **Texture:** Ein vom Textur-Manager verwaltetes Textur-Objekt. Die gespeicherten Informationen (siehe **TextureInfo**) dienen zur eindeutigen Identifizierung der Textur, sodass Duplikate vermieden werden (speicherschonend). Die Textur speichert außerdem eine ID, mit deren Hilfe OpenGL diese Textur setzen kann. Diese ID ist allerdings nicht die ID über die man via **TextureManager** auf die Textur zugreifen kann. Sie dient lediglich OpenGL.
- **TextureInfo:** Beschreibt Eigenschaften einer Textur. Darunter fallen der Name, die Dimension und ein Flag das angibt ob die Textur Transparenz enthält.
- **TextureManager:** Verwaltet alle geladenen Texturen und Laderoutinen (siehe **TextureLoaderPlugin**). Beim Laden einer Textur wird geprüft, ob das Objekt nicht bereits existiert. Falls dies der Fall ist, wird die ID dieser Textur zurückgegeben, über das auf die Textur zugegriffen werden kann. Andernfalls lädt es, bei existieren einer geeigneten Laderoutine, die Textur und speichert sie mit allen charakteristischen Eigenschaften ab (siehe **TextureInfo**). Dieses Vorgehen hat den Vorteil, das ein mehrfaches Laden einer Textur verhindert wird und somit der Speicher geschont und Ladevorgang verkürzt wird. Um ein neues Dateiformat als Textur laden zu können, enthält der **TextureManager** eine Methode zum Hinzufügen neuer Laderoutinen. Eine solche Laderoutine lädt alle Dateien einer bestimmten Dateiendung. Im **TextureManager** sind nicht mehrere Laderoutinen für einen Dateityp erlaubt, die Laderoutine muss eindeutig sein.
- **TextureLoaderPlugin:** Die abstrakte Klasse für alle Textur-Laderoutinen. Eine solche Laderoutine ist immer für Dateien eines bestimmten Typs zuständig, wie z.B. BMP oder TGA. Aufgabe ist es, die Bilddaten auszulesen und diese zusammen mit der **TextureInfo** (siehe weiter oben) zurückzugeben. Geladen werden können Texturen mit und ohne Transparenz. Im **TextureManager** sind nicht mehrere Laderoutinen für einen Dateityp erlaubt, die Laderoutine muss eindeutig sein.
- **Material:** Beschreibt die Eigenschaften einer Oberfläche. Eine Oberfläche lässt sich durch spezifische Farben, Faktoren und Texturen definieren. Spezifische Farben sind zum einen die Grundfarbe, sowie Farben die mit der Lichtquelle zusammenhängen. Diese geben die Farbe bei ambienten und diffusen Lichtanteil an, die Glanzfarbe (Specular color) und die emittierende Farbe. Der Glanz eines Materials lässt sich über den Glanzfaktor (Specular factor) festlegen. Um dem Material ihr Aussehen zu geben, können Texturen verwendet werden. Texturen sind Bilddateien die über die 3D-Modelle gelegt werden. Wie diese Texturen kombiniert werden, wird in dem dazu gehörigen **TextureState** gespeichert. Die Anzahl der maximal zu kombinierenden Texturen ist für ein Material limitiert (bisher auf 4).
- **TextureState:** Beschreibt wie zwei Texturen miteinander in OpenGL kombiniert werden sollen. OpenGL bietet dazu die mächtige Technik des Multitexturing. Die per Multitexturing gegebenen Möglichkeiten lassen sich in **TextureState** beschreiben. Es lassen sich die Kombinationsart (z.B. Modulate, Add, ...), die Quellen für die Farbgumente und die Farbkomponenten für jede dieser Quellen einstellen (z.B. SrcColor, OneMinusSrcColor). Außerdem können die Farbwerte am Ende dieser Kombination skaliert werden. Möglich sind die Skalierungsfaktoren 1, 2 und 4. Auch die Art der Texturwiederholungen kann hier eingestellt werden. Zur Verfügung steht zum einen die direkte Wiederholung (Repeat), die Texturkoordinaten werden direkt verwendet. Texturkoordinaten größer als 1.0 oder kleiner als 0.0 führen zu einer Kachelung der Textur. Der zweite Wiederholungstyp ist Clamp, Texturkoordinaten die den Bereich [0, 1] überschreiten werden auf 0 bzw. 1 gesetzt. Eine Kachelung ist nicht möglich.

- **MaterialManager:** Alle Materialien werden hier verwaltet. Materialien können hier erstellt und geändert werden, Zugriff besteht über eine eindeutige Material-ID. Zusätzlich zu diesen benutzerdefinierten Materialien werden hier auch alle Standardmaterialien gespeichert. Solche Materialien sind zum Beispiel verfügbar für die Linien (Selektiert, Gesperrt, Normal) oder für die zusätzlich zu rendernden Normalen. Alle Standardmaterialien können zwar geändert werden, aber nicht gelöscht, da es reservierte Materialien sind. Material-IDs < 0 sind für Renderer-spezifische Materialien vorgesehen, alle anderen können vom Benutzer verwendet werden.

4.5.3 Zentrale Konfigurationsschnittstelle



Viele der im Renderer-Modul verwendeten Singletons haben eigene Eigenschaften, mit dessen Hilfe diese Klassen konfiguriert werden können. Damit die anderen Module den internen Aufbau des Renderer-Moduls nicht kennen müssen, wurde eine zentrale Konfigurationsklasse eingebaut. Alle Einstellungen können hier abgefragt und gesetzt werden. Die Konfigurationsklasse leitet alle Anfragen an die entsprechenden Klassen weiter. Die Kapselung hat den Vorteil, das die GUI für das Options-Menü nur mit dieser Klasse kommunizieren muss und eine Trennung von allgemein gültigen und fensterabhängigen Einstellungen (z.B. ViewMode) besteht. Wichtig: Die Konfigurationsschnittstelle ist erst im Renderer verfügbar, wenn dieser erfolgreich initialisiert wurde. Ansonsten gibt der Renderer NULL zurück!

4.6 Architektursicht – Laufzeit

Die Verwendung des Renderers und seinen Schnittstellen wird am Beispiel eines zu rendernden Dreiecks (mit Material und Textur) demonstriert:

```

//Speichern aller wichtigen Pointer
Renderer* r = Renderer::getInstance();
TextureManager* texMgr = r->getTextureManager();
MaterialManager* matMgr = r->getMaterialManager();

//Erstellen des zu rendernden Modells, hier ein Dreieck (Mit Normale und Texturkoordinaten)
vector<Vertex> triangle;
triangle.push_back(Vertex(Vector(-1.0, 1.0, 1.0)),Vector( 0.0, 0.0, 1.0), 0.0, 1.0));
triangle.push_back(Vertex(Vector( 1.0, -1.0, 1.0)),Vector( 0.0, 0.0, 1.0), 1.0, 0.0));
triangle.push_back(Vertex(Vector( 1.0, 1.0, 1.0)),Vector( 0.0, 0.0, 1.0), 1.0, 1.0));

//Material und Textur laden, Textur dann in Material einbinden
int nMat = matMgr->createNewMaterial();
int nTex = texMgr->loadTexture("Wood.bmp");
matMgr->getMaterial(nMat)->setTexture(0, nTex);

//Rendern des Dreiecks
r->render(PRIM_Triangle, triangle, nMat, nMat, AABB());
  
```

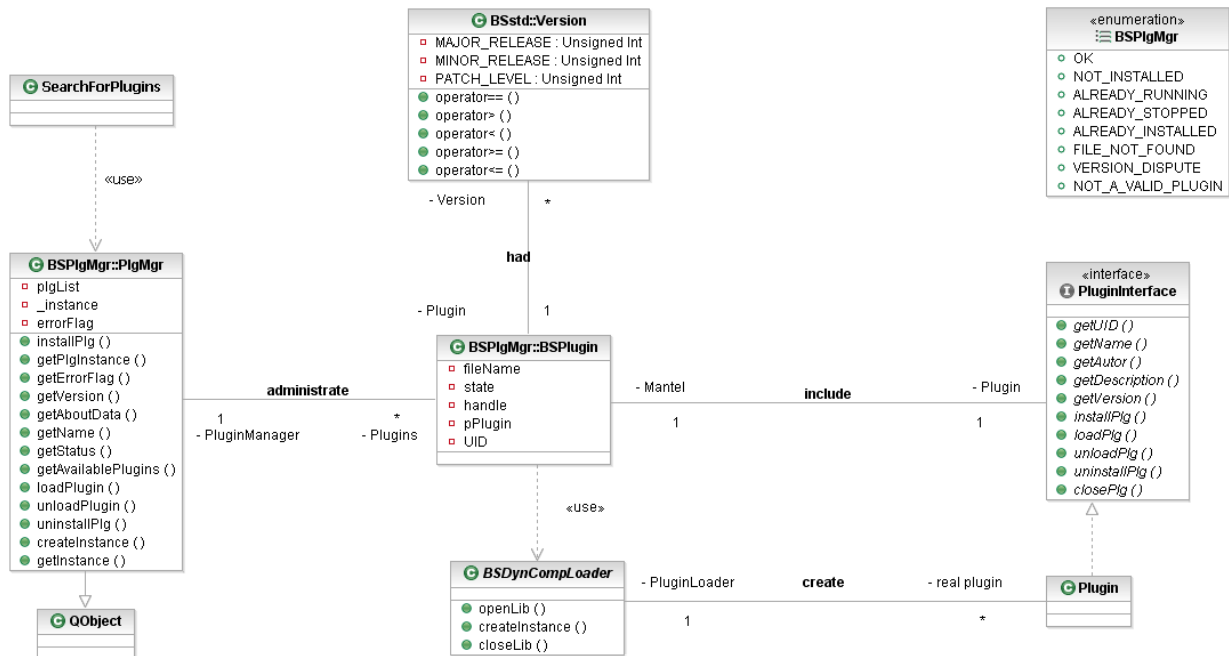
4.7 Schnittstellen nach außen

Die Schnittstelle für andere Module ist der Renderer-Singleton (siehe 4.5.1), welcher die Renderaufträge annimmt. Außerdem bietet er Zugriff auf den Textur-Manager, Material-Manager (siehe 4.5.2) und die zentrale Konfigurationsschnittstelle (siehe 4.5.3).

5 Pluginsystem

Dieses Modul ist verantwortlich für die Installation und Verwaltung der Plugins. Außerdem ermöglicht es den Zugriff auf die Installierten Plugins.

5.1 UML



5.2 Funktionale und nicht funktionale Anforderungen

- Plugins bringen nachträglich Funktionen in das Programm ein, ohne das dieses neu kompiliert werden muss.
- Plugins sind dynamische Bibliotheken (.so unter Linux, .dll unter Windows).
- Sie können auf die Funktionalität von anderen Plugins zugreifen.
- Das Pluginsystem kann neue Plugins installieren und auch wieder deinstallieren.
- Des weiteren sind Plugins entweder im Status 'geladen', also können vom Benutzer benutzt werden, oder im Status 'ungeladen'. Diese Plugins sind zwar dann installiert, aber können nicht benutzt werden.
- Andere Module bzw. Plugins können jederzeit:
 - ◆ feststellen, ob ein bestimmtes Plugin installiert ist.
 - ◆ sich einen Zeiger auf ein Plugin holen.
 - ◆ sich eine Liste aller verfügbaren Plugins holen.
 - ◆ den aktuellen Status eines Plugins anzeigen und ändern.
 - ◆ Plugins (de-) installieren.
- Die installieren Plugins und ihr Status werden beim nächsten Programmstart wiederhergestellt.
- Alle Plugins die in einen festgelegten Ordner sind (Standardmäßig ./Plugins) werden je nach Einstellung automatisch beim Programmstart Installiert und wenn gewünscht, geladen.

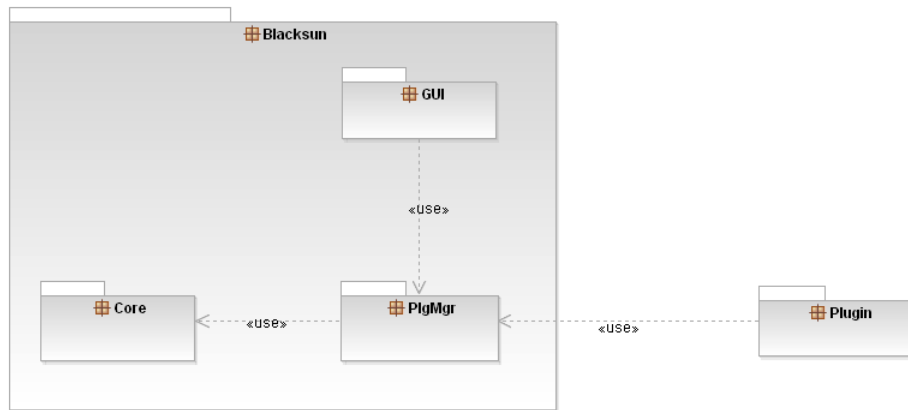
5.3 Mengengerüste

Da die Funktionen des Pluginsystem nicht oft benötigt werden (im normalen Betrieb werden Plugins nur einmal installiert und bei jedem Programmstart geladen), allerdings viel mit dynamischen Objekten gearbeitet wird, ist hier nicht die Geschwindigkeit ausschlaggebend, sondern die Fehlerbehandlung. So werden alle Funktionsparameter überprüft und alle möglichen Fälle abgefangen. Dies ist nötig, da die Funktionen nicht nur innerhalb des Programms sondern auch von Plugins aufgerufen werden.

Es fallen nicht viele Daten an: Pro Plugin wird nur ein Zeiger auf das Plugin, einer auf die Datei, der Status des Plugins und der Dateiname gespeichert.

5.4 Architektursicht – Gesamtsicht

Das Modul Pluginsystem ist ziemlich unabhängig von allen anderen Modulen. Es benötigt nur von den Kernkomponenten den Logger, um Meldungen auszugeben. Außerdem werden Signale über die Änderung der installierten Plugins an die GUI gesendet damit diese nicht die ganze Zeit den Status der Plugins überwachen muss, sondern nur dann informiert wird, wenn sich was ändert.



5.5 Architektursicht – Klassensicht

Die Klassen im Überblick:

- **PlgMgr:** Diese Klasse ist für die Verwaltung der Plugins zuständig. Für jedes installierte Plugin wird ein - das jeweilige Plugin repräsentierende - *BSPlugin* gespeichert. Alle Plugins können über ihre eindeutige UID (siehe 8.1) und ihre Version bestimmt werden. Zudem ist diese Klasse die Schnittstelle für alle Zugriffe auf das Plugin. Sie ermöglicht die unter 5.2 beschriebene
- **BSPlugin:** Die Objekte dieser Klasse repräsentieren jeweils ein Plugin. Im Konstruktor wird der Dateiname der zu öffnenden Bibliothek und der Startstatus angegeben. Im Konstruktor wird dann über die Funktionen von dem *DynCompLoader* versucht, die Bibliothek zu öffnen und einen Zeiger auf das eigentliche Plugin zu bekommen. Schlägt das fehl, wird stattdessen NULL als Zeiger gespeichert. Der Destruktor von *BSPlugin* gibt den Speicherplatz für das Plugin wieder frei und schließt die Bibliothek. Damit diese wieder geschlossen werden kann, wird auch der Zeiger auf die Bibliothek gespeichert.
- **DynComLoader:** Der *DynCompLoader* ist eigentlich keine Klasse, sondern nur eine Sammlung von Funktionen für das Laden, holen einer Instanz und schließen einer Bibliothek. Da es in jedem Betriebssystem unterschiedliche Funktionen gibt, prüft der Precompiler über `#ifdef WIN32` bzw. `#ifndef WIN32` das jeweilige Betriebssystem und setzt dann die jeweils richtigen Funktionen ein. Übersicht:

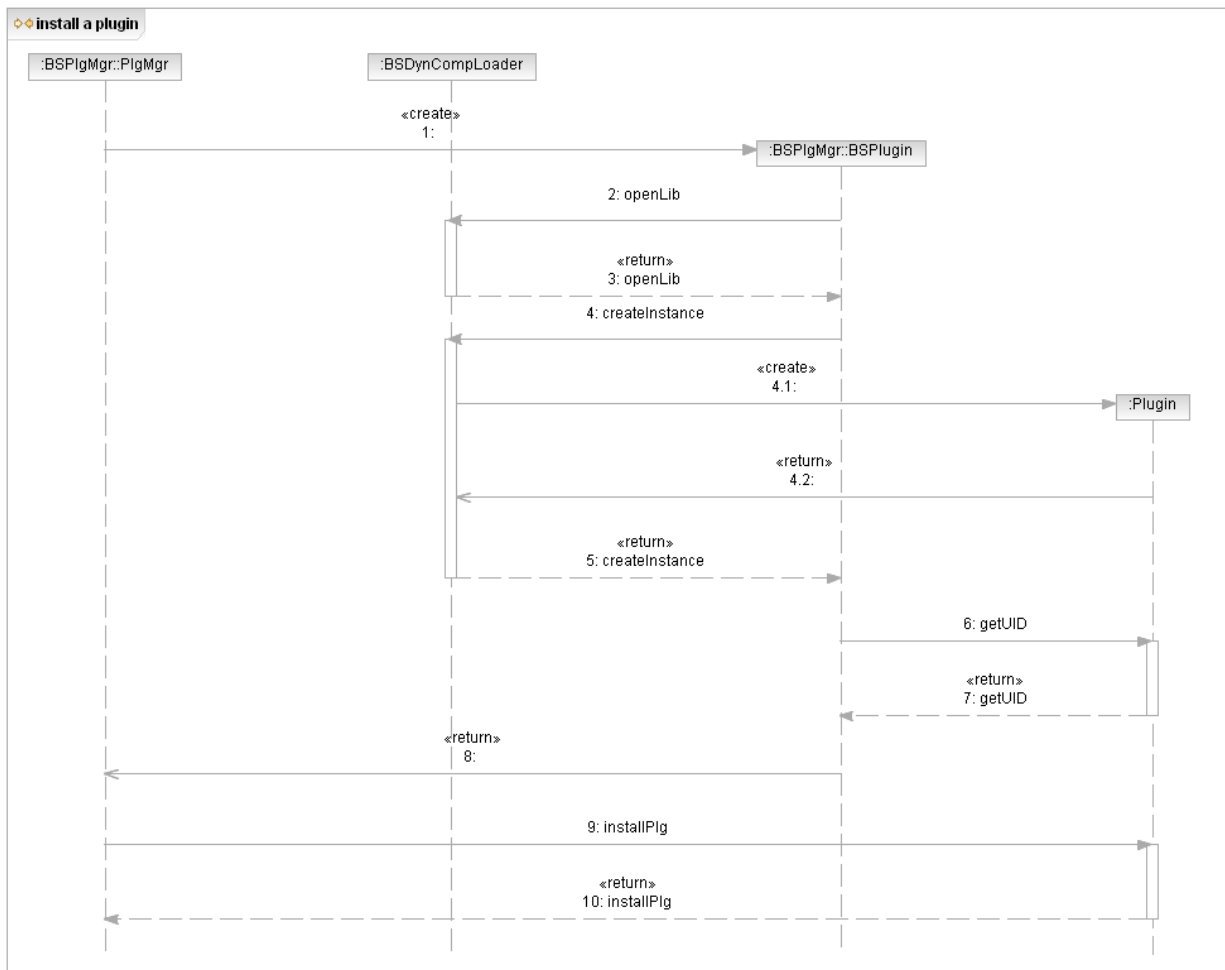
	Windows	Linux
	<code>#include <windows.h></code>	<code>#include <dlfcn.h></code>
<code>#define DCL_OPEN_LIB(a)</code>	<code>LoadLibrary((const WCHAR*) a)</code>	<code>dlopen(a, RTLD_LAZY)</code>
<code>#define DCL_GET_OBJECT</code>	<code>GetProcAddress</code>	<code>dlsym</code>
<code>#define DCL_CLOSE_LIB</code>	<code>FreeLibrary</code>	<code>dlclose</code>
<code>#define DCL_INSTANCE</code>	<code>HINSTANCE</code>	<code>void*</code>

Falls ein Fehler auftritt, wird NULL zurückgegeben. Die eingebaute Fehlerbehandlung von der *dlfcn.h* wird nicht benutzt, da diese in der *windows.h* nicht in diesem Umfang existiert. Deshalb werden im Vorfeld vom *PlgMgr* überprüft, ob die Datei existiert bzw. ob es eine gültige Bibliothek ist. Wird also NULL zurückgegeben, rührt das deshalb meistens davon, dass die Funktion extern "C" `PlgInt* createInstance()` in der Bibliothek nicht gefunden wurde.

- **SearchForPlugins:** Diese Klasse hat den Aufgabe, dass es den in den Optionen festgelegten Pfad nach zu installierende Plugins durchsucht, gefundenen Plugins dann mit den bereits installierten abgleicht und diese dann installiert und falls in den Optionen gewünscht, auch lädt.

5.6 Architektursicht – Laufzeit

Die folgenden Sequenzdiagramme zeigen den Ablauf einer Installation bzw. der Deinstallation eines Plugins:



5.7 Schnittstellen nach außen

Es gibt folgende Schnittstellen:

- **Informative:** Diese Schnittstellen der Form `getXxx(UUID)` sind Getter und liefern die gewünschten Informationen über das mit der UUID ausgewählte Plugin. Ist das Plugin nicht vorhanden, wird ein Defaultwert zurückgegeben und ein entsprechender Flag im Fehlerspeicher gesetzt.
- **Fehlerbehandlung:** Die Funktion `getErrorFlag()` liefert den aktuellen Fehlerspeicher zurück. Mit `clearErrorFlag()` wird der Fehlerspeicher auf 0x000 zurückgesetzt.
- **Steuerung:** Über die Funktionen `installPlg`, `uninstallPlg`, `loadPlugin` und `unloadPlugin` werden Plugins (de-) installiert und ihr Status geändert. Die jeweilige Funktion ruft außerdem die äquivalente Funktion des jeweiligen Plugins auf.
- **Singleton:** (siehe 13.3)

5.8 Einflussfaktoren und Randbedingungen

- Durch die Plattformunabhängigkeit wurden im DynCompLoader (siehe 5.5) die benötigten Funktionen auf den kleinsten gemeinsamen Nenner gebracht.

Bei der Auswahl der Technik mussten verschiedene Faktoren berücksichtigt werden:

- **Die Kommunikation zwischen den Plugins:**

Da es vorgesehen war, dass viele Funktionen mit Plugins realisiert werden sollte, war es klar, dass es eine flexible Kommunikation zwischen zwei Plugins geben muss. Am besten wäre es, wenn ein Plugin beliebig viele Funktionen des anderen aufrufen kann. Dabei können alle Funktionen beliebig viele Parameter jedes Typ (auch welche, die das Pluginsystem bzw. das Hauptprogramm gar nicht kennt) haben.

Ansatz: *Kommunikation über Interface:*

- Vorgehen: Es gibt ein Interface, das ein paar wichtige Funktionen, die jedes Plugin haben muss (z.B. Name), enthält. Von diesem werden alle Plugins abgeleitet. Ein Plugin, das auf Funktionen anderer zugreifen möchte, muss sich die von dem Interface abgeleitete Klasse (Noch besser ist ein 2. Interface) includieren. Dann kann es über den Zeiger auf das Plugin, den es vom Pluginmanager bekommt, direkt auf das Plugin und seine Funktionen zugreifen (siehe 8.1)
- Vorteil: Funktionen können ganz normal angesprochen werden
- Nachteil: Pluginmanager überwacht nicht die Zugriffe auf andere Plugins, das Plugin muss sich selbst darum kümmern, ob das benötigte Plugin nicht schon deinstalliert/entladen wurde
- **Informationen über andere Plugins:**
Ein Plugin, das andere aufruft, muss sicherstellen können, dass das andere Plugin noch nicht entladen/deinstalliert wurde (s.o.). Das heißt, Plugins müssen eindeutig sein. Das beinhaltet aber auch, dass Plugins ja auch weiterentwickelt werden, dass es eine Art Versionskontrolle geben muss.
Ansatz: *UID (unique ID):*
 - Vorgehen: Jedes Plugin hat eine eindeutige UID und eine Version (siehe 8.1)

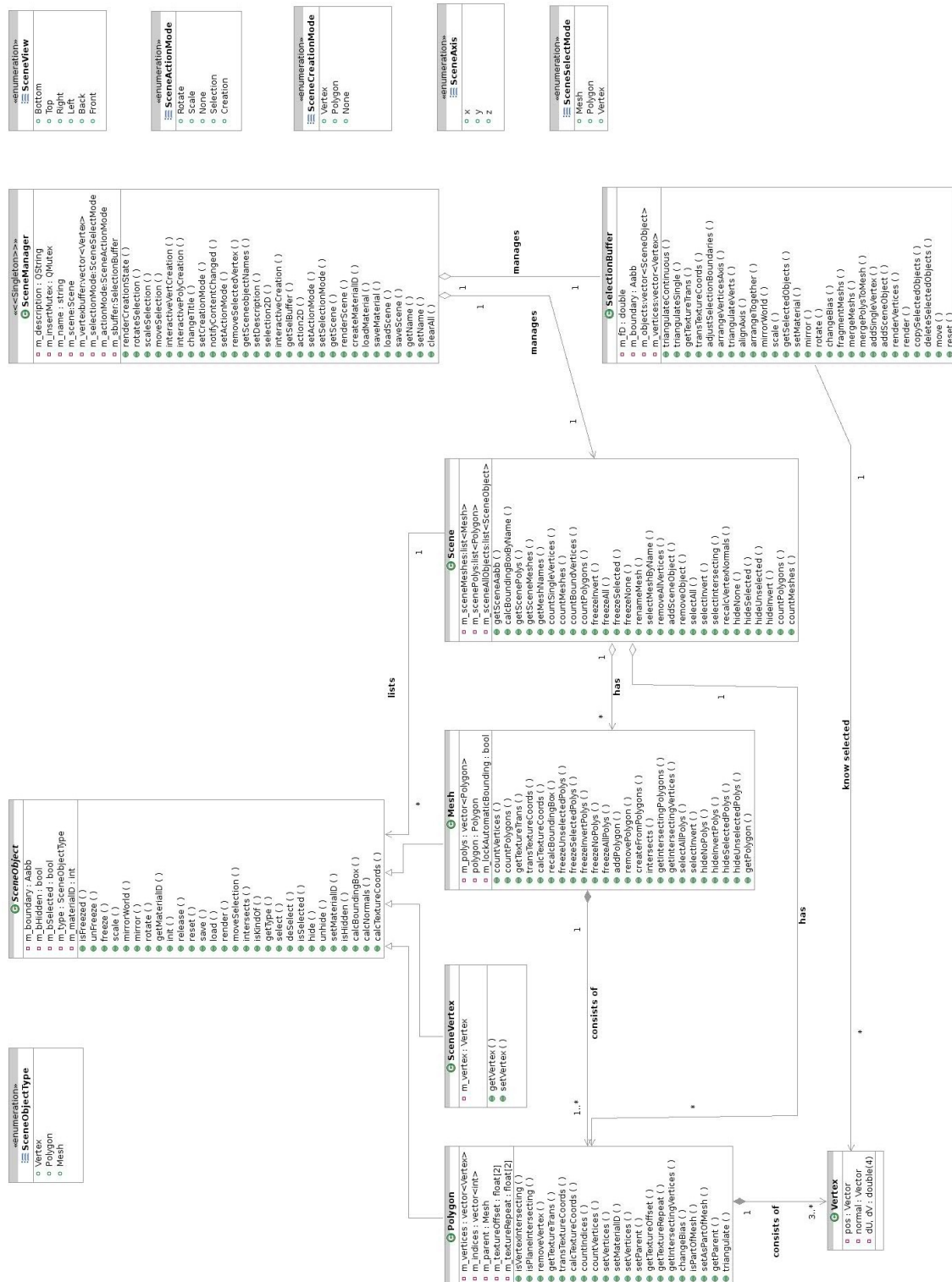
5.9 Globale Entwurfsentscheidungen

- Durch die Verwendung von QT wurde alles dem QT – System angepasst (QString, QMap und QList als Schnittstellendatentypen bei den Plugins statt STL-, C++ oder C-String, bzw. STL – Map und - List).

6 Scenegraph

Der Scenegraph ist eine auf den 3D-Editor „Blacksun“ zugeschnittene Datenstruktur. Hauptziel ist die Verwaltung von 3D-Modellierungsdaten zur internen Speicherung. Statische Modellierungselemente werden durch die GUI bzw. das Pluginssystem angelegt und anschließend im Scenegraph gespeichert. Der Scenegraph übergibt dann dem Renderer die darzustellenden Informationen.

6.1 UML



6.2 Funktionale und nicht funktionale Anforderungen

1.2.1 Grundsätzliche Anforderungen

- Verwaltung aller per Definition darstellbaren Szenenelemente, welche nach Typ unterschieden werden
 - ◆ Vertex (einzelner physischer Punkt, welcher durch 3 Raumpositionen bestimmt wird)
 - ◆ Polygon (entspricht semantisch einem durch 3 Punkte bestimmtem Dreieck)
 - ◆ Mesh (aus Polygonen zusammengesetztes Konstrukt)
- Der Scenegraph wird von den Komponenten GUI und Plugins mit Eingabedaten versorgt. Für diese muß eine passende und möglichst minimale Schnittstelle gegeben sein, damit das Einfügen schnell vonstatten geht
- Selektierte Szenenelemente müssen gesondert behandelt werden, da diese eine alternative Repräsentation sowie andere Bearbeitungsmöglichkeiten besitzen .
- Möglichkeit zum Laden und Speichern einer Szene in eine Datei zur späteren Weiterbearbeitung. Dies wird durch Selbstserialisierung jedes eigenen Szenenelements erreicht.
- Um Kollisionen zwischen Szenenelementen zu erkennen wird für jedes Objekt eine sogenannte Bounding Box mitgespeichert. Dies ist eine maximale Hülle eines 3D-Objektes und wird im Szenengraphen speziell zur Trefferprüfung bei Cursorselektionen verwendet.

1.2.2 Anforderungen betreffend Komponente GUI / Plugins

- Wenn sich ein beliebiges Detail am Szenenzustand geändert hat, ist die Funktion zur Neudarstellung des Renderers aufzurufen.
- Jedes Szenenobjekt muß über eine ID für das Material verfügen. Physikalisch werden Materialien im Renderer::MaterialManager gehalten, der Scenegraph kennt dagegen nur die vom MaterialManager vergebene ID.
- Die GUI muß Qt-Slots bereitstellen, falls sich ganz allgemein Meshes geändert haben oder auch gezielte Mesh-Änderungen stattfanden.

1.2.3 Anforderungen betreffend Komponente Renderer

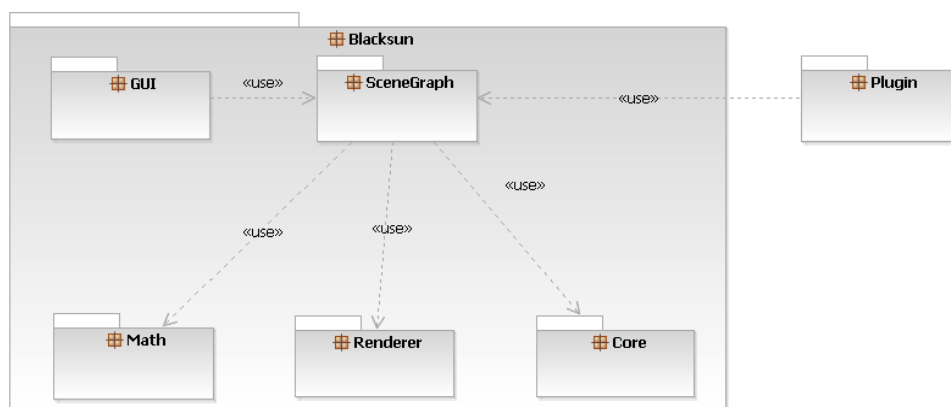
- Wenn sich ein beliebiges Detail am Szenenzustand geändert hat, ist die Funktion zur Neudarstellung des Renderers aufzurufen.
- Jedes Szenenobjekt muß über eine ID für das Material verfügen. Physikalisch werden Materialien im Renderer::TextureManager gehalten, der Scenegraph kennt dagegen nur die vom TexturManager vergebene ID. Um jedoch Materialänderungen durch den Benutzer zu erfassen
- Zugriffsfunktionen auf das Untermodul MaterialManager des Renderers sind vonnöten.

6.3 Mengengerüste

Einer Szene ist prinzipiell keine Grenze in Sachen Polygonanzahl gesetzt. Eine praktische Begrenzung findet entweder nur durch den Hauptspeicherausbau eines Anwenders statt oder einer Überforderung der Grafikkarte durch zuviele darzustellende 3D-Daten.

6.4 Architektursicht – Gesamtsicht

Das folgende Paketdiagramm zeigt die Integration des SceneGraph-Moduls:



6.5 Architektursicht – Klassensicht

Folgende Klassen sind beteiligt:

- **SceneManager:** Dieser stellt die Schnittstelle zu den anderen Fremdkomponenten dar. Die Klasse wurde als Singleton implementiert und existiert somit programmweit nur ein einziges Mal. Der SceneManager hält zudem jeweils eine Instanz der Klasse *Scene* und eine Instanz der Klasse *SelectionBuffer*, womit diese auch einmalig im Programm vorkommen. Und der SceneManager kümmert sich um den geregelten Zugriff auf diese beiden Module der Scenegraph-Komponente.

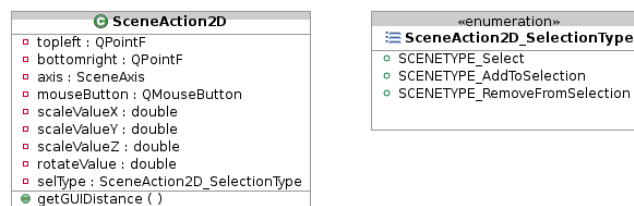
Wenn in der GUI eine Änderung durch den Anwender stattfand, ruft die GUI den SceneManager mit dem Befehl zum Neuzeichnen des Szeneninhaltes auf. Etwa wenn ein Ansichtsfenster verdeckt wurde und nach dem darauffolgenden Sichtbarmachen seinen Inhalt erneut wiedergeben muß.

Ein Großteil der Funktionalität kapselt die Logik des Erstellens und der Auswahl wieder.

Dazu speichert der SceneManager den aktuellen Zustand des Szenenaktionsmodus. Die Klasse GLWidget (siehe Paket GUI) gibt dazu bei Änderungen in der GUI diese Änderungen direkt an den SceneManager weiter. Wenn etwa in der Hauptwerkzengleiste der Selektionsmodus von Polygon auf Mesh geändert wurde, wird durch die Funktion *setSelectionMode* dies registriert.

Bei einem Mausklick in der GUI wird dieser dann ungefiltert an den SceneManager weitergegeben, der dann anhand seines Zustands die entsprechenden Aktionen durchführt.

Im Falle einer Selektion ist noch der Typ der Selektion von Interesse. Eine Selektion kann neben der klassischen Auswahl außerdem noch additiv oder subtraktiv gelten.



Klasse zur Übergabe der Selektionsparameter

Auch die Schnittstelle zum Laden und Speichern einer Szene hat der SceneManager inne. Dazu wird nach einem Aufruf einer Dateiaktion der Befehl an die Instanz der Klasse *Scene* weitergegeben, die wiederum alle enthaltenen Szenenobjekten zum Speichern bzw. Laden veranlasst. Das Speichern und Laden benutzt zudem Packerfunktionalitäten (zip), welche durch das Qt-Framework zur Verfügung gestellt werden.

Bei jedweden Änderungen an den Meshes einer Szene wird die GUI mithilfe von Qt-Signalen davon in Kenntnis gesetzt, da die GUI die Meshes im Objektexplorer verwaltet. Wird beispielsweise ein Mesh gelöscht, wird ein Signal *meshRemoved* mit dem Namen des Meshes als Parameter ausgelöst. Genauso verhält es sich im Fall des Hinzufügens eines Meshes.

- **SelectionBuffer:** Der SelectionBuffer verwaltet die Selektion von Szenenobjekten. Er merkt sich dazu die Verweise auf die tatsächlichen Instanzen der Objekte. Wenn etwa der Editorbenutzer in der Szene das Polygon X selektiert, so wird der SelectionBuffer in seiner Liste von selektierten Szenenobjekten, welche vom Typ *SceneObject* sind, die Adresse von Polygon X speichern.

Alle Funktionalität, die auf der Menge der selektierten Objekte basiert, wird über den SelectionBuffer abgewickelt:

- ◆ Allgemeine Editierungsmöglichkeiten: Einfügen/Kopieren/Ausschneiden/Löschen von Auswahlen
- ◆ Allgemeine Szenenänderungen: Verschieben/Rotieren/Skalieren von Auswahlen
- ◆ Spezielle Szenenänderungen: Materialien zuweisen oder verschieben/Vertices ausrichten

Sämtliche Modifikatoren-Plugins benötigen zudem die aktuelle Auswahl um darauf die jeweiligen Änderungen durchzuführen. Darum bietet der SelectionBuffer mehrere Getter-Methoden für die derzeitige Auswahl, je nach Typ.

Von entscheidender Bedeutung ist der eigene Aufruf des Renderers. Dies ist nötig, weil selektierte Szenenobjekt anders dargestellt werden sollen als unselektierte Objekte. Etwa mit einer dickeren und anderen Linienfarbe, was im allgemeinen Einstellungsdialog des Editors individuell änderbar ist.

Auch kapselt der SelectionBuffer die Funktionalität zur Umwandlung von Polygonen zu Meshes und vice versa. Der Anwender selektiert zum Beispiel eine bestimmte Anzahl von Polygonen im Ansichtsfenster und wählt dann den entsprechenden Menüpunkt in der GUI. Die inverse Operation dazu, vom Mesh wieder hin zu den Einzelpolygonen ist ebenfalls Teil der Implementation.

- **SceneObject:** Es gibt viele Methoden, die alle Objekten besitzen müssen, welche in eine Szene einfügbar sind. Daher wurden diese in eine virtuelle Basisklasse ausgelagert um eine uniforme Behandlung zu ermöglichen. Beispielsweise ist allen Szenenobjekten gemein, dass deren Position vom Anwender geändert werden kann. Auch soll jedes Szenenobjekt im Ansichtsfenster an- und abgewählt werden können um Modifikationen nur auf ausgewählte Objekte zu beschränken.

Alle Szenenobjekte die von SceneObject erben, müssen Funktionen für Verschieben, Rotieren, Skalieren und Spiegeln ausprogrammieren. Auch die Möglichkeiten zum Verstecken und Einfrieren von Polygonen müssen gegeben sein.

Desweiteren muß jedes konkret darstellbare Objekt, welches von SceneObject erbt, Funktionen zum Laden und Speichern implementieren. Somit ist ein Szenenobjekt selbst für seine Serialisierung verantwortlich, wie es sich für ein objektorientiertes Design empfiehlt. Ähnlich verhält es sich mit der *render*-Funktion, welche ebenfalls von jeder Unterklasse separat zu definieren ist.

An gemeinsamen Eigenschaften wäre da einerseits der Besitz eines umgebenden Bereiches, einer sogenannten Bounding Box (siehe Paket Math), zu nennen. Diese wird für Kollisionserkennung gebraucht und auch der Renderer-Komponente beim Rendering-Aufruf mitgegeben.

Auch verfügt jedes Szenenobjekt über eine eigene Identifikationsnummer für das zugewiesene Material.

Falls eine spätere Verarbeitung eine Unterscheidung zwischen verschiedenen Szenenobjekten zu treffen hat, ist eine Art Variantentest vonnöten. Deswegen hält ein Szenenobjekt noch die Information um welche Ausprägung es sich genau handelt. Die dazu nötigen Einträge finden sich in der Enumeration *SceneObjectType*, die bei Anlage einer neuen konkreten Ableitung zu ergänzen ist.

- **Scene:** Dabei handelt es sich um eine verwaltende Containerklasse für Objekte einer dargestellten Szene im 3D-Editor. Die meisten Methoden leiten die Aufrufe nur an die zugehörigen Szenenobjekte weiter und dienen größtenteils der Funktionalität zur Selektion und dem Verstecken von Objekten. Zum Hinzufügen und Entfernen von Szenenobjekten gibt es die Methoden *addSceneObject* und *removeSceneObject*.

Verwaltet wird diese Klasse vom SceneManager, der externe Aufrufe im allgemeinen zu Aufrufen der Klasse Scene transformiert.

Im Besonderen wird durch diese Klasse die konkreten Szenenmanipulationen Selektieren, Verbergen und Einfrieren auf oberster Ebene ausgeführt.

- **Mesh:** Die Klasse Mesh ist eine Organisationseinheit, welche dazu dient eine Gruppe von Polygonen dauerhaft zu verbinden. Ein Mesh besteht somit aus einer beliebigen Vielzahl von Polygonen, aber immer mindestens einem Polygon. Ein Mesh ohne zugeordnetes Polygon ist nicht vorgesehen. Die Polygone bleiben weiterhin einzeln selektier- und editierbar, die Klasse Mesh erleichtert nur deren Verwendung, indem etwa einfache Objekte der realen Welt aus verschiedenen Polygonen modelliert werden können.

Um eine Instanz der Klasse Mesh zu erzeugen, braucht man eine Menge von zuvor erstellten Polygonen die per Methodenaufruf von *createFromPolygons* übergeben werden. Nichtsdestotrotz ist es auch möglich, Polygone auch einzeln per *addPolygon* aufzunehmen. Für größere Mengen empfiehlt sich dennoch die erstgenannte Methode, weil hier die Bounding Box nur einmal nach Einfügen aller Polygone berechnet wird, was beim einzelnen Einfügen nicht der Fall wäre. Neben dem Einfügen ist natürlich auch das spätere Entfernen einzelner Polygone aus einem erstellten Mesh möglich.

Da Meshes von SceneObject erben, bieten sie ausprogrammierte Funktionen für Verschieben, Rotieren, Skalieren und Spiegeln an. Auch die Möglichkeiten zum Verstecken und Einfrieren von Meshes sind gegeben.

Das Laden und Speichern besteht wegen des serialisierten Ansatzes fast nur aus dem Durchlauf aller zugehörigen Polygone mit jeweils einem Aufruf für die jeweilige Dateiaktion. Hier zählt sich die Designentscheidung zur Selbstverwaltung der Szenenobjekte somit durch Einfachheit aus.

Jedes Mesh einer Szenerie wird eindeutig über dessen Namen gespeichert. Dieser Name kann vom Benutzer (Siehe Objektexplorer der Komponente GUI) auch abgeändert werden.

- **Polygon:** Polygon ist eine von SceneObject abgeleitete konkrete Klasse, welche ein aus Eckpunkten (=Vertices) zusammengesetztes Gebilde ist. Im Blacksun-Editor besteht jedes fertig konstruierte Polygon aus mindestens 3 Vertices.
Eine Szene setzt sich genau genommen nur aus einzelnen Polygonen oder Gruppen von Polygonen (siehe Klasse Mesh) zusammen.

Neben den reinen Vertexdaten muß ein Polygon sich auch noch Indize zu den Vertices merken. Diese dienen der Festlegung der Reihenfolge von Polygonen. Diese können nämlich entweder im oder gegen den Uhrzeigersinn angegeben sein, was wichtig für die korrekte Darstellung durch den Renderer ist. Die Indexreihenfolge kann der Anwender der Polygonklasse entweder selbst beim Erzeugen mitgeben oder die vorgefertigte *triangulate*-Funktion verwenden, welche eine sequentielle Indexreihenfolge erzeugt. Dadurch ist es konzeptbedingt möglich, dass ein Vertex öfters in einer Indizesliste auftaucht. Solche Vertices werden *shared vertices* (=gemeinsam benutzte Vertices) genannt. Der gehäufte Einsatz von gemeinsam benutzten Vertices führt zu speicherschonendem Laufzeitverhalten, da die Anzahl der Rohdaten sinkt.

Weitere Eigenschaften der Klasse geben noch Aufschluß über die Platzierung von Materialien auf dem Polygon im Ansichtsfenster der Anwendung. Neben der Identifikationsnummer eines Materials wird ein Verschiebefaktor (=Offset) gespeichert, womit ein Material auf dem Polygon verschoben wird. Auch die Anzahl der Wiederholungen eines Materials wird angegeben, wobei eine einzelne Wiederholung den Standardfall darstellt. Durch die Angabe von beispielsweise 3 Wiederholungen würde etwa ein Material durch den Renderer gekachelt in dreifacher Ausführung auf dem Polygon platziert.

Ein Polygon kann einem Gruppenobjekt vom Typ Mesh zugeordnet sein, welches von da an als Vaterobjekt des Polygons gilt. Die Zuordnung läuft über den Namen eines Meshes, der pro Szene eindeutig ist.

Da Polygone von SceneObject erben, bieten sie ausprogrammierte Funktionen für Verschieben, Rotieren, Skalieren und Spiegeln an. Auch die Möglichkeiten zum Verstecken und Einfrieren von Polygonen sind gegeben.

Implementierungstechnisch enthält ein Polygon eine Liste von Vertices (Struktur Vertex im Paket Renderer) und eine Liste von Integern für die Indize. Als Containertyp wurde der STL-Vektor gewählt, weil der wahlfreie Zugriff ebenso häufig im Programm vorkommt wie einfaches durchiterieren.

Zur Kollisionsabfrage steht neben denen von SceneObject übernommenen Methoden für orthogonale und perspektivische Auswahl noch eine weitere Art der Kollisionsabfrage zur Verfügung. Da sich der Anwender entschließen kann, anstatt ganzer Polygone auch nur einzelne Vertices von Polygonen auszuwählen, wurde die Funktion *getIntersectingVertices* hinzugefügt. Diese fügt nur einzelne Vertices dem SelectionBuffer als Auswahl hinzu, wenn diese auch tatsächlich markiert worden sind.

Allgemein ist noch zu beachten, dass bei jeder Änderung der Struktur des Polygons die Bounding Box neu berechnet werden muß um einen konsistenten Zustand zu wahren. Auch beim Rotieren eines Polygons ist dies zu beachten, da achsenausgerichtete Bounding Boxes und keine orientierten Bounding Boxes verwendet werden (siehe Entwurfsentscheidungen).

- **SceneVertex:** Ein SceneVertex dient zur Repräsentation von ungebundenen Vertices, es kapselt so objektorientiert die Vertex-Struktur. Die Klasse erbt von SceneObject und stellt deswegen alle allgemeinen Szenenfunktionen zur Verfügung.

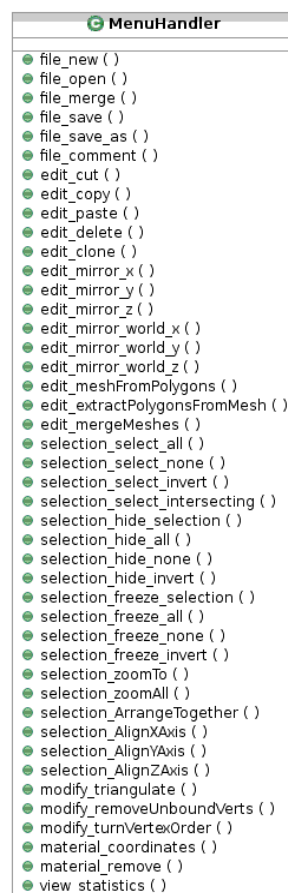
Sobald ungebundene SceneVertexte zu Polygonen zusammengesetzt werden, werden die Objekte gelöscht und die damit verbundenen Vertices daraufhin nur noch von der Klasse Polygon verwaltet.

- **SceneUtils:** Diese Hilfsklasse sammelt allgemeine Funktionen, die von verschiedenen Scenegraph-Subsystemen verwendet werden.

Darunter fallen spezialisierte Ausgabefunktionen für editoreigene Klassen wie AABB oder Vektoren

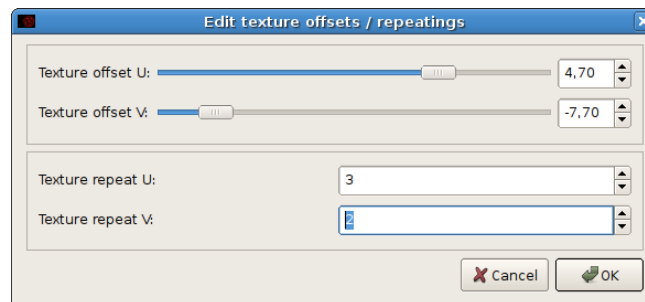
Auch eine Funktion *create2DPlane* zur Erstellung einer mathematisch interpretierbaren Ebene aus 2 Vektoren und einer Achse ist hier abgelegt.

- **MenuHandler:** Diese Klasse stellt die Schnittstelle zwischen Scenegraph und dem Menü des Editors dar. Alle Menüpunkte die mit dem Scenegraphen zusammenhängen werden in dieser Klasse als Slots angeboten. Diese Funktionen werden an den SceneManager weitergeleitet, der die spezifische Funktionalität anbietet. Die Namensgebung folgt dabei der Bezeichnung im Menü: Zuerst der Menüoberpunkt, dann ein Unterstrich gefolgt vom eigentlichen Befehl. Zum Beispiel findet sich die Funktion *selection_zoomAll* im Menü im Oberpunkt "Selection" und dann im aufgeklappten Menü bei "Zoom all".



*Zur Verfügung gestellte
Slots*

- **TextureOffRepDlg:** Diese Klasse stellt die Funktionalität für den Dialog zur Texturverschiebung bereit. Mithilfe diesem Dialogs kann eine Textur auf einem Mesh oder Polygon einerseits verschoben werden und andererseits die Anzahl der Wiederholungen geändert werden.



6.6 Architektursicht – Laufzeit

Die Verwendung des Scenegraph und seiner Schnittstellen wird exemplarisch am Beispiel des Einfügens einiger weniger Vertices durch ein einfaches Plugin demonstriert:

```
// Zugriff auf Instanz des SceneManagers erlangen
SceneManager *sm = SceneManager::getInstance();

Mesh box;
Polygon front;

// Liste der Vertices aufbauen
vector<int> vFront;
vFront.push_back(Vertex(Vector(-1.0, 1.0, 1.0), Vector(0.0, 0.0, 1.0), 0.0, 1.0));
vFront.push_back(Vertex(Vector( 1.0,-1.0, 1.0), Vector(0.0, 0.0, 1.0), 1.0, 0.0));
vFront.push_back(Vertex(Vector( 1.0, 1.0, 1.0), Vector(0.0, 0.0, 1.0), 1.0, 1.0));

// Liste der Indize aufbauen
vector<int> iFront;
iFront.push_back(0);
iFront.push_back(1);
iFront.push_back(2);

// Listen setzen
front.setVertices(vFront);
front.setIndices(iFront);

// Material setzen
front.setMaterialID(nMaterialBox);
// Mesh um Seite erweitern
box.addPolygon(front);

// Mesh in die Szene einfügen
sm->insertObject(box);
```

6.7 Schnittstellen nach außen

Die Schnittstelle für andere Module ist das SceneManager-Singleton, über welche Szenenmanipulationen abgewickelt werden. Außerdem verwaltet er die Instanz der dargestellten Szene und den SelectionBuffer.

6.8 Globale Entwurfsentscheidungen

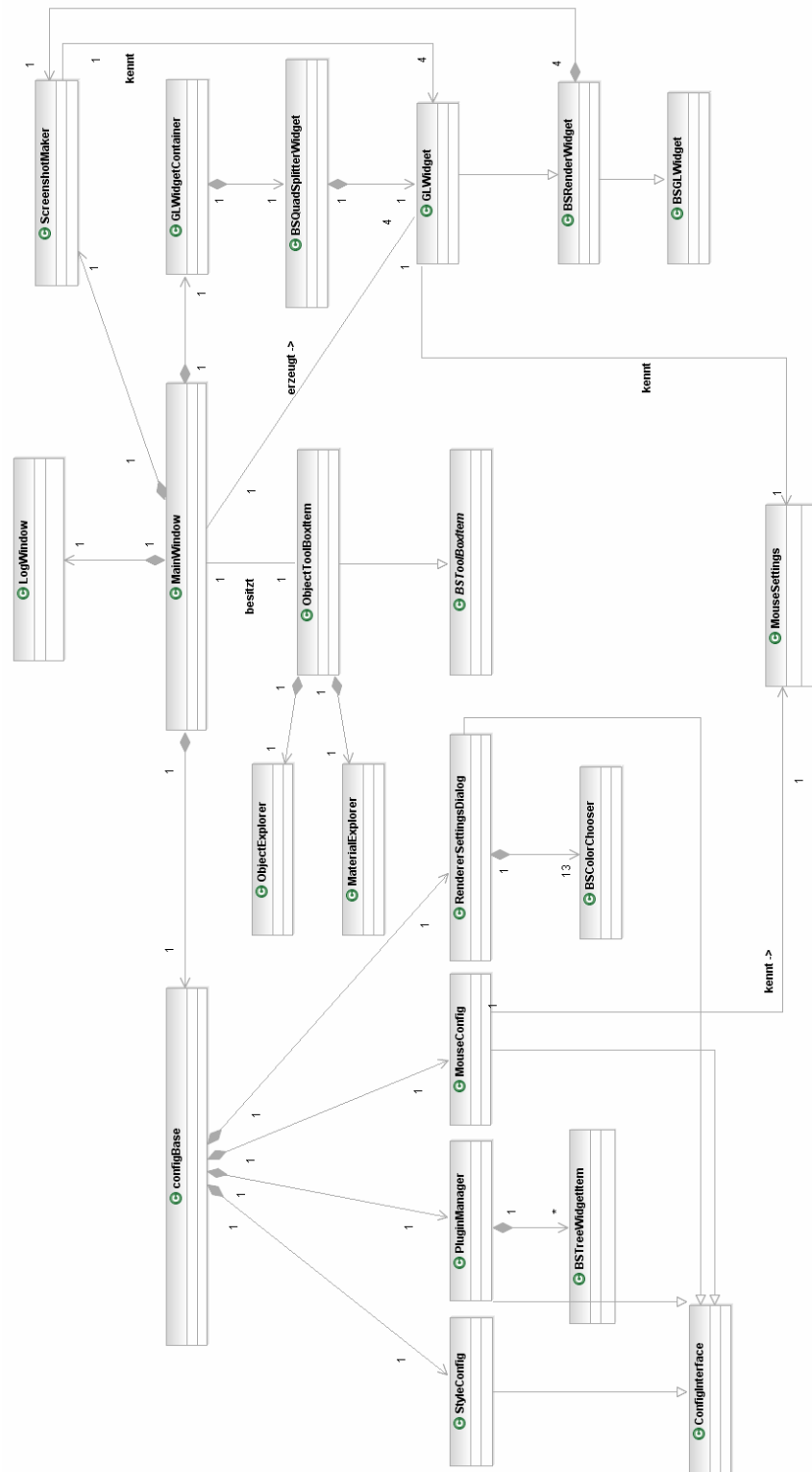
- Zur allgemeinen Haltung von Objektlisten, wie etwa den Vertices eines Polygon, konnten entweder native C-Arrays verwendet werden oder STL-Container. Die Entscheidung fiel relativ schnell und eindeutig auf die Containerklassen der STL. Einerseits sind diese sehr ausgereift und ebenfalls bei auf Performanz ausgelegten Anwendungen einsetzbar und andererseits ist so ein Mitschleifen von Arraygrößen nicht nötig. Da die Arrays dynamisch wachsen können, hätte für jedes C-Array deswegen extra eine Integervariable für die aktuelle Arraygröße gepflegt werden müssen. Dies entfällt bei STL-Containern, weil sie sich selbst um das Vergrößern und dem damit einhergehenden Reservieren von Speicherplatz kümmern.

- Bei den umschließenden Kollisionsboxen für Polygone und Meshes standen die achsenausgerichteten (AABB) und orientierten (OBB) Bounding Boxen zur Auswahl. Die Boxen eines AABB befinden sich immer senkrecht zu den Koordinatenachsen, wohingegen sich Boxen eines OBB frei im Raum rotieren lassen. Die AABB können unter Umständen größer werden, je nach zu umschließendem Objekt, weil sie immer die maximale Ausdehnung in jeder Achse einfangen müssen. Die OBB kann an das Objekt angepasst werden und somit meistens ein Objekt mit weniger Zwischenraum umschließen.
Schließlich wurde zu Gunsten von AABBs entschieden, weil diese deutlich einfach handhabbar sind und keine umfangreichen Winkeloperationen nötig sind. Wegen diesen benötigen OBBs auch deutlich mehr Rechenaufwand, was ebenfalls gegen einen Einsatz dieser spricht. Die dadurch verlorengelende Genauigkeit ist für unsere Zwecke vernachlässigbar.
- Nach Studium spezifischer Literatur wurde ein polygonorientierter Ansatz (siehe Klasse Polygon) gewählt. Als Alternative gäbe es dazu den sogenannten brush-basierten Ansatz, welcher mit Festkörpern arbeitet aus denen dann Formen herausgestanzt werden. Hierbei werden hauptsächlich boolsche Verknüpfungen verwendet, welche aber in abgeänderter Form auch beim gewählten polygonorientierten Verfahren angewand werden kann.

7 Benutzeroberfläche/GUI

Die GUI ist das Modul mit dem der Benutzer letztendlich arbeitet. Die GUI greift über spezifizierte Schnittstellen auf sämtliche Komponenten des Editors zu. Die GUI ist durch Plugins erweiterbar. Diese können z.B. neue Buttons hinzufügen, neue Actions in die Aktionsleiste oder Menüs einfügen oder die Toolbar um neue Widgets erweitern.

7.1 UML



7.2 Funktionale und nicht funktionale Anforderungen

Anforderungen betreffend Komponente Szenengraph:

Verwaltung der entsprechenden Modi (Create Modus, Move Modus, usw.).
Weiterleiten von Klick Events in OpenGL Widgets.

- **Anforderungen betreffend Komponente Renderer:**

Verwalten der entsprechenden Anzeigemöglichkeiten z.B. mit oder ohne Gitter, nur Wireframe, usw.
Verändern der Kameraposition bei Benutzerinteraktion.
Einstellungsmodul um die Einstellungen anzupassen.

- **Anforderungen betreffend Komponente Pluginsystem:**

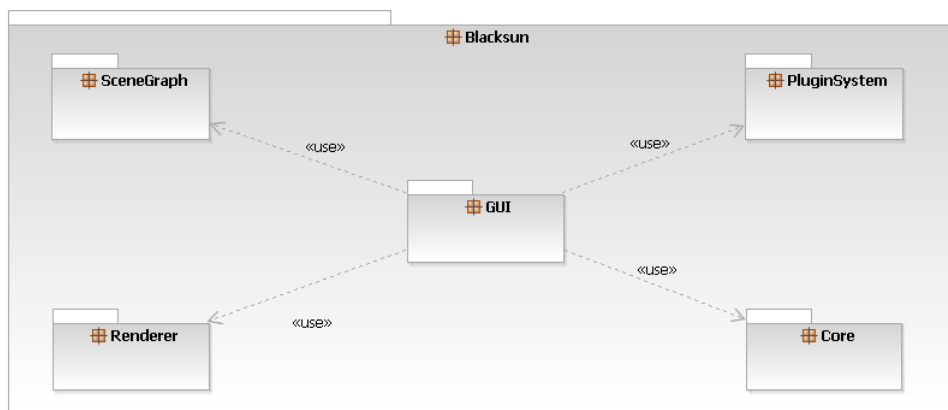
Anbieten verschiedener Funktionen um die GUI an die Bedürfnisse verschiedener Plugins anzupassen.
Pluginmanager zum Verwalten von Plugins (Installieren, löschen, laden, entladen)

- **Anforderungen betreffend Komponente Core:**

Fenster um Logmeldungen grafisch darzustellen.

7.3 Architektursicht – Gesamtsicht

Das folgende Paketdiagramm zeigt die Integration der GUI:



7.4 Architektursicht – Klassensicht

Folgende Klassen sind beteiligt:

- **MainWindow:** Dies ist das Hauptfenster der Applikation. Es verwaltet die GLWidgets, Toolbar, Aktionsleiste und die Statusbar. Dies ist eine Singleton Klasse und existiert programmweit nur ein einziges Mal. Diese Klasse besitzt außerdem Funktionen um die GUI zu manipulieren. Siehe Schnittstellen nach außen.
- **BSToolBoxItem:** Diese Klasse dient als Basisklasse für Widgets in der Toolbox. Sie stellt schon implementierte Funktionen bereit um BSGroupBox-Widgets hinzuzufügen und zu entfernen, sowie Funktionen um Widgets zu einer existierenden BSGorupBox hinzuzufügen.
- **ConfigBase:** Dieses Widget stellt das Konfigurationsfenster dar. Es dient dazu um verschiedene Konfigurationsmodule auf einer einheitlichen Oberfläche darzustellen. Es erstellt selbständig die Standard-Einstellungs-Widgets wie den Pluginmanager oder die Styleconfig.
- **ConfigInterface:** Dieses ist eine abstrakte Basisklasse die implementiert werden muss wenn ein eigenes Konfigurationswidget im Konfigurationsfenster darzustellen ist. Um die Korrekte funktion des Dialogs sicherzustellen, ist es wichtig das das statusChanged(bool) Signal richtig aufgerufen wird.
- **PluginManager:** Dieses Modul implementiert die abstrakte Basisklasse ConfigInterface und dient zur Verwaltung von Plugins. Mit dieser ist es möglich Plugins zu installieren, löschen, laden und zu entladen.
- **StyleConfig:** Dieses Modul implementiert die abstrakte Basisklasse ConfigInterface und dient dazu das Look&Feel der Applikation einzustellen.

- **GLWidget:** Dieses Modul dient dazu um die Szene darzustellen. Dazu enthält es eine Referenz auf den Renderer. Ausserdem bietet das Kontextmenü neben anderen Optionen die Möglichkeit die Rendereinstellungen anzupassen, z.B. ob die Szene als Wireframe gerendert werden soll oder Texturiert.
- **LogWindow:** Dieses Fenster stellt den aktuellen Inhalt des Logs dar.
- **BSColorChooser:** Diese Klasse dient dazu, dass der Benutzer eine Farbe auswählen kann.
- **BSFlowLayout:** Dieses Layout ist ein erweitertes QGridLayout. Sobald ein Item aus diesem Layout entfernt sind werden die verbleibenden Items gegebenenfalls zurückgeschoben so dass keine Lücke im Layout entsteht.
- **BSGLWidget:** Dieses widget ist ein erweitertes QGLWidget. Es dient als basisklasse für die meisten OpenGL-Widget in BlackSun. Es hat die Eigenschaft, dass alle von ihm abgeleitet Widget den gleichen Context teilen. Was nicht heisst das alle den selben rendering context haben. Sie haben lediglich den gleichen Shared Context. Das bedeutet sie teilen sich die GL Display lists etc. Falls sie nicht wissen was ein shared context ist ziehen sie bitte die OpenGL Dokumentation zu rate.
- **BSGroupBox:** Dieses Widget ist eine erweiterung QGroupBox. Es hat die eigenschaft bei einen Klick auf seine Checkbox seinen inhalt zu verbergen.
- **BSQuadSplitterWidget:** Dieses Widget stellt einen sogenannten "QuadSplitter" da. Er verhält sich wie ein QSplitter nur das er seinen Bereich in 4Widgets statt nur 2 aufteilt. Der benutzer ist in der Lage die Größe aller 4 Widgets gleichzeitig zu verändern. Die anderen BSQuadSplitter* Klassen dienen nur der implementierung und sollten nicht direkt verwendet werden.
- **BSRenderWidget:** Mithilfe dieses Widgets ist es möglich die Szene darzustellen. Es besitzt ein Kontextmenü für Einstellungen am Renderer und es folgt den Einstellungen die der Benutzer in der MouseSettings festgelegt wurden.
- **GLWidgetContainer:** Dieses Widget verwaltet die GLWidgets. Es sendet auch die entsprechenden signale aus um anzeigen zu können ob z.b. Alle Widgets das Grid anzeigen. Auch lassen sich hier auf die optionen alle GLWidgets setzen. Bei einer veränderung der optionen im GLWidget überprüft der GLWidgetContainer ob alle anderen widgets die gleiche option gesetzt haben und stößt dann das entsprechende signal aus.
- **MaterialExplorer:** Dieses Widget dient dazu Materialien im Hauptfenster zu verwalten.
- **MouseConfig:** Diese Klasse ist ein Konfigurationsmodul mit deren hilfe der Benutzer die mauseinstellungen verändern kann.
- **MouseSettings:** Diese Singleton Klasse dient als Puffer zwischen QSettings und den Mauseinstellungen. Sie list bei erstellung die Einstellungen aus den QSettings und sie erlaubt sie stellt die einstellungen als propertys zur verfügung. Dies dient dazu um nicht bei jeden mouse move event in einem BSRenderWidget auf die QSettings und damit auf die Festplatte zugreifen zu müssen.
- **ObjectExplorer:** Dieses Widget dient dazu die Meshes der Szene zu verwalten. De benutzer ist in der lage diese umzubennen oder zu löschen.
- **PluginManager:** Mithilfe dieses Konfigurationsmodules ist der Benutzer in der lage Plugins zu laden und zu entladen. Auch kann der benutzer den Pfad festlegen in dem nach Plugins gesucht werden soll.
- **RendererSettingsDialog:** In diesem Konfigurationsmodul kann der Benutzer die Einstellugen für den Renderer vornehmen.
- **ScreenshotMaker:** Mit diesem Widget ist es möglich die Aktuelle Szene in ein Bild zu rendern.

7.5 Architektursicht – Laufzeit

Dieses Beispiel zeigt wie man einen QPushButton in das erste Toolbox Widget einfügen kann:

```
QPushButton createBoxButton = new QPushButton("Box");
BSGui::MainWindow::getInstance()->getAvailableToolBoxItems().at(0)->addWidgetToGroupBox(
    "modeButtonsGroupBox", createBoxButton);
```

7.6 Schnittstellen nach außen

Die GUI soll flexibel genug sein um von Plugins angepasst werden zu können. Neue Widgets können zur Toolbar einfach über die Klasse MainWindow mit der entsprechenden Funktionen hinzugefügt werden. Die Funktion `getAvailableToolBoxItems()` liefert alle Widgets, welche zur Zeit in der Toolbox geladen sind.

Die `BSToolBoxItem` Klasse bietet einige einfache Operation um Widgets hinzuzufügen. Diese bauen jedoch auf den `objectName` Attribut der Klasse `QObject` auf. Deshalb sollte man möglichst eindeutige Objektnamen vergeben. Sollten diese Funktionen nicht ausreichen, das entsprechende Widget zu manipulieren, muss auf die entsprechenden Funktionen der Klasse `QWidget` zurückgegriffen werden um das Widget seinen Wünschen anzupassen.

7.7 Einflussfaktoren und Randbedingungen

Die Entscheidung für das QT Toolkit kam einerseits daher, dass es Plattform unabhängig ist und das es wie unsere Applikation auch in C++ geschrieben wurde.

7.8 Globale Entwurfsentscheidungen

Da es nicht möglich ist zyklische Abhängigkeiten in dynamischen Bibliotheken zu haben geht das direkte aufrufen von Funktionen nur in eine Richtung. Da der häufigere Fall der ist, dass von der GUI ein Funktionsaufruf einer Funktion der anderen Module erfolgt, ist das Aufrufen von Funktionen nur in dieser Richtung möglich. Da es jedoch vorkommen kann, dass andere Module der GUI etwas mitteilen müssen (z.B. eine Änderung in der Logdatei oder auch Änderungen an der internen Datenstruktur die nicht von der GUI versucht wurden) gibt es die Möglichkeit durch QT's Signal- und Slot-Mechanismus Signale zu schicken die von der GUI aufgefangen und entsprechend verarbeitet werden können.

8 Pluginspezifikation

8.1 Aufbau der dynamischen Bibliothek

- Die Bibliothek sollte folgende Dateien enthalten:
 - evtl. ein Interface des Plugins (empfohlen)
 - eine Klasse für das Plugin (Header- und Source – Datei)
 - evtl. andere Klassen / Dateien
- Inhalt der Dateien:
 - Das minimale Interface des Plugins:

Die Datei MeinPluginInterface.hh:

```
#include <PluginInterface.hh>

class MeinPlugin : public PlgInt
{
public:
    static const UID uid = 3835135349UL;

    MeinPlugin() {}
    ~MeinPlugin() {}

    // Funktionen, die dieses Plugin anderen zur Verfügung stellt
    // in der Form:
    // <Rückgabewert> <Funk.Name>(<Funktionsparameter>) = 0;
    // . . .
};
```

Man kann auch die Plugin – Klasse auch direkt von PlgInt ableiten, allerdings hat die Verwendung des Interfaces den Vorteil, das eine besser zu kontrollierende Schnittstelle zu anderen Plugins vorhanden ist.

- Der minimale Inhalt der eigentlichen Plugin-Klasse:

Die Datei MeinPlugin.h:

```
#include "MeinPluginInterface.hh"

class MeinPlugin : public MeinPluginInterface
{
public:
    MeinPlugin();
    virtual ~MeinPlugin();

    // Implementierte Funktionen von PlgInt:
    UID getUID();
    QString getName();
    QString getAutor();
    QString getDescription();
    Version getVersion();
    bool installPlg();
    bool loadPlg();
    bool unloadPlg();
    bool uninstallPlg();
    void closePlg();

    // Implementiert Funktionen von MeinPluginInterface
    // (s.o.):
    // <Rückgabewert> <Funk.Name>(<Funktionsparameter>);
};
```

Die Datei MeinPlugin.cpp:

```
#include "MeinPlugin.h"
// Interfaces von anderen Plugins, die dieses Plugin benutzt
// (siehe 8.1.3)

MeinPlugin::MeinPlugin()
{
    // Konstruktor des Plugins, wird nur einmal in createInstance
    // aufgerufen.
}

MeinPlugin::~MeinPlugin()
{
    // Destruktor des Plugins
}

UID MeinPlugin::getUID()
{
    // Liefert die eindeutige UID des Plugins. (siehe 8.1.2)
}

QString MeinPlugin::getName()
{
    // Gibt den Namen des Plugins zurück.
}

QString MeinPlugin::getAutor()
{
    // Liefert den Autor des Plugins. Es sind auch Hyperlinks
    // möglich [??]
}

QString MeinPlugin::getDescription()
{
    // Liefert die Beschreibung des Plugins. Es sind HTML - Tags
    // möglich [??]
}

Version MeinPlugin::getVersion()
{
    // Gibt die Version zurück. (siehe 8.1.2)
}

// Die nachfolgenden Funktion wird vom Pluginmanager aufgerufen
// wenn das Plugin: (Hinweis: falls sie 'false' zurückgegeben,
// dann wird der Vorgang abgebrochen, das heißt, das Plugin
// wird nicht installiert/geladen/... . Mögliche Gründe hierfür
// sind fehlende andere Plugins. Das Plugin muss sich selbst
// darum kümmern, das der Benutzer das Problem erkennt (mit z.B.
// Ausgabe über den Logger oder mit Messageboxen))
bool MeinPlugin::installPlg()
{
    // installiert wird oder wenn es schon installiert war und
    // BlackSun gestartet wird.
}

bool MeinPlugin::loadPlg()
{
    // geladen wird.
}

bool MeinPlugin::unloadPlg()
{
    // entladen wird. (Hinweis: geladene Plugins werden am
    // Programmende nicht entladen)
}

bool MeinPlugin::uninstallPlg()
{
    // deinstalliert wird. (Hinweis: installierte Plugins werden
    // am Programmende nicht deinstalliert)
}
```

```

void MeinPlugin::closePlg()
{
    // Diese Funktion wird aufgerufen, wenn das Programm
    // geschlossen wird.
}

// Funktionen, die dieses Plugin anderen zur Verfügung stellt:
// . . .
// . . .
// . . .

// Über den nachfolgenden Code kann der Pluginmanager die Plugins
// laden.
static MeinPlugin* _instance = 0; // oder = NULL

extern "C" PlgInt* createInstance()
{
    if( _instance == 0)
        _instance = new MeinPlugin();

    return static_cast<PlgInt*> (_instance);
}

```

Man kann auch die Plugin – Klasse auch direkt von PlgInt ableiten, allerdings hat die Verwendung des Interfaces den Vorteil, das eine besser zu kontrollierende Schnittstelle vorhanden ist.

- Erklärungen:
 - UID (Unique ID): Also eine eindeutige ID. Der eigentliche Datentyp einer UID ist ein unsigned long (32 Bit). Im Idealfall unterscheiden sich die UIDs aller Plugins. Am einfachsten ist dies mit einem (Hash-) Algorithmus zu erreichen (z.B. CRC32) und diesem als Eingabe den Inhalt der Interface - Datei geben.
 - Das Prinzip der Abwärtskompatibilität: Ruft ein Plugin einen Zeiger auf ein anderes Plugin (siehe 8.2.3) ab, so geschieht dies über die UID und die Version des anderen Plugins. Die Vereinbarung sagt jetzt, das alle Plugins mit der gleichen UID abwärts kompatibel zu früheren Versionen sein müssen. Ist dies nicht gewährt, muss die UID des neuen Plugins geändert werden. So liefert der Pluginmanager eine Fehlermeldung, wenn ein Plugin gefordert wird, die geforderte UID auch vorhanden ist, die benötigte Version aber über der der Installierten liegt. Umgekehrt wird aber ein Plugin zurückgegeben, wenn die UID vorhanden ist und die geforderte Version kleiner oder gleich der installierten ist.
- Aufruf einer Funktion eines anderen Plugins:

```

// Includieren des Interfaces des anderen Plugins in der MeinPlugin.cpp:
#include <AnderesPlugin.hh>

// Aufrufen der benötigten Funktion:
AnderesPlugin* ap = PlgMgr::getInstance()
->getPlgInstance(AnderesPlugin::uid,Version(1,0,0));

if(ap == NULL)
    // Fehler!!

// Aufruf der Funktion des anderen Plugins:
ap->andereFunktion()

```

Hinweis: Für ein ausführliches Beispiel siehe Kapitel „11. Pluginerstellung“ des Handbuches! Hier wird unter anderem darauf eingegangen, wie ein Plugin mit dem Editor kommuniziert oder wie es eine Schaltfläche in der GUI registriert.

9 Plugins

Der größte Teil der Funktionalität des Editors wird über zusätzliche Plugins hinzugefügt. Es gibt drei Arten von Plugin:

- **Grundkörper-Plugins:** Hier werden Körper anhand bestimmter Parameter generiert und in die Szene eingefügt
- **Modifikator-Plugins:** Selektierte Modell- und Szenendaten werden hier manipuliert
- **Sonstige Plugins:** Alle sonstigen Tools, die zur Modellierung hilfreich sind

In den folgenden Kapiteln werden diese Plugins näher beschrieben:

Grundkörper-Plugins:

- Block
- Box
- GeoSphere
- Grid (Gitter)
- Helix
- nBox (Zylinder)
- Pyramid
- Sphere (Kugel)
- Tube (Röhre)
- Torus
- Goblet (Kelch)

Modifikator-Plugins:

- Convex Hull (Konvexe Hülle)
- Explosion
- Extrude
- Insert vertex (Vertex einfügen)
- Make Hole
- Revolving (Rotationskörper)
- Smooth Corners (Weiche Ecken)
- Spike (Stacheln)
- Subdivision
- Split face (Face aufteilen)

Sonstige Plugins:

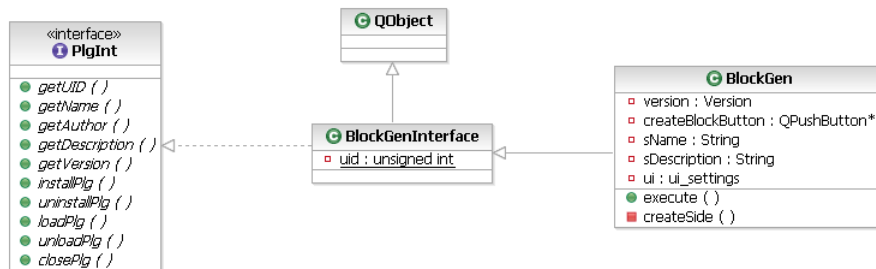
- Manual edit (Manuelle Bearbeitung)
- Materialeditor
- Texturcoordinate-Editor
- Im- und Exporter-Pack
- TextureLoaderPack
- Updater

10 Grundkörper-Plugins

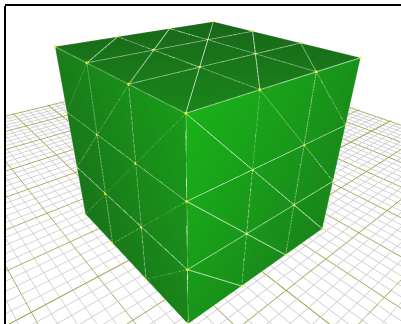
10.1 Block

Dieses Plugin erstellt einen Würfel, der in beliebig viele Teile untergliedert ist.

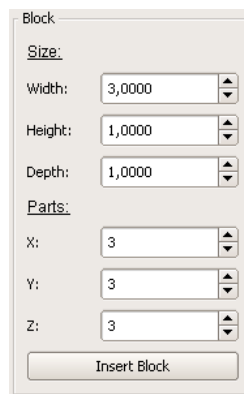
10.1.1 UML



10.1.2 Screenshots



Bespiel einer Box (3x3x3)



Einstellungs-Widget

10.1.3 Funktionale und nicht funktionale Anforderungen

- Die Ausmaße des Würfels werden über die *Länge*, *Breite* und *Höhe* beschrieben.
- Die Untergliederungen müssen für alle Achsen/Ebenen unabhängigkeit voneinander einstellbar sein.
- Der generierte Block muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.
- Bei ungültigen Parametern (z.B. Breite ≤ 0) soll kein Block erstellt werden.

10.1.4 Mengengerüste


Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

$$\begin{aligned} \text{Indices (Integer, je 4 Byte):} & \quad 2 * [(6 * X * Y) + (6 * Z * Y) + (6 * X * Z)] \\ \text{Vertices (je 112 Byte):} & \quad 2 * [(X + 1) * (Y + 1) + (Z + 1) * (X + 1) + (X + 1) * (Z + 1)] \end{aligned}$$

10.1.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **BlockGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **BlockGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.1.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung des Blocks selber läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für alle 6 Seiten des Blocks
    Wandere durch Höhensegmente der Seite
        Wandere durch alle Breitensegmente der Seite
            Generiere den aktuellen Vertex aus aktuellem Höhen- und Breitensegment
            Falls es nicht die letzte Reihe ist...
                Erstelle 6 Indices (=2 Faces) für dieses Segment und verbinde damit
                aktuelle mit nächster Reihe
```

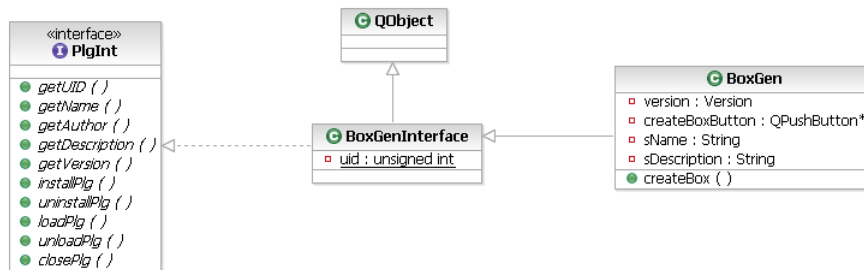
10.1.7 Globale Entscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Breite = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Block generiert.

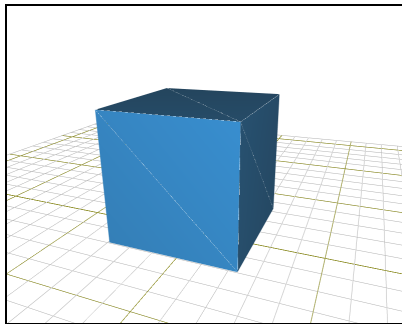
10.2 Box

Dieses Plugin erstellt eine Box mit Höhe, Breite und Tiefe von 1.

10.2.1 UML



10.2.2 Screenshots



Beispiel einer Box (3x3x3)

10.2.3 Funktionale und nicht funktionale Anforderungen

- Die Ausmaße des Würfels sollen *Länge*, *Breite* und *Höhe* von 1 sein.
- Der generierte Block muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.


10.2.4 Mengengerüste

Das generierte Mesh ist hardcodiert und wird ohne Berechnungen in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte):	36
Vertices (je 112 Byte):	24

10.2.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **BoxGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **BoxGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient.

10.2.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Boxicon angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Durch die Hardcodierung sind keine Berechnungen nötig.

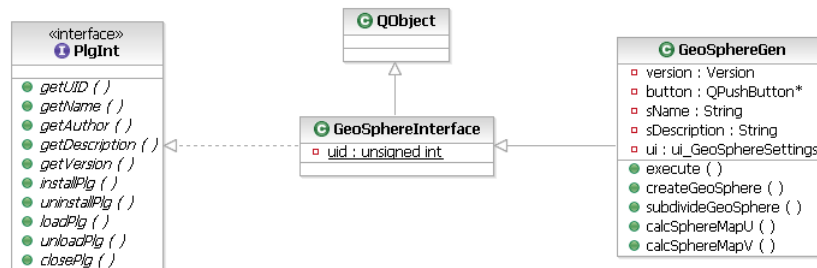
10.2.7 Globale Entwurfsentscheidungen

- Das Plugin wurde hardcodiert um eine Einheitsbox zu erhalten. Desweiteren diente dieses (erste) Plugin zu Testzwecken, weshalb Rechenfehler ausgeschlossen werden sollten.

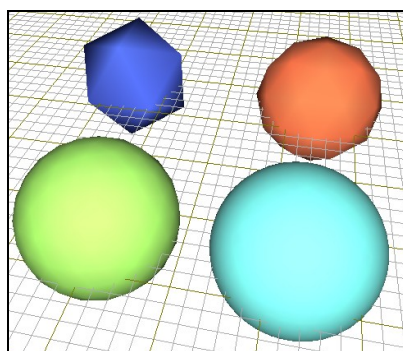
10.3 GeoSphere

Dieses Plugin erstellt eine komplette GeoSphere. Eine GeoSphere ist eine Sphere, die durch Verfeinerung eines Grundkörpers (Icosaeder = 20-Flächen) entsteht. Jede Verfeinerungsstufe teilt das Face in 4 Subfaces und verschiebt die alten Eckpunkte. Aufgrund dieser Art der Verfeinerung wirkt die GeoSphere plastischer, natürlicher und regelmäßiger als die 'normale' Sphere, da sie keine 'Pole' aufweist (vgl. dazu Screenshots der Sphere).

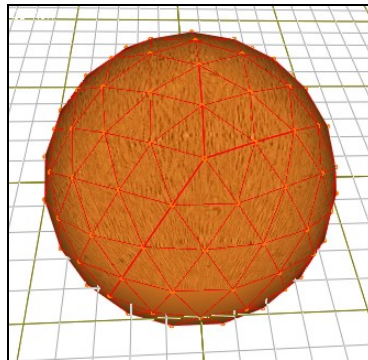
10.3.1 UML



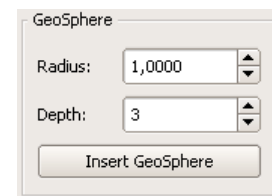
10.3.2 Screenshots



GeoSpheren unterschiedlicher Tiefe



Einteilung einer GeoSphere



Einstellungs-Widget

10.3.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Die GeoSphere soll über die Parameter 'Radius' und 'Depth' beschrieben werden
- Der Detailgrad muss einstellbar sein über die Unterteilungstiefe (Depth)
- Die generierte Kugel muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Bei ungültigen Parametern (z.B. $Radius \leq 0$) soll keine GeoSphere erstellt werden


10.3.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte): $6 * 20 * (4^{\wedge} Depth)$
 Vertices (je 112 Byte): $20 * (4^{\wedge} Depth)$

10.3.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **GeoSphereGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **GeoSphereGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.3.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der GeoSphere selber läuft dabei wie folgt ab (Pseudocode):

```

Prüfen ob Parameter gültig sind
Erstelle den Icosaeder (20-Flächen) als Basisobjekt
Wandere durch alle Faces des Basisobjekts (0 bis 19)
    Verfeinere das Dreieck über Methode 'subdivideGeoSphere'

subdivideGeoSphere:
Falls Depth=0
    Füge das Face unverändert hinzu
Sonst
    Berechne 3 neue Positionen zwischen jede Kante des Faces(um es in 4 Subfaces zu teilen)
    Verschiebe die neuen Positionen, sodass diese auf der Kugeloberfläche liegen
    Führe für jedes der 4 Subfaces 'subdivideGeoSphere' mit Depth-1 aus
  
```

10.3.7 Einflussfaktoren und Randbedingungen

- Als Basiskörper (*Depth* = 0) wurde der Icosaeder gewählt, da das der gebräuchlichste Körper ist, der für die Verfeinerung hergenommen wird.

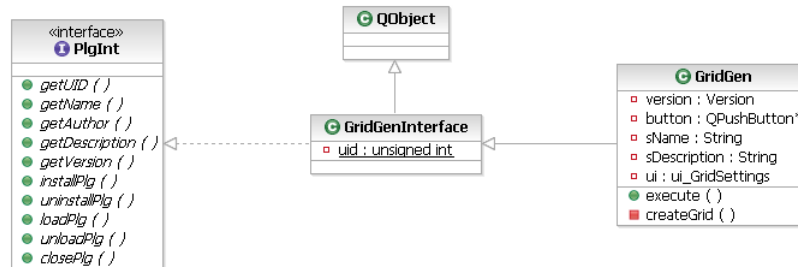
10.3.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. *Depth* < 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine GeoSphere generiert.

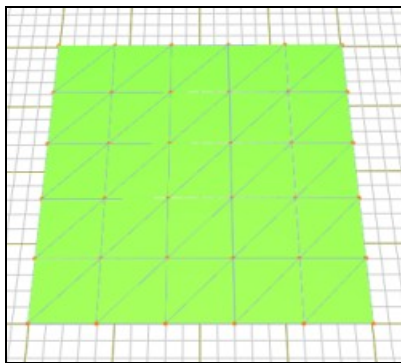
10.4 Grid (Gitter)

Dieses Plugin erstellt ein Grid. Ein Grid ist ein flaches Objekt, dass eine beliebig unterteilte Ebene darstellt.

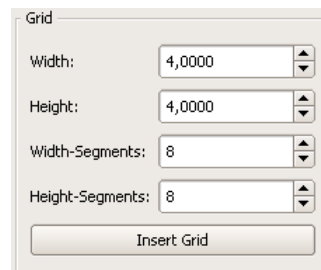
10.4.1 UML



10.4.2 Screenshots



Beispiel eines Grids



Einstellungs-Widget

10.4.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Das Gitter soll über die Parameter 'Width', 'Height' und der Segmenteinteilung beschrieben werden
- Der Detailgrad muss einstellbar sein über die Unterteilungsanzahl (Width- und Height-Segments)
- Das generierte Gitter muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Das Mesh soll über shared Vertices verfügen, d.h. die Dreiecke des Gitters sollen sich so viele Vertices wie möglich teilen, sodass die Vertexanzahl gering bleibt
- Bei ungültigen Parametern (z.B. Width = 0) soll kein Gitter erstellt werden

10.4.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:


Indices (Integer, je 4 Byte): $6 * WidthSegments * HeightSegments$

Vertices (je 112 Byte): $(WidthSegments + 1) * (HeightSegments + 1)$

10.4.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **GridGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **GridGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.4.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung des Gitters selber läuft dabei wie folgt ab (Pseudocode):

```

Prüfen ob Parameter gültig sind
Wandere durch alle zu erstellenden Höhen-Segmente (0 bis HeightSegments-1)
  Wandere durch alle zu erstellenden Breiten-Segmente (0 bis WidthSegments-1)
    Generiere den aktuellen Vertex aus aktuellem Ring und Segment
    Falls es nicht des Höhen- oder Breitensegment ist, ...
      Erstelle 6 Indices (= 2 Faces) für diesen Gitterteil und verbinde damit aktuellen
      mit darüberliegendem Höhensegment

```

10.4.7 Einflussfaktoren und Randbedingungen

- Um die Anzahl der Vertices gering zu halten (und damit Speicherplatz zu sparen), wird das Gitter mit 'Shared Vertices' berechnet. Benachbarte Faces teilen sich so gemeinsam einen oder mehrere Vertices. Dieses Teilen von Vertices wird über die Indices erreicht. So haben zwei Faces, die sich einen Vertex teilen, beide den gleichen Index. Insgesamt werden so ca. 80% der Vertices eingespart, die sonst zu oft vorhanden wären.

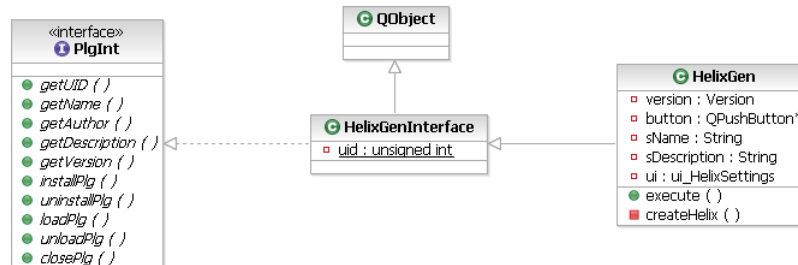
10.4.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. *Width* = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Gitter generiert.

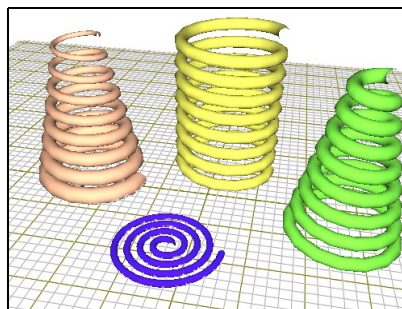
10.5 Helix

Dieses Plugin erstellt eine 3-dimensionale Helix (Spirale). Je nach Einstellung können z.B. ebene, zylinderumwickelnde oder kegel-umwickelnde Spiralen generiert werden. Ebenso lassen sich Körper erstellen werden, die zum Ende hin immer dicker/schlanker werden. Die Enden der Helix sind immer offen.

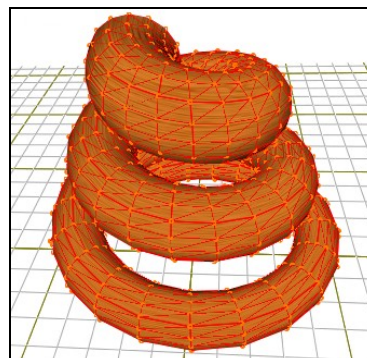
10.5.1 UML



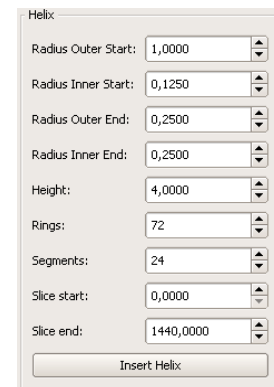
10.5.2 Screenshots



Beispiel von Spiralen



Erkennbare Unterteilungen



Einstellungs-Widget

10.5.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Die Helix soll über die Parameter 'Radius Outer Start/End', 'Radius Inner Start/End', 'Rings', 'Segments' und den Start- und Endwinkel ('Slice start/from') beschrieben werden.
- Über die Unterteilung des inneren und äußeren Radius in Start- und Endradius soll eine optionale Verjüngung zu einem der Enden erreicht werden
- Der Detailgrad muss einstellbar sein über eine radiale Einteilung (Segments) und eine Höhenunterteilung (Rings)
- Die generierte Helix muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Das Mesh soll über shared Vertices verfügen, d.h. die Dreiecke der Helix sollen sich so viele Vertices wie möglich teilen, sodass die Vertexanzahl gering bleibt
- Bei ungültigen Parametern (z.B. Radius ≤ 0) soll keine Helix erstellt werden
- Sollte der Startwinkel $>$ Endwinkel sein, so soll ein nach innen gestülpter Helix generiert werden (die Facevorderseite zeigt nach innen)


10.5.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte): $6 * Rings * (Segments + 1)$
 Vertices (je 112 Byte): $(Rings + 1) * (Segments + 1)$

10.5.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **HelixGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **HelixGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.5.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der Helix selber läuft dabei wie folgt ab (Pseudocode):

```

Prüfen ob Parameter gültig sind
Wandere durch alle zu erstellenden Ringe (0 bis Rings-1)
    Wandere durch alle zu erstellenden Ringsegmente (0 bis Segments-1)
        Generiere den aktuellen Vertex aus aktuellem Ring und Segment
        Falls es nicht der letzte Ring ist, ...
            Erstelle 6 Indices (= 2 Faces) für dieses Segment und verbinde damit aktuellen mit
            nächsten Ring

```

10.5.7 Einflussfaktoren und Randbedingungen

- Um die Anzahl der Vertices gering zu halten (und damit Speicherplatz zu sparen), wird die Helix mit 'Shared Vertices' berechnet. Benachbarte Faces teilen sich so gemeinsam einen oder mehrere Vertices. Dieses Teilen von Vertices wird über die Indices erreicht. So haben zwei Faces, die sich einen Vertex teilen, beide den gleichen Index. Insgesamt werden so ca. 80% der Vertices eingespart, die sonst zu oft vorhanden wären.
- Eine nicht geschlossene Helix (falls er keine 360° einschließt und Höhe > 0.0) hat keine geschlossenen Enden. Diese können, falls erwünscht, mit anderen Funktionen des Editors erzeugt werden (Triangulation, ...).

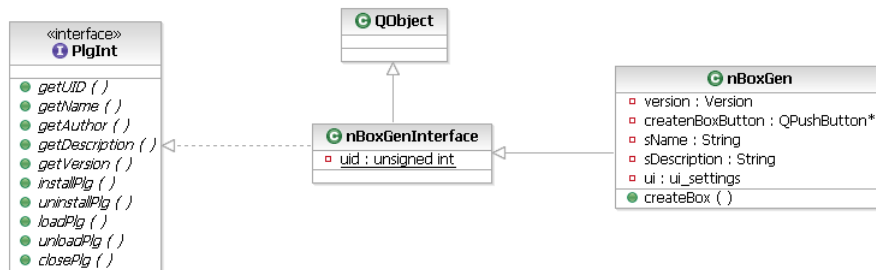
10.5.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. $Radius\ Outer\ Start = Radius\ Outer\ End = 0$), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine Helix generiert.
- Die Angabe 'Rings' zur radialen Unterteilungen gilt für der gesamten Helix. So hat ein Helix mit 180° genauso viele Unterteilungen wie beispielsweise eine 720°-Helix.

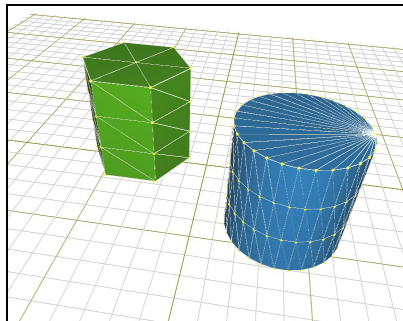
10.6 nBox (Zylinder)

Dieses Plugin erstellt eine Box mit beliebig vielen Ecken. Generiert wird immer eine komplette Box.

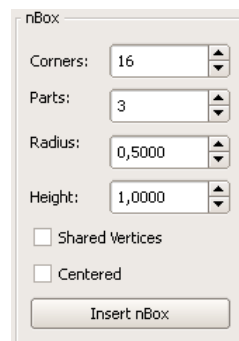
10.6.1 UML



10.6.2 Screenshots



Beispiel von Zylindern



Einstellungs-Widget

10.6.3 Funktionale und nicht funktionale Anforderungen

- Die Ausmaße der Box werden über die *Höhe*, die *Anzahl der Ecken* und den *Radius* beschrieben.
- Es soll wählbar sein, ob die Vertices geteilt werden und ob der Deckel/Boden vom Mittelpunkt oder einem Eckpunkt aus berechnet werden.
- Die Angabe der Segmente der Höhe soll möglich sein.
- Die generierte Box muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.
- Bei ungültigen Parametern (z.B. Radius ≤ 0) soll keine Box erstellt werden.

10.6.4 Mengengerüste


Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte): $i_{\text{Mantel}} + 2 \cdot i_{\text{Deckel}}$
 Vertices (je 112 Byte): $v_{\text{Mantel}} + 2 \cdot v_{\text{Deckel}}$

i_{Deckel} : $3 \cdot \text{Ecken} - 2$
 $i_{\text{Deckel (centered)}}$: $3 \cdot (\text{Ecken} - 1) + 3$
 v_{Deckel} : Ecken
 $v_{\text{Deckel (centered)}}$: $\text{Ecken} + 1$
 i_{Mantel} : $6 \cdot \text{Ecken} \cdot \text{Segmente}$
 v_{Mantel} : $(1 + \text{Segmente}) \cdot \text{Ecken} \cdot 2$
 $v_{\text{Mantel (sharedVertices)}}$: $(1 + \text{Segmente}) \cdot \text{Ecken}$

10.6.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **nBoxGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **nBoxGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.6.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der nBox selber läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für Deckel und Boden
    Für jede Ecke
        Speichere aktuellen Vertex des Radiusvektors
        Rotiere Radiusvektor
        Erstelle in Abhängigkeit von Centered 3*Ecken bzw. 3*(Ecken-2) Indices
Wandere durch die zu erstellenden Seiten
    Kopiere entsprechende Vertices aus Deckel/Boden und generiere restliche
    aus Segmente und Höhe
    Erstelle 6 Indices für jedes Segment der Seite. Bei sharedVertices verbinde
    damit aktuelle mit nächster Seite
```

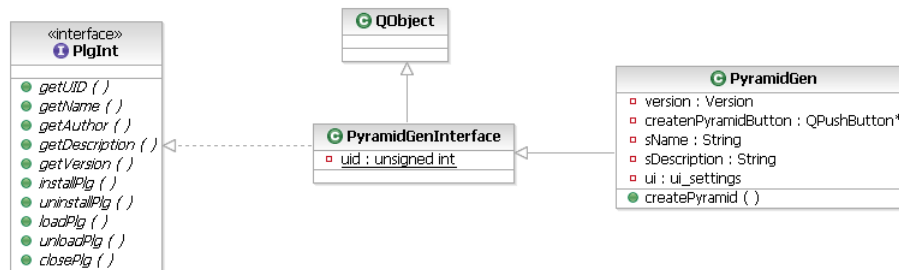
10.6.7 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Ecken = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine nBox generiert.
- Dieses Plugin ersetzt ein Zylinderplugin, da dieser durch sharedVertices und ausreichend Ecken entsteht.

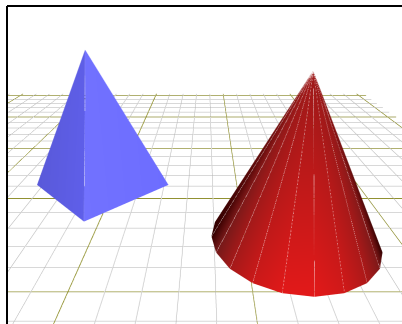
10.7 Pyramid

Dieses Plugin erstellt eine Pyramide.

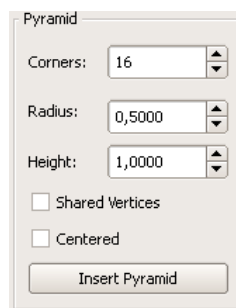
10.7.1 UML



10.7.2 Screenshots



Beispiel von Pyramiden



Einstellungs-Widget

10.7.3 Funktionale und nicht funktionale Anforderungen

- Die Ausmaße der Pyramide werden über die *Höhe*, *Anzahl der Ecken* und den *Radius* beschrieben.
- Es soll wählbar sein ob Vertices geteilt werden und ob der Boden vom Mittelpunkt oder einer Ecke aus berechnet wird.
- Die generierte Pyramide muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.
- Bei ungültigen Parametern (z.B. Radius ≤ 0) soll keine Pyramide erstellt werden.

10.7.4 Mengengerüste


Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte):	$3 * \text{Ecken} + 3 * (\text{Ecken} - 2)$
Vertices - centred (je 112 Byte):	$6 * \text{Ecken}$
Indices (Integer, je 4 Byte):	$3 * \text{Ecken} + \text{Ecken} + 1$
Vertices – shared Vertices (je 112 Byte):	$2 * \text{Ecken} + \text{Ecken} + 1$

10.7.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **PyramidGenInterface**: Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **PyramidGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.7.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der Pyramide selber läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für jede Ecke
    Speichere aktuellen Vertex des Radiusvektors
    Rotiere Radiusvektor
    Erstelle in Abhängigkeit von Centered 3*Ecken bzw. 3*(Ecken-2) Indices
Wandere durch die zu erstellenden Seiten
    Kopiere entsprechenden Vertex aus Boden und in Abhängigkeit von shared Vertices füge
    Vertex der Spitze hinzu.
    Erstelle 3 Indices für jede Seite. Bei shared Vertices verbinde damit die aktuelle und die
    nächste Seite.
```

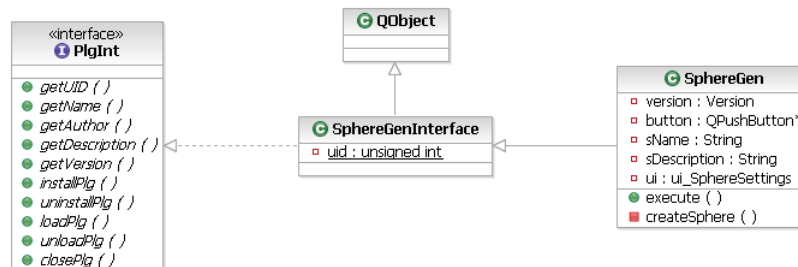
10.7.7 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Radius = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine Pyramide generiert.
- Dieses Plugin ersetzt ein Kegelpugin, da dieser durch sharedVertices und ausreichend Ecken entsteht.

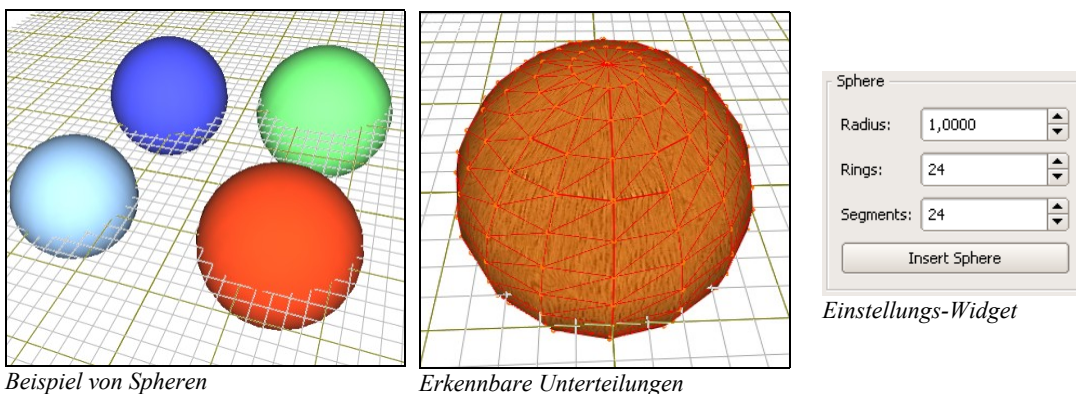
10.8 Sphere (Kugel)

Dieses Plugin erstellt eine Kugel mit beliebigem Detailgrad. Generiert werden immer geschlossene Kugeln (also z.B. keine Kugel-Segmente, Halbkugeln, ...).

10.8.1 UML



10.8.2 Screenshots



10.8.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Die Kugel soll über die Parameter 'Radius', 'Rings' und 'Segments' beschrieben werden
- Der Detailgrad muss einstellbar sein über eine radiale Einteilung (*Segments*) und eine Höhenunterteilung (*Rings*)
- Die generierte Kugel muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Das Mesh soll über shared Vertices verfügen, d.h. die Dreiecke der Kugeln sollen sich so viele Vertices wie möglich teilen, sodass die Vertexanzahl gering bleibt
- Bei ungültigen Parametern (z.B. $Radius \leq 0$) soll keine Kugel erstellt werden


10.8.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

$$\begin{aligned}
 \text{Indices (Integer, je 4 Byte):} & \quad 6 * \text{Rings} * (\text{Segments} + 1) \\
 \text{Vertices (je 112 Byte):} & \quad (\text{Rings} + 1) * (\text{Segments} + 1)
 \end{aligned}$$

10.8.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- ***SphereGenInterface***: Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- ***SphereGen***: Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.8.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der Sphere selber läuft dabei wie folgt ab (Pseudocode):

```

Prüfen ob Parameter gültig sind
Wandere durch alle zu erstellenden Ringe (0 bis Rings-1)
    Wandere durch alle zu erstellenden Ringsegmente (0 bis Segments-1)
        Generiere den aktuellen Vertex aus aktuellem Ring und Segment
        Falls es nicht der letzte Ring ist, ...
            Erstelle 6 Indices (= 2 Faces) für dieses Segment und verbinde damit aktuellen mit
            nächsten Ring

```

10.8.7 Einflussfaktoren und Randbedingungen

- Um die Anzahl der Vertices gering zu halten (und damit Speicherplatz zu sparen), wird die Sphere mit 'Shared Vertices' berechnet. Benachbarte Faces teilen sich so gemeinsam einen oder mehrere Vertices. Dieses Teilen von Vertices wird über die Indices erreicht. So haben zwei Faces, die sich einen Vertex teilen, beide den gleichen Index. Insgesamt werden so ca. 80% der Vertices eingespart, die sonst zu oft vorhanden wären.

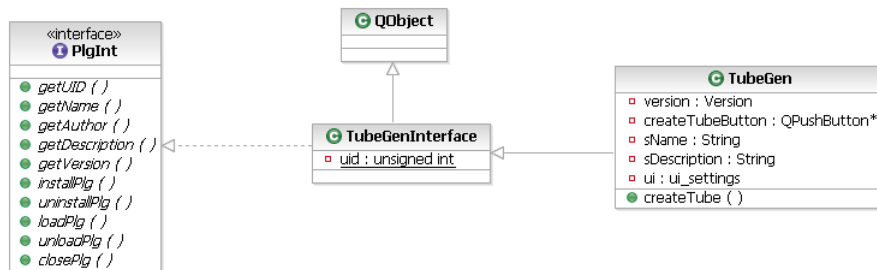
10.8.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. *Radius* = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine Sphere generiert.

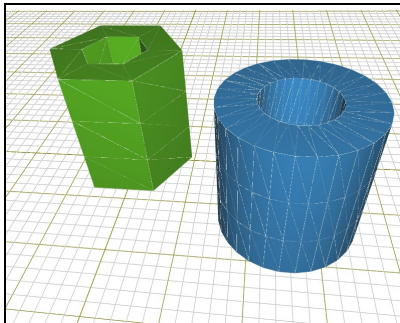
10.9 Tube (Röhre)

Dieses Plugin erstellt eine Röhre mit beliebig vielen Ecken.

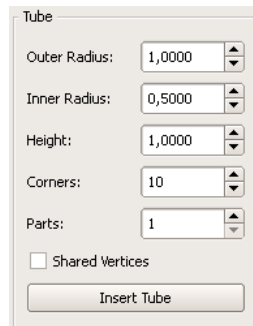
10.9.1 UML



10.9.2 Screenshots



Beispiel von Röhren



Einstellungs-Widget

10.9.3 Funktionale und nicht funktionale Anforderungen

- Die Ausmaße der Röhre werden über die *Höhe*, die *Anzahl der Ecken* und den *äußeren* und *inneren Radius* beschrieben.
- Es sollen der äußere und innere Radius unabhängig voneinander eingestellt werden können (Ausnahme: Innere Radius kleiner und ungleich äußerer Radius).
- Es soll wählbar sein, ob die Vertices geteilt werden sollen
- Die Angabe der Segmente der Höhe soll möglich sein
- Die generierte Röhre muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.
- Bei ungültigen Parametern (z.B. Radius ≤ 0) soll keine Röhre erstellt werden.


10.9.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte):	$12 * \text{Ecken} + 6 * \text{Segmente} * \text{Ecken}$
Indices – shared Vertices (Integer, je 4 Byte):	$4 * \text{Ecken} + (\text{Segmente}+1) * \text{Ecken}$
Vertices (je 112 Byte):	$4 * \text{Ecken} + (\text{Segmente}+1) * 2 * \text{Ecken}$

10.9.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **TubeGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **TubeGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.9.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung der Röhre selber läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für Deckel und Boden
    Für jede Ecke
        Speichere aktuellen Vertex des Radiusvektors (innen und aussen)
        Rotiere Radiusvektor
        Erstelle für jede Seite 6 Indices
Wandere durch die zu erstellenden Seiten
    Kopiere entsprechende Vertices aus Deckel/Boden und generiere restliche aus Segmente
    und Höhe für innen und aussen.
    Erstelle 6 Indices für jedes Segment der Seite für innen und aussen. Bei sharedVertices
    verbinde damit aktuelle mit nächster Seite.
```

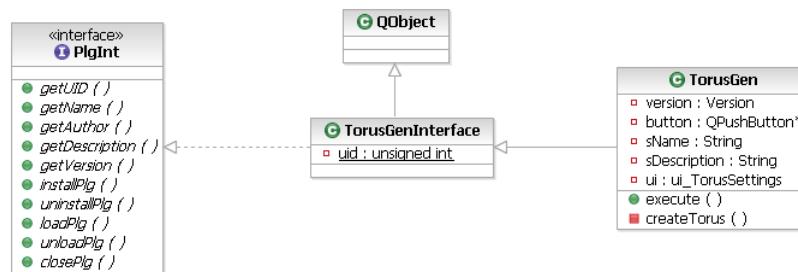
10.9.7 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Breite = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird keine Röhre generiert.

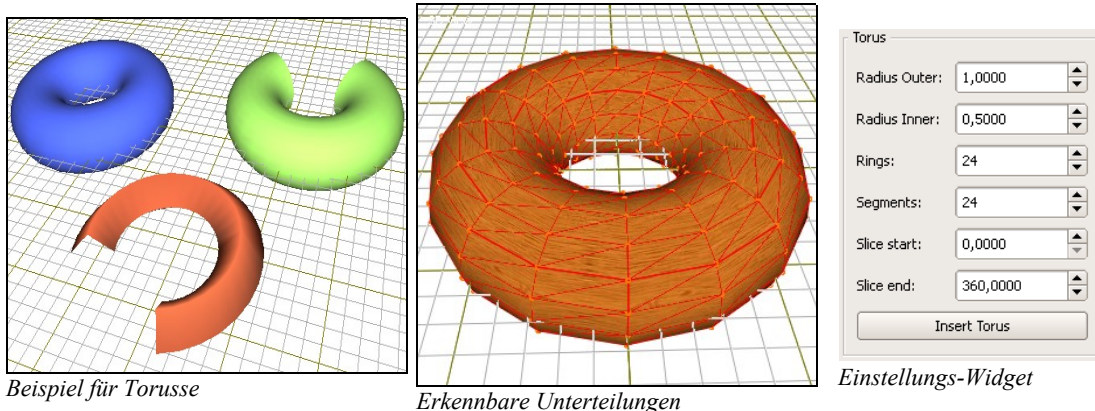
10.10 Torus

Dieses Plugin generiert einen kompletten bzw. einen Teil eines Torus. Falls letzteres erstellt wird, hat der resultierende Torus offene Enden, ist also nicht geschlossen.

10.10.1 UML



10.10.2 Screenshots



10.10.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Der Torus soll über die Parameter 'Radius Outer', 'Radius Inner', 'Rings', 'Segments' und den Start- und Endwinkel ('Slice start/from') beschrieben werden
- Der Detailgrad muss einstellbar sein über eine radiale Einteilung (Rings) und eine Ringunterteilung (Segments)
- Der generierte Torus muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Das Mesh soll über shared Vertices verfügen, d.h. die Dreiecke des Torus sollen sich so viele Vertices wie möglich teilen, sodass die Vertexanzahl gering bleibt
- Bei ungültigen Parametern (z.B. Radius Outer = 0) soll kein Torus erstellt werden
- Sollte der Startwinkel > Endwinkel sein, so soll ein nach innen gestülpter Torus generiert werden (die Facevorderseite zeigt nach innen)


10.10.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

$$\begin{aligned} \text{Indices (Integer, je 4 Byte):} & \quad 6 * \text{Rings} * (\text{Segments} + 1) \\ \text{Vertices (je 112 Byte):} & \quad (\text{Rings} + 1) * (\text{Segments} + 1) \end{aligned}$$

10.10.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **TorusGenInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **TorusGen:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Common objects'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

10.10.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung des Torus selber läuft dabei wie folgt ab (Pseudocode):

```

Prüfen ob Parameter gültig sind
Wandere durch alle zu erstellenden Ringe (0 bis Rings-1)
    Wandere durch alle zu erstellenden Ringsegmente (0 bis Segments-1)
        Generiere den aktuellen Vertex aus aktuellem Ring und Segment
        Falls es nicht der letzte Ring ist, ...
            Erstelle 6 Indices (= 2 Faces) für dieses Segment und verbinde damit aktuellen mit
            nächsten Ring

```

10.10.7 Einflussfaktoren und Randbedingungen

- Um die Anzahl der Vertices gering zu halten (und damit Speicherplatz zu sparen), wird der Torus mit 'Shared Vertices' berechnet. Benachbarte Faces teilen sich so gemeinsam einen oder mehrere Vertices. Dieses Teilen von Vertices wird über die Indices erreicht. So haben zwei Faces, die sich einen Vertex teilen, beide den gleichen Index. Insgesamt werden so ca. 80% der Vertices eingespart, die sonst zu oft vorhanden wären.
- Ein nicht geschlossener Torus (falls er keine 360° einschließt) hat keine geschlossenen Enden. Diese können, falls erwünscht, mit anderen Funktionen des Editors erzeugt werden (Triangulation, ...).

10.10.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. *Radius Outer* = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Torus generiert.
- Die Angabe '*Rings*' zur radialen Unterteilungen gilt für den gesamten Torus. So hat beispielsweise ein halber Torus genauso viele Unterteilungen wie ein 'ganzer' Torus.

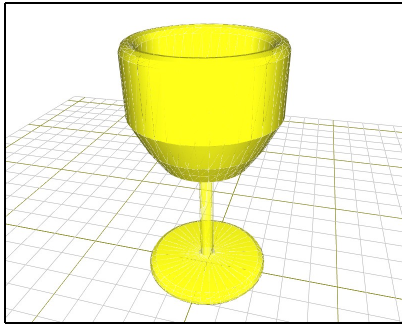
10.11 Goblet (Kelch)

Dieses Plugin erstellt einen Kelch.

10.11.1 UML

Siehe Rotationskörper-Plugin (Kapitel 11.6)

10.11.2 Screenshots



Beispiel eines Kelchs

10.11.3 Funktionale und nicht funktionale Anforderungen

- Der Detailgrad soll über die Anzahl der *Ecken* einstellbar sein
- Der Kelch soll nicht komplett hardkodiert werden.
- Der generierte Kelch muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.

10.11.4 Mengengerüste

Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte): $25 * 6 * \text{Ecken}$
 Vertices (je 112 Byte): $26 * \text{Ecken}$

10.11.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet (Goblet-Plugin ist aus Funktionsgründen das Plugin zur Generierung von Rotationskörpern, da dieses Plugin einen Kelch als Testobjekt generiert):

- **RevolvingInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **Revolving:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (🌀), das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient.

10.11.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung des Kelchs selber läuft dabei wie folgt ab (Pseudocode):
 Siehe Rotationskörper-Plugin (Kapitel 11.6)

10.11.7 Globale Entwurfsentscheidungen

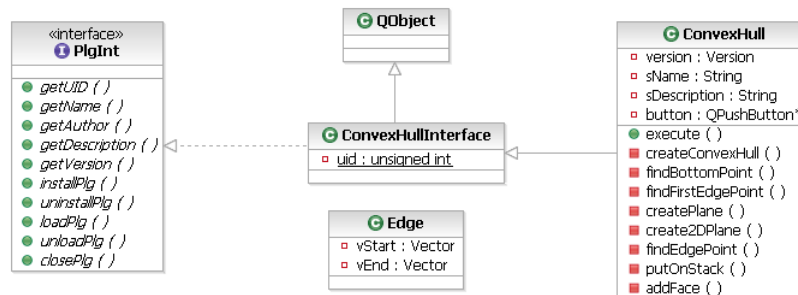
- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Ecken = 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Kelch generiert.
- Um nicht den kompletten Kelch hardkodieren zu müssen wird die Rotationstechnik angewendet. Dadurch muss lediglich die halbe Silhouette gegeben sein.

11 Modifikator-Plugins

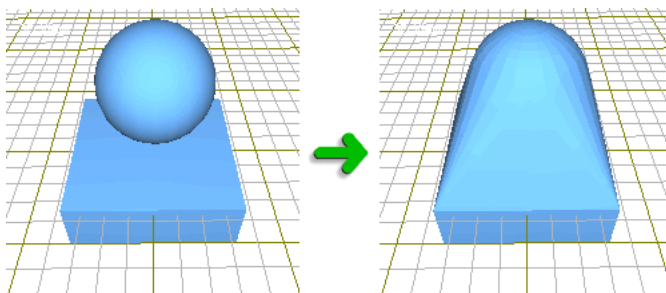
11.1 Convex Hull (Konvexe Hülle)

Dieses Plugin generiert mithilfe der 'Convex Hull'-Technik ein neues Objekt, dass alle selektierten Vertices umschließt. Es ist zum Beispiel besonders hilfreich um komplexe Objekte zu vereinen.

11.1.1 UML



11.1.2 Screenshots



Convexe Hüller zweier selektierter Meshes

11.1.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Der Algorithmus soll eine 'Hülle' berechnen, die alle selektierten Vertices umschließt
- Es sollte ein Algorithmus gewählt werden, der eine gute Laufzeit besitzt
- Ein neues Mesh soll in die Szene eingefügt werden, das die konvexe Hülle enthält
- Die generierte Konvexe Hülle soll außer den Vertexpositionen auch korrekte Normalenvektoren besitzen.


11.1.4 Mengengerüste

Es wird ein neues Mesh generiert, das alle selektierten Vertices enthält. Der Speicherplatzbedarf und die Komplexität dieses Meshes hängt sehr stark von der Anordnung der selektierten Daten ab. Während der Generierung wird außerdem noch ein Stack verwaltet, der alle Kanten (bestehend aus Anfangs- und Endposition) enthält.

11.1.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **ConvexHullInterface**: Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **ConvexHull:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient.
- **Edge:** Speichert eine Kante, bestehend aus einer Start- und einer Endposition

11.1.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung der konvexen Hülle über einen Klick auf den Button angestoßen. Es wird dabei ein neues Mesh erzeugt und in die Szene eingefügt. Die Generierung der konvexen Hülle selber läuft dabei wie folgt ab (Pseudocode):

```
Speichere alle Positionen der selektierten Vertices in eine Liste
Finde den ersten Punkt (Punkt mit minimaler y-Koordinate)
Finde den zweiten Punkt, sodass alle Punkte 'links' daneben liegen
Trage in den Kanten-Stack eine Kante vom ersten zum zweiten Punkt ein
Trage in den Kanten-Stack eine Kante vom zweiten zum ersten Punkt ein

Solange noch elemente im Kanten-Stack sind ...
    Speichere das oberste Element vom Stack und lösche dieses danach
    Finde die Position, die ein Dreieck mit der Kante bildet, bei der alle
    Punkte 'dahinter' liegen (hinter der Ebene die das Dreieck aufspannt)
    Erstelle dieses Dreieck
    Falls dieses noch nicht eingetragen wurde ...
        Füge dieses Dreieck in das neue Mesh ein
        Füge eine neue Kante vom Kanten-Startpunkt bis zum neuen Punkt ein
        Füge eine neue Kante vom neuen Punkt bis zum Kanten-Endpunkt ein

Füge das neue Mesh in die Szene ein
```

11.1.7 Einflussfaktoren und Randbedingungen

- Um den Algorithmus einfach zu halten, kann auf die 'shared Vertices' verzichtet werden, die die Vertexanzahl gering hält.

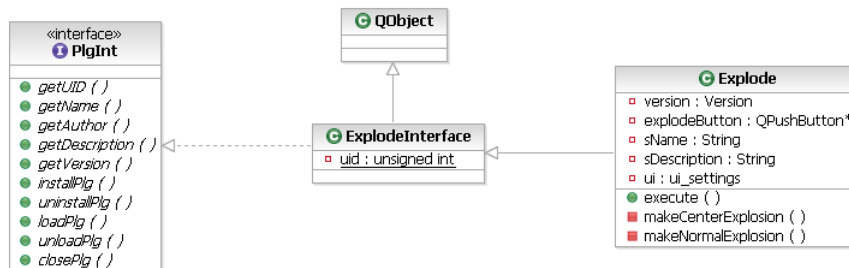
11.1.8 Globale Entwurfsentscheidungen

- Es wurde der sog. 'Grift Warp'-Algorithmus gewählt, der eine sehr gute Laufzeit hat und einfach zu implementieren ist.

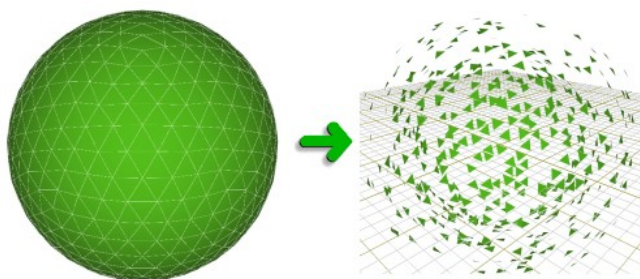
11.2 Explosion

Dieses Plugin lässt Objekte in ihre Faces explodieren/fragmentieren.

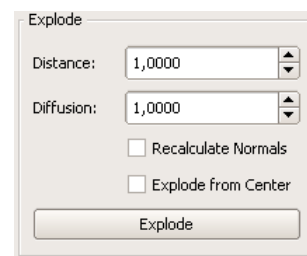
11.2.1 UML



11.2.2 Screenshots



Eine Kugel vor und nach der Manipulation



Einstellungs-Widget

11.2.3 Funktionale und nicht funktionale Anforderungen

- Das Ausmaß der Explosion wird durch den Abstand bestimmt.
- Um den Effekt realistischer wirken zu lassen soll eine Streuung möglich sein.
- Es soll die Möglichkeit bestehen die Normalen neu zu berechnen.
- Die Explosion soll entweder entlang der Normalen oder vom Zentrum der Auswahl aus durchgeführt werden.
- Bei ungültigen Parametern (z.B. Abstand < 0) soll keine Veränderung angewandt werden.


11.2.4 Mengengerüste

SharedVertices werden aufgebrochen, wodurch sich die Anzahl der Vertices ändert. Die Änderung des Speicherplatzbedarfs ist wie folgt:

Indices (Integer, je 4 Byte): bleibt gleich
 Vertices (je 112 Byte): ca. 4 mal mehr

11.2.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **ExplodeInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **Explode:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

11.2.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Explosion über das Einstellungs-Widget angestoßen. Hier wird das bestehende Mesh für den Szene-Graphen bearbeitet und an diesen weitergeleitet, damit es in der Szene aktualisiert wird. Die Modifikation des Objekts läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für jedes Objekt
  SharedVertices ggf. splitten
  Falls gewünscht, für jeden Vertex Normalenvektor neu berechnen
  Jeden Vertex um Abstand mit einberechnung der Streuung verschieben
```

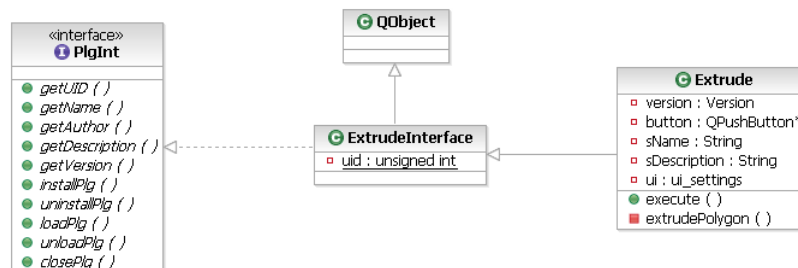
11.2.7 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen (z.B. Abstand < 0), werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Block generiert.

11.3 Extrude

Dieses Plugin erzeugt eine Erhebung beziehungsweise Einsenkung in einem selektierten Face. Bei der Selektion eines Meshes werden alle Faces des Meshes extrudiert.

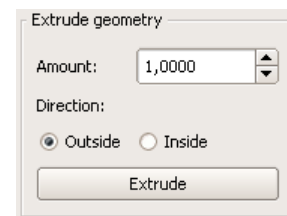
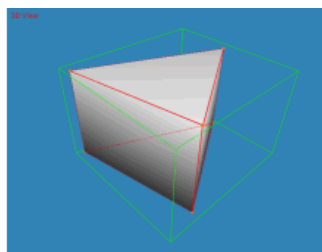
11.3.1 UML



11.3.2 Screenshots



Beispiel eines extrudierten Faces



Einstellungs-Widget

11.3.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Die Höhe der Extrusion soll über die Parameter 'Amount' beschrieben werden
- Die Bestimmung ob eine Erhöhung oder eine Absenkung gewünscht ist, wird über den Parameter 'Direction' entschieden
- Die neu hinzugekommene Fläche muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen
- Bei ungültigen Parametern (z.B. Amount = 0) soll keine Kugel erstellt werden


11.3.4 Mengengerüste

Jedes zu modifizierende Face (3 Vertices) wird zu 6 Vertices extrapoliert. Dabei werden diese Vertices zu insgesamt 5 Faces (1 Top, 1 Bottom, 6 Side) durch Verknüpfung der Indices verbunden. Der Speicherplatzbedarf nach der Extrusion ist wie folgt:

Indices (Integer, je 4 Byte): $(3 + 3 + 6 * 3) * \text{Faceanzahl}$
 Vertices (je 112 Byte): $(3 + 3) * \text{Faceanzahl}$

11.3.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **ExtrudeInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **Extrude:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient.

11.3.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Extrudierung über das Einstellungs-Widget angestoßen. Hierzu werden die aktuell selektierten Meshes und Einzelpolygone vom Scenograph angefordert und anschließend mit dieser Menge die Modifikation durchgeführt.

Danach wird der Scenograph von der Modifikation über die Notify-Funktion in Kenntnis gesetzt.

11.3.7 Einflussfaktoren und Randbedingungen

- Um die Anzahl der Vertices gering zu halten (und damit Speicherplatz zu sparen), werden die neu erzeugten Vertices pro Face als 'Shared Vertices' berechnet. Benachbarte Faces teilen sich so gemeinsam einen oder mehrere Vertices. Dieses Teilen von Vertices wird über die Indices erreicht. So haben zwei Faces, die sich einen Vertex teilen, beide den gleichen Index enthalten. Insgesamt werden so 75% der Vertices eingespart, die sonst zu oft vorhanden wären.

Hypothetische Anzahl bei Verzicht auf 'Shared Vertices': 24 Vertices

Realisierter Stand: 6 Vertices

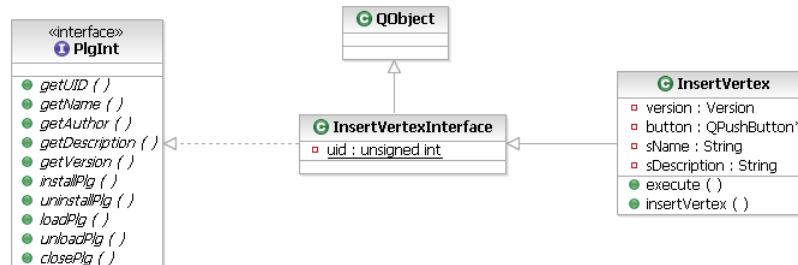
11.3.8 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen, wird vor der eigentlichen Extrudierung der Parameter 'Amount' geprüft. Sollten diese keinen Sinn machen ($=0$), wird keine Extrudierung durchgeführt.

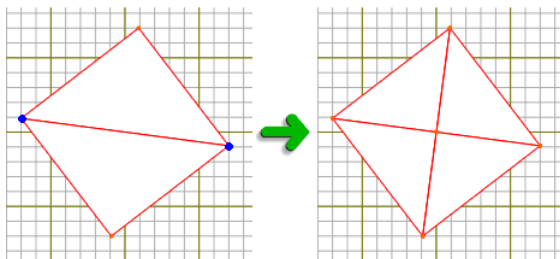
11.4 Insert vertex (Vertex einfügen)

Dieses Plugin fügt zwischen zwei Vertices einen neuen Vertex ein und verbindet ihn so, sodass dieser in die benachbarten Faces eingebunden ist.

11.4.1 UML



11.4.2 Screenshots



Neu eingefügter und verbundener Vertex (blaue Vertices sind markiert)

11.4.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es soll zwischen zwei selektierten Vertices einen neuen Vertex einfügen
- Der neu erstellte Vertex soll in die benachbarten Faces eingebunden werden, diese also gegenfalls teilen
- Der neue Vertex muss auch korrekte Normalenvektoren und Texturkoordinaten besitzen


11.4.4 Mengengerüste

Jedes an den selektierten Vertices gebundene Face wird manipuliert. Nach der Manipulation hat ein Mesh mit i Indices und v Vertices folgenden Speicherplatzbedarf:

Indices (Integer, je 4 Byte): $i + 3$ (Best Case), $i * 3$ (Worst Case, falls alle Faces geteilt werden müssen)
 Vertices (je 112 Byte): $v + 1$

11.4.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **InsertVertexInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **InsertVertex:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient.

11.4.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird das Einfügen eines Vertices über einen Klick auf den Button angestoßen. Das Einfügen des neuen Vertex läuft dabei wie folgt ab (Pseudocode):

```
Prüfe, ob es genau zwei Vertices selektiert wurden (nur Position wird überprüft)
Speichere die an den Vertices gebundenen Polygone

Wandere durch alle daran gebundenen Polygone
  Rufe 'insertVertex' für das Polygon auf um darin ein neues Vertex zu erstellen

insertVertex:
Wandere durch alle Faces des Polygons
  Falls der Vertex im aktuellen Face eingefügt werden soll, ...
    Erstelle neuen Vertex
    Speichere die Indices des Faces, zum späteren Splitten
  Ansonsten, ...
    Füge das Face unverändert ein

Wandere durch alle zu splittenden Faces
  Erstelle 6 Indices (= 2 Faces) das das aktuelle Face am neuen Vertex teilt
  Füge die 2 neuen Faces ein
```

11.4.7 Einflussfaktoren und Randbedingungen

- Es müssen genau 2 Vertices selektiert werden. Würde man mehr erlauben, würde es viel zu viele Kombinationen geben, diese in die Faces zu integrieren und zu verbinden.
- Die selektierten Vertices müssen nicht komplett individuell sein. Es reicht vollkommen aus, wenn diese zusammen 2 verschiedene Positionen besitzen. So kann man beispielsweise 6 Vertices selektieren, wo jeweils 3 davon auf der gleichen Position liegen. Dies macht es in der Handhabung sehr viel flexibler und komfortabler, da man nur sehr selten 2 komplett individuelle Vertices auswählen kann.

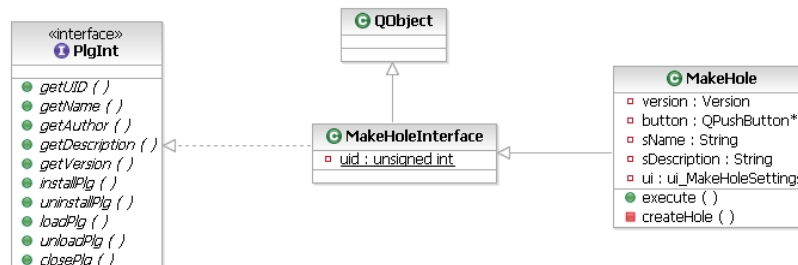
11.4.8 Globale Entwurfsentscheidungen

- Das Einfügen des Vertex zerstört die eventuell vorhanden 'shared Vertices' nicht. Ebenso wird das neu eingefügte Vertex so in die Faces gebunden, dass sich diese Faces das Vertex teilen. Dies minimiert die Zahl der Vertices.

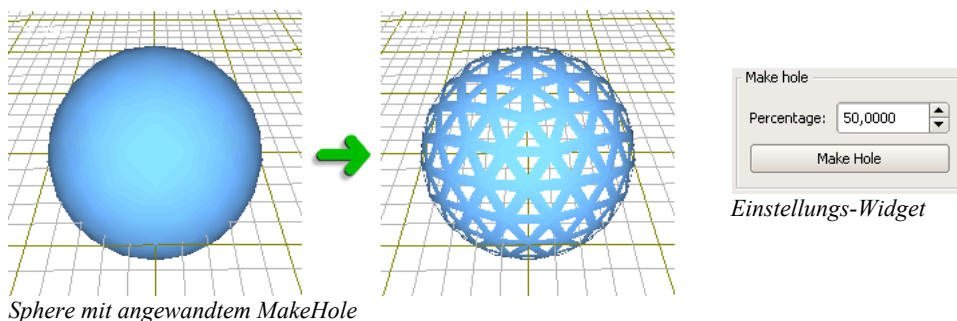
11.5 Make Hole

Dieses Plugin fügt in alle selektierten Faces neue Vertices ein (genauer gesagt 3 Stück) und erstellt daraus mithilfe der Originalvertices ein Loch mit einem bestimmten Verhältnis.

11.5.1 UML



11.5.2 Screenshots



11.5.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Das Verhältnis zwischen Loch und dem entstehendem Rest (quasi der 'Rahmen') soll einstellbar sein ('Percentage')
- Es soll kein neues Mesh erzeugt, sondern das selektierte Mesh direkt manipuliert werden
- Das Model muss nach der Manipulation auch korrekte Normalenvektoren und Texturkoordinaten besitzen

11.5.4 Mengengerüste


Jedes Face des/der selektierten Mesh/Meshes wird manipuliert. Nach der Manipulation hat ein Mesh mit i Indices und v Vertices folgenden Speicherplatzbedarf:

Indices (Integer, je 4 Byte): $i * 6$

Vertices (je 112 Byte): $v * 2$ (falls es keine shared Vertices gibt), sonst im Worst-Case: $i * 2$

11.5.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **MakeHoleInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **MakeHole:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

11.5.6 Architektursicht – Laufzeit

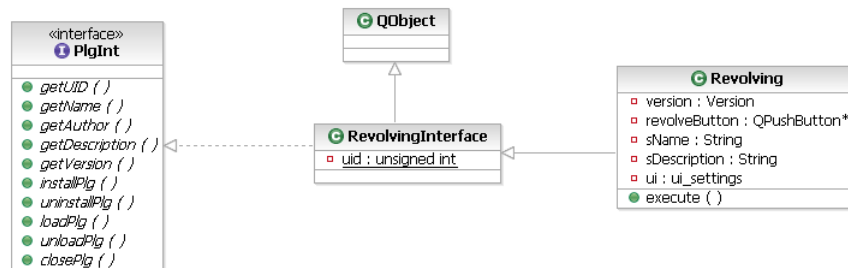
Nach dem Laden des Plugins über das Pluginsystem wird die Generierung der Löcher über das Einstellungs-Widget angestoßen. Die selektierten Meshes dabei direkt manipuliert. Die Generierung der Stacheln selber läuft dabei wie folgt ab (Pseudocode):

```
Wandere durch alle selektierten Meshes
  Wandere durch Polygone des aktuellen Meshes
    Wandere durch alle Faces des Polygons
      Erstelle 3 neue Vertices (die inneren Vertices)
      Erstelle 18 Indices (= 6 Faces aus den 3 neuen + 3 alten Vertices)
      Füge diese Daten ein (ersetze alte Daten)
```

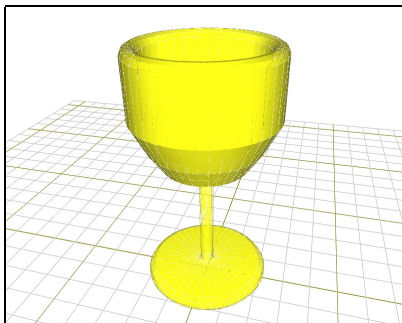
11.6 Revolving (Rotationskörper)

Dieses Plugin rotiert eine 2D Linie um eine Achse und generiert dadurch einen 3D Körper.

11.6.1 UML



11.6.2 Screenshots



Beispiel eines Rotationskörpers

11.6.3 Funktionale und nicht funktionale Anforderungen

- Ein Winkel soll bestimmen, wie weit rotiert werden soll.
- Mit der Anzahl der Ecken kann bestimmt werden wie rund der Rotationskörper wird.
- Die Rotationsachse soll wählbar sein.
- Der Rotationskörper muss außer den Vertexpositionen noch korrekte Normalenvektoren und Texturkoordinaten besitzen.
- Es soll die Möglichkeit gegeben sein dass der erste und letzte Punkt der 2D Linie verbunden werden.
- Bei ungültigen Parametern soll keine Veränderung stattfinden.

11.6.4 Mengengerüste


Das generierte Mesh wird zuerst vorberechnet und erst dann in die Szene eingefügt. Der Speicherplatzbedarf bei der Erstellung ist wie folgt:

Indices (Integer, je 4 Byte): $(\text{Vertices der Linie} - 1) * 6 * \text{Ecken}$
 Vertices (je 112 Byte): $\text{Vertices der Linie} * \text{Ecken}$

11.6.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **RevolvingInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **Revolving:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

11.6.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung über das Einstellungs-Widget angestoßen. Hier wird ein neues Mesh für den Szene-Graphen erstellt und an diesen weitergeleitet, damit es in die Szene eingefügt wird. Die Generierung des Rotationskörpers selber läuft dabei wie folgt ab (Pseudocode):

```
Prüfen ob Parameter gültig sind
Für jede Ecke
    Für jeden Vertex der Linie
        Generiere den aktuellen Vertex durch rotieren des Vorgängers
    Erstelle 6 Indices für jedes Segment und verbinde damit aktuelle mit nächster Seite.
```

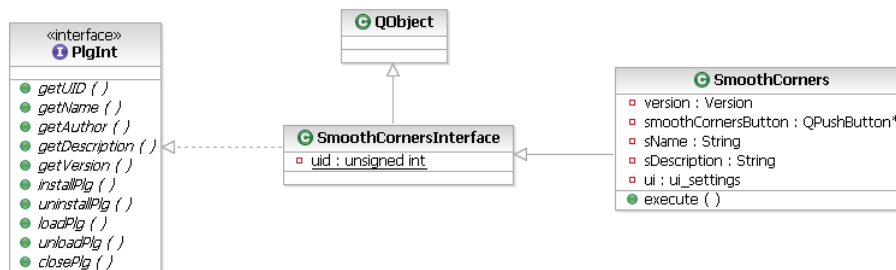
11.6.7 Globale Entwurfsentscheidungen

- Um keine unvorhersehbaren Ergebnisse zu erhalten oder unnötige Berechnungen durchzuführen, werden vor der eigentlichen Generierung die Parameter geprüft. Sollten diese keinen Sinn machen, wird kein Rotationskörper erstellt.

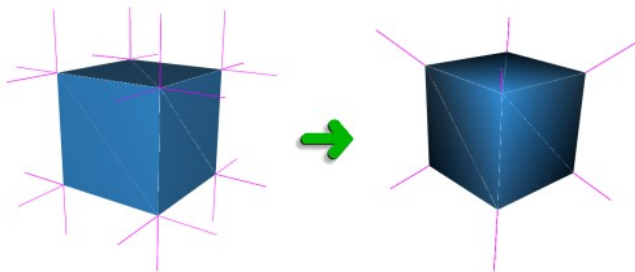
11.7 Smooth Corners (Weiche Ecken)

Dieses Plugin lässt die Ecken von Objekten weich erscheinen (indem er die Normalenvektoren angleicht, die für die Beleuchtung wichtig sind).

11.7.1 UML



11.7.2 Screenshots



Eine Box vor und nach der Manipulation

11.7.3 Funktionale und nicht funktionale Anforderungen


- Es sollen nur Normalen bearbeitet und keine Positionen geändert oder Vertices hinzugefügt/entfernt werden.

11.7.4 Mengengerüste

Vertices und Indices werden nicht hinzugefügt oder entfernt, deshalb ändert sich deren Anzahl nicht.

11.7.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **SmoothCornersInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **SmoothCorners:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon (), das zur besseren graphischen Erkennbarkeit dient.

11.7.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird der Effekt über das Icon angestoßen. Hier wird das bestehende Mesh für den Szene-Graphen bearbeitet und an diesen weitergeleitet, damit es in der Szene aktualisiert wird. Die Modifikation des Objekts läuft dabei wie folgt ab (Pseudocode):

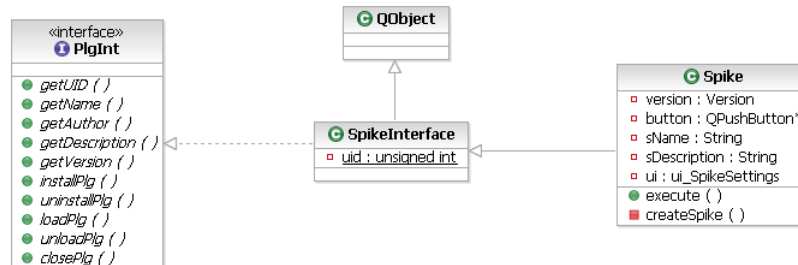
```

Für jedes Objekt
  Alle Vertices mit selber Position suchen
  Für jede Ecke (Anzahl Vertices mit selber Position >= 3) gemeinsame Normale berechnen
  
```

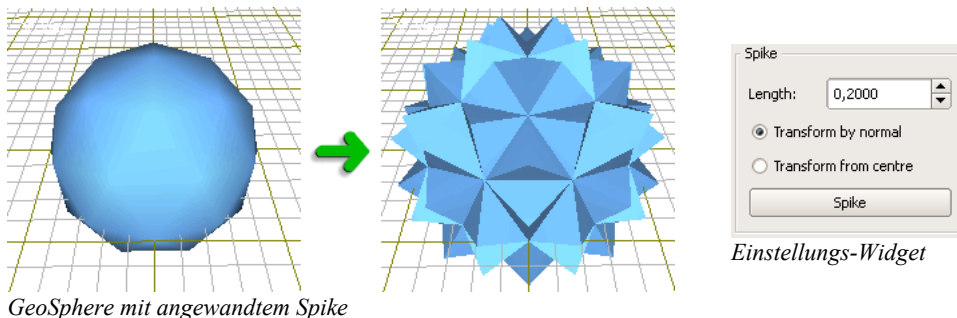
11.8 Spike (Stacheln)

Dieses Plugin fügt in alle selektierten Faces einen neuen Vertex ein und erstellt durch Verschiebung daraus einen Stachel.

11.8.1 UML



11.8.2 Screenshots



11.8.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Die Länge der generierten Stacheln soll einstellbar sein
- Der Stachel soll entweder mithilfe des Normalenvektors oder um das Modellzentrum verschoben werden
- Es soll kein neues Mesh erzeugt, sondern das selektierte Mesh direkt manipuliert werden
- Das Model muss nach der Manipulation auch korrekte Normalenvektoren und Texturkoordinaten besitzen

11.8.4 Mengengerüste

Jedes Face des/der selektierten Meshes wird manipuliert. Nach der Manipulation hat ein Mesh mit i Indices und v Vertices folgenden Speicherplatzbedarf:

Indices (Integer, je 4 Byte): $i * 3$


Vertices (je 112 Byte): $i * 3$

D.h.: Am Ende gibt es zu jedem Index genau einen Vertex, da sich jeder Vertex durch seine Position und Normalenvektor unterscheidet. Daher sind hier keine 'shared vertices' möglich.

11.8.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **SpikeInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.

- **Spike:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Button erstellt, welcher in die 'Modifiers'-Groupbox der Toolbox eingefügt wird. Dieser enthält außerdem anstatt eines Namens ein Icon () , das zur besseren graphischen Erkennbarkeit dient. Außerdem wird ein Widget erstellt, das zum Einstellen der Parameter dient (siehe Screenshots).

11.8.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Generierung der Stacheln über das Einstellungs-Widget angestoßen. Die selektierten Meshes werden direkt manipuliert. Die Generierung der Stacheln selber läuft dabei wie folgt ab (Pseudocode):

```
Wandere durch alle selektierten Meshes
  Wandere durch Polygone des aktuellen Meshes
    Wandere durch alle Faces des Polygons
      Erstelle 9 neue Vertices (3 für jedes neue Face)
      Erstelle 9 Indices (= 3 Faces)
      Füge diese Daten ein (ersetze alte Daten)
```

11.8.7 Einflussfaktoren und Randbedingungen

- Es muss auf die 'shared Vertices' verzichtet werden, da sich kein Face mit einem anderen ein oder mehrere Vertices teilen können. Der Grund ist, dass für jede Vertexposition der Normalenvektor immer verschieden ist (aufgrund der Form des Stachels).

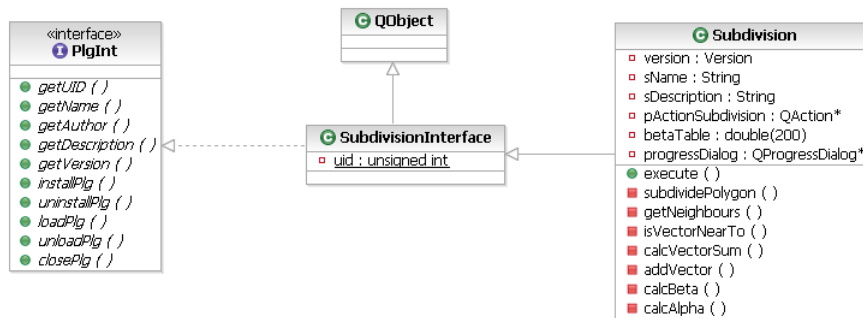
11.8.8 Globale Entwurfsentscheidungen

- Es werden nicht nur positive Längen akzeptiert, sondern auch negative. Positive Längen bewirken, dass die Stacheln nach außen 'wachsen', negative Längen lassen diese nach innen 'wachsen'.

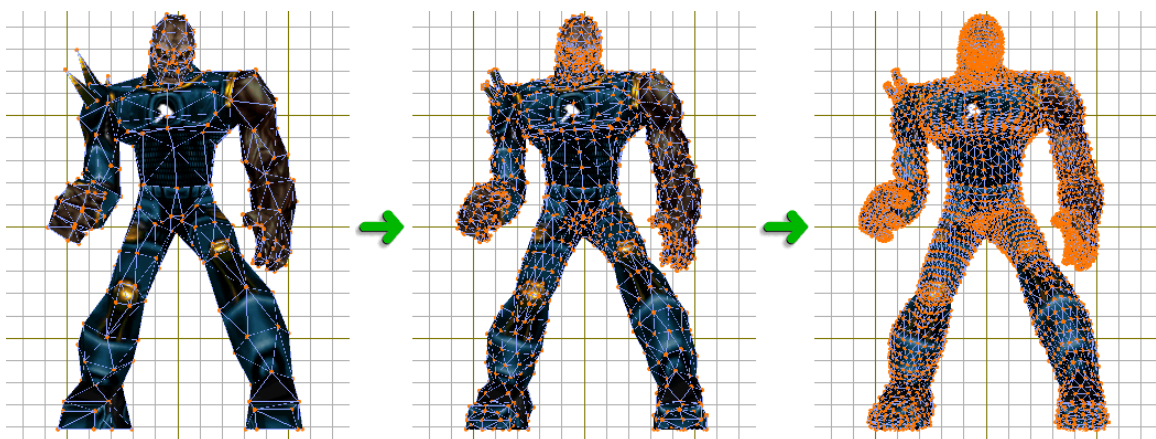
11.9 Subdivision

Subdivision ist ein Verfahren, dass ein Modell durch das Einfügen neuer Vertices plastischer macht. Jedes selektierte Face eines Meshes wird dabei in 4 Faces geteilt, wobei die Originalvertices nach einer bestimmten Formel verschoben werden. Subdivision hat den Vorteil, dass das Originalmodell nicht sehr detailliert modelliert werden muss, um am Ende ein plastisches Modell zu erhalten. Außerdem lassen sich so Modelle in verschiedenen Detailgraden berechnen, die beispielsweise in Spielen verwendet werden können (sog. LOD – Weiter entfernte Objekte werden nicht so detailliert gerendert als nahe Objekte).

11.9.1 UML



11.9.2 Screenshots



Original, Subdivision Level 1, Subdivision Level 2

11.9.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Das Modell soll nach der Bearbeitung feiner aufgelöst berechnet sein, sodass es runder und plastischer wirkt.
- Es soll kein neues Mesh erzeugt, sondern das selektierte Mesh direkt manipuliert werden
- Die neu berechneten Vertices müssen auch korrekte Normalenvektoren und Texturkoordinaten besitzen

11.9.4 Mengengerüste

Jedes selektierte Face wird direkt durch Subdivision verfeinert. Nach der Manipulation hat ein Mesh mit i Indices und v Vertices folgenden Speicherplatzbedarf:

Indices (Integer, je 4 Byte): $i * 4$

Vertices (je 112 Byte): $i * 4$

11.9.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **SubdivisionInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **SubdivisionFace:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Menü in das 'Modify'-Menü eingefügt, mit dessen Hilfe der Vorgang ('Subdivision Loop'-Algorithmus) in Gang gesetzt werden kann.

11.9.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird die Berechnung des verfeinerten Modells über einen Klick auf den Menüeintrag angestoßen. Der Algorithmus (Subdivision Loop) läuft dabei wie folgt ab (Pseudocode):

```

Speichere alle Vertices und Indices des selektierten Meshes ab
Generiere die vorberechnete Liste für Beta
Wandere durch alle daran Polygone des Meshes
    Rufe 'subdividePolygon' auf das aktuelle Polygon auf

subdividePolygon:
Wandere durch alle Faces des Polygons
    Erstelle 3 neue Vertices (in die Mitte jeder Kante)
    Erstelle 12 Indices (= 4 Faces)
    Wandere durch alle 3 'alten' Originalvertices durch
        Erstelle eine Liste mit allen Nachbarvertices zu diesem aktuellen Vertex
        Berechne Beta aus der Anzahl der gefundenen Nachbarvertices (n):


$$\beta(n) = \frac{1}{n} \left( \frac{5}{8} - \left( \frac{3}{8} + \frac{1}{4} * \cos\left(\frac{2\pi}{n}\right) \right) \right)$$


        Berechne die Summe aller Positionen der Nachbarn
        Berechne aus diesen Daten die neue Position v des Vertex:


$$v_{new} = v_{old} (1 - n * \beta(n)) + \beta(n) v_{neighboursum}$$


    Füge die 4 Faces anstelle des aktuell zu bearbeiteten Faces ein
  
```

11.9.7 Einflussfaktoren und Randbedingungen

- Da die Mesh-Struktur auf den verwendeten Subdivision-Algorithmus nicht ausgerichtet ist, dauert die Berechnung von großen Modellen recht lange (vorallem die Berechnung der benachbarten Vertices). Aus diesem Grund wird bei der Berechnung ein Fortschrittsbalken eingeblendet, falls diese ein bestimmtes Zeitlimit übersteigt.
- Es gibt verschiedene Formeln zur Berechnung von Beta und des neuen Positionsvektors. Die oben gezeigten Formeln haben sich allerdings für unsere Zwecke als die brauchbarsten herausgestellt, da sie die besten Ergebnisse liefern.

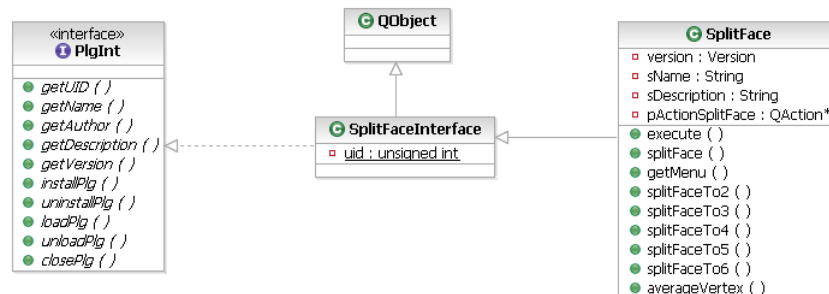
11.9.8 Globale Entwurfsentscheidungen

- Es wurde der sog. 'Subdivision Loop'-Algorithmus gewählt, der auf Dreiecksbasis (gegenüber Algorithmen die auf Quadrate angewandt werden) arbeitet und relativ einfach zu implementieren ist.
- Es werden viele trigonometrische Berechnungen durchgeführt. Um diese zu beschleunigen, werden diese in einer Liste vorberechnet, auf die der Algorithmus zugreifen kann. Dies spart die mehrmalige Berechnung ein und desselben Wertes.
- Um den Algorithmus einfach zu halten, wurde verzichtet, die Vertexanzahl über die 'shared Vertices' gering zu halten.

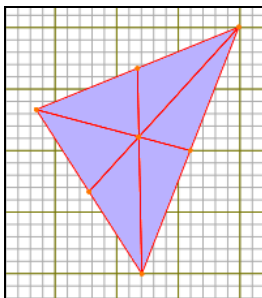
11.10 Split face (Face aufteilen)

Dieses Plugin fügt zwischen zwei Vertices einen neuen Vertex ein und verbindet ihn so, sodass dieser in die benachbarten Faces eingebunden ist.

11.10.1 UML



11.10.2 Screenshots



Ein in 6 Teile gesplittetes
bzw. geteiltes Dreieck

11.10.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Alle selektierten Faces sollen in 2, 3, 4, 5 oder 6 Teile gesplittet/geteilt werden können
- Der neue Vertex muss auch korrekte Normalenvektoren und Texturkoordinaten besitzen

11.10.4 Mengengerüste

Jedes selektierte Face wird direkt manipuliert. Nach der Manipulation (Splitten in n Teile) hat ein Mesh mit i Indices und v Vertices folgenden Speicherplatzbedarf:

$$\begin{aligned} \text{Indices (Integer, je 4 Byte):} & \quad i * n \\ \text{Vertices (je 112 Byte):} & \quad i * n / 3 \end{aligned}$$

11.10.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **SplitInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von **QObject**. **QObject** wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **SplitFace:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Menü mit Untermenüeinträgen in das 'Modify'-Menü eingefügt, mit dessen Hilfe der Vorgang für eine bestimmte Teilungszahl in Gang gesetzt werden kann.

11.10.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird das Splitten über einen Klick auf den jeweiligen Menüeintrag angestoßen. Das Splitten läuft dabei wie folgt ab (Pseudocode am Beispiel des Splits in 6 Subfaces):

```
Wandere durch alle daran selektierten Polygone
  Wandere durch alle Faces des Polygons
    Erstelle 4 neue Vertices in diesem Face (Einen in der Mitte, 3 in die Mitte jeder
Kante)
    Erstelle 18 Indices (= 6 Faces)
    Füge die 6 neuen Faces anstelle des aktuell bearbeiteten Faces ein
```

11.10.7 Einflussfaktoren und Randbedingungen

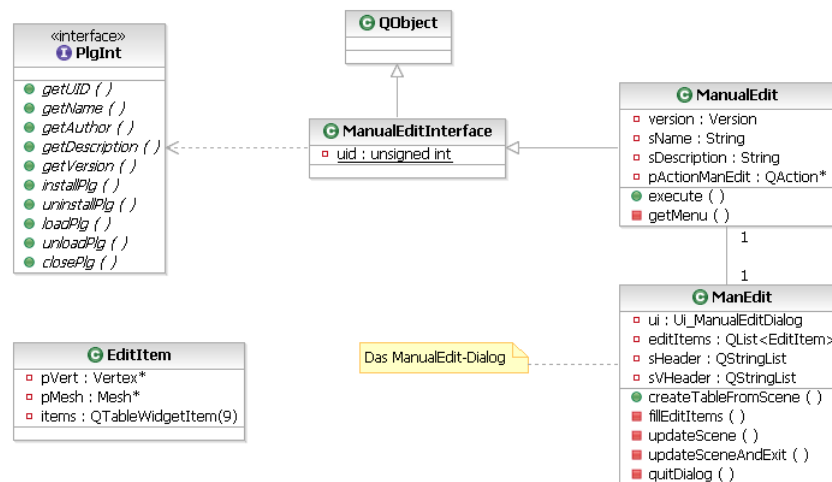
- Der Einfachheit halber kann auf die Verwendung von 'shared Vertices' verzichtet werden, um die Vertexanzahl zu minimieren.

12 Sonstige Plugins

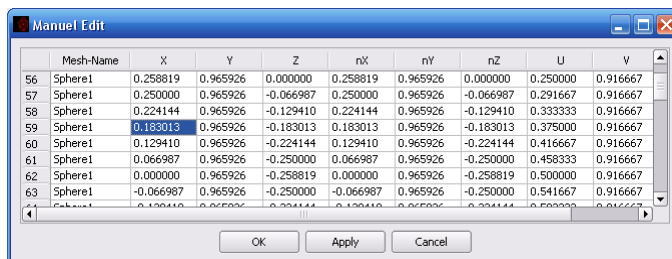
12.1 Manual edit (Manuelle Bearbeitung)

Um Vertexdaten (z.B. die Position) komfortabel manuell zu bearbeiten, wurde dieses Plugin entwickelt. Es bietet einen Dialog mit einer Tabelle, in der alle selektierten Vertices aufgelistet sind und dort bearbeitet werden können.

12.1.1 UML



12.1.2 Screenshots



Das Fenster zur Bearbeitung der Vertexdaten

12.1.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es soll alle selektierten Vertices in einer Tabelle darstellen
- Dargestellt werden sollen der Name des Meshes (zur leichteren Zuordnung), an dem der Vertex gebunden ist, die Vertexposition (X/Y/Z), der Normalenvektor (nX/nY/nZ) sowie die Texturkoordinate der 1. Texturschicht (U/V)
- Die Szene soll über einen 'Apply'-Button aktualisiert werden können, ohne das Dialogfenster immer schließen zu müssen um den Unterschied sehen zu können.
- Die Darstellung der Werte soll eine Genauigkeit von 6 Nachkommastellen haben

12.1.4 Mengengerüste

Für jeden selektierten Vertex wird eine Struktur (EditItem) angelegt, die die Zuordnung zwischen QTableWidgetItem und dem zu bearbeitenden Vertex speichert. Bei einem Öffnen des Dialogs wird die Liste neu generiert.

12.1.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **ManualInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **ManualEdit:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Menüeintrag in das 'Modify'-Menü eingefügt, mit dessen Hilfe auf das ManualEdit-Dialog zugegriffen werden kann.
- **ManEdit:** Diese Klasse stellt das Dialogfenster mithilfe von Qt dar. Es speichert eine Zuordnung zwischen Tabelleneinträgen und Vertices in der Szene, damit die Handhabung vereinfacht wird.
- **EditItem:** Hier wird die Zuordnung zwischen Tabelleneintrag, dem selektierten Vertex und das daran gebundene Mesh gespeichert.

12.1.6 Architektursicht – Laufzeit

Nach dem Laden des Plugins über das Pluginsystem wird Plugin über den Eintrag im Menü 'Modify' aufgerufen. Beim Öffnen des Dialogfensters wird die Tabelle neu angelegt, sodass die selektierten Vertices aufgelistet werden. Pro Vertex wird eine Zeile mit 9 Spalten angelegt und verwaltet.

12.1.7 Einflussfaktoren und Randbedingungen

- Um die Zuordnung eines in der Tabelle befindlichen Vertex genauer bestimmten zu können, wird auch der Mesh-Name angezeigt, an dem es gebunden ist.
- Um die Handhabung mit den angezeigten Vertexdaten zu vereinfachen, wird eine Struktur zur Zuordnung verwendet.

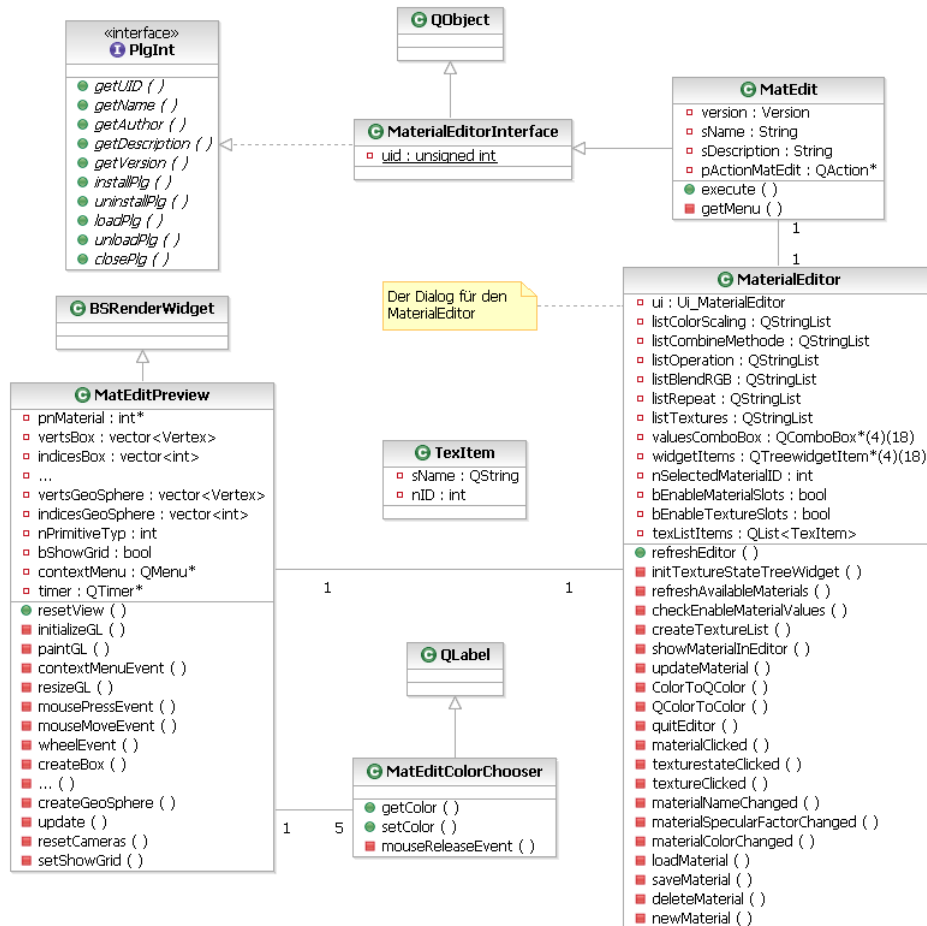
12.1.8 Globale Entwurfsentscheidungen

- Falls ein Mesh-Name nicht bekannt ist (z.B. wenn über den Selektionsmodus 'Vertex' selektiert wird), wird die Spalte dafür im Dialog leer gelassen.
- Es können alle Zellen bearbeitet werden, bis auf die für die Mesh-Namen, da diese nur der leichten Zuordnung und Auffindung einer bestimmten Zeile dienen.

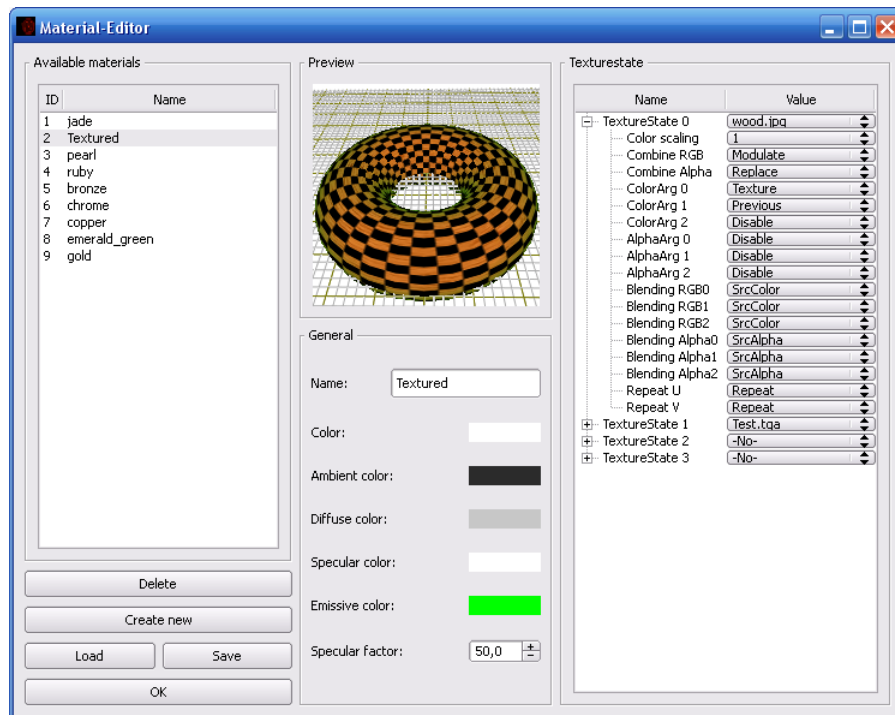
12.2 Materialeditor

Um die Oberflächeneigenschaften, die in den sog. Materialien gespeichert sind, zu verwalten, wird dieser Material-Editor verwendet. Hier können alle Eigenschaften eingestellt und deren Wirkung bzw. Aussehen an einem Vorschauobjekt begutachtet werden.

12.2.1 UML



12.2.2 Screenshots



Der Material-Editor

12.2.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es sollen alle verfügbaren Materialeigenschaften einstellbar sein
- Alle verfügbaren Materialien soll aufgelistet werden (mit Name und ID)
- Das aktuell ausgewählte Material soll in einem Vorschaufenster dargestellt werden. Änderungen einer Materialeigenschaft muss eine aktualisierte Vorschau zur Folge haben
- Das Vorschauobjekt soll gewechselt werden können (z.B. zu Box, Torus, Sphere, ...)
- Texturen sollen hier bequem geladen werden können
- Materialien müssen verwaltet werden können. Darunter zählt die Erstellung/Einfügen neuer Materialien (New), das Löschen eines Materials (Delete), sowie das Laden und Speichern in und aus einer Datei (Load/Save)

12.2.4 Mengengerüste

Es wird direkt auf das Material zugegriffen und bearbeitet. Daher müssen fast ausschließlich GUI-Elemente verwaltet werden, insbesondere die Liste der verfügbaren Materialien und die ComboBoxen zum Einstellen der Texturestates der Texturschichten 0-4 (Die Kombinationseinstellungen der Schichten, siehe dazu Kapitel über Renderer) und Laden der Texturen.

12.2.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **MaterialEditorInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **MatEdit:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Menüeintrag in das 'Material'-Menü eingefügt, mit dessen Hilfe auf das Texturkoordinaten-Editor-Dialog zugegriffen werden kann.

- **MaterialEditor:** Diese Klasse stellt das Dialogfenster mithilfe von Qt dar und enthält vorallem die Liste der verfügbaren Materialien, die Texturestate-Einstellungen und das Vorschaufenster.
- **MatEditPreview:** Das Vorschaufenster für die Materialien. Es enthält eine Reihe von Standardobjekten (Box, Sphere, Torus, Grid, ...), die zur Ansicht des Materials dienen. Verwendet wird der OpenGL-Renderer. Einstellungen, wie z.B. Einschalten des Grids oder die Auswahl des Vorschauobjekts, lassen sich über ein Kontextmenü vornehmen.
- **MatEditColorChooser:** Stellt ein Label (eine farbige Fläche) dar, die bei einem Klick ein Farbauswahldialog öffnet (mit der Möglichkeit Alpha einzustellen). Wird das Auswahldialog geschlossen, färbt sich das Label in dieser Farbe.
- **TexItem:** Die Struktur zur Zuordnung einer Textur (Name, ID) zu einem ComboBox-Eintrag (Index)

12.2.6 Architektursicht – Laufzeit

Das ausgewählte Material wird direkt manipuliert. Während der Bearbeitung werden fast ausschließlich GUI-Elemente aktuell gehalten, insbesondere die Listen und die Baumstruktur (mit 18 ComboBoxen je Schicht).

12.2.7 Einflussfaktoren und Randbedingungen

- Das Laden von Texturen sollte hier möglich sein (siehe Anforderungen). Daher wurde es in die ComboBox eingebaut, die neben 'Texturestate x' liegt. Neben der Auswahl einer Textur wurde ein 'Load from file...' eingefügt, mit dessen Hilfe Bilddateien geladen werden können (ein geeignetes geladenes TextureLoaderPlugin vorausgesetzt, siehe dazu Kapitel zum Plugin 'TextureLoaderPack')
- Die einstellbaren Farben des Materials können Transparenz speichern (RGBA). Daher muss auf das modifizierte Farbauswahldialog zugegriffen werden, das Qt ebenfalls anbietet.

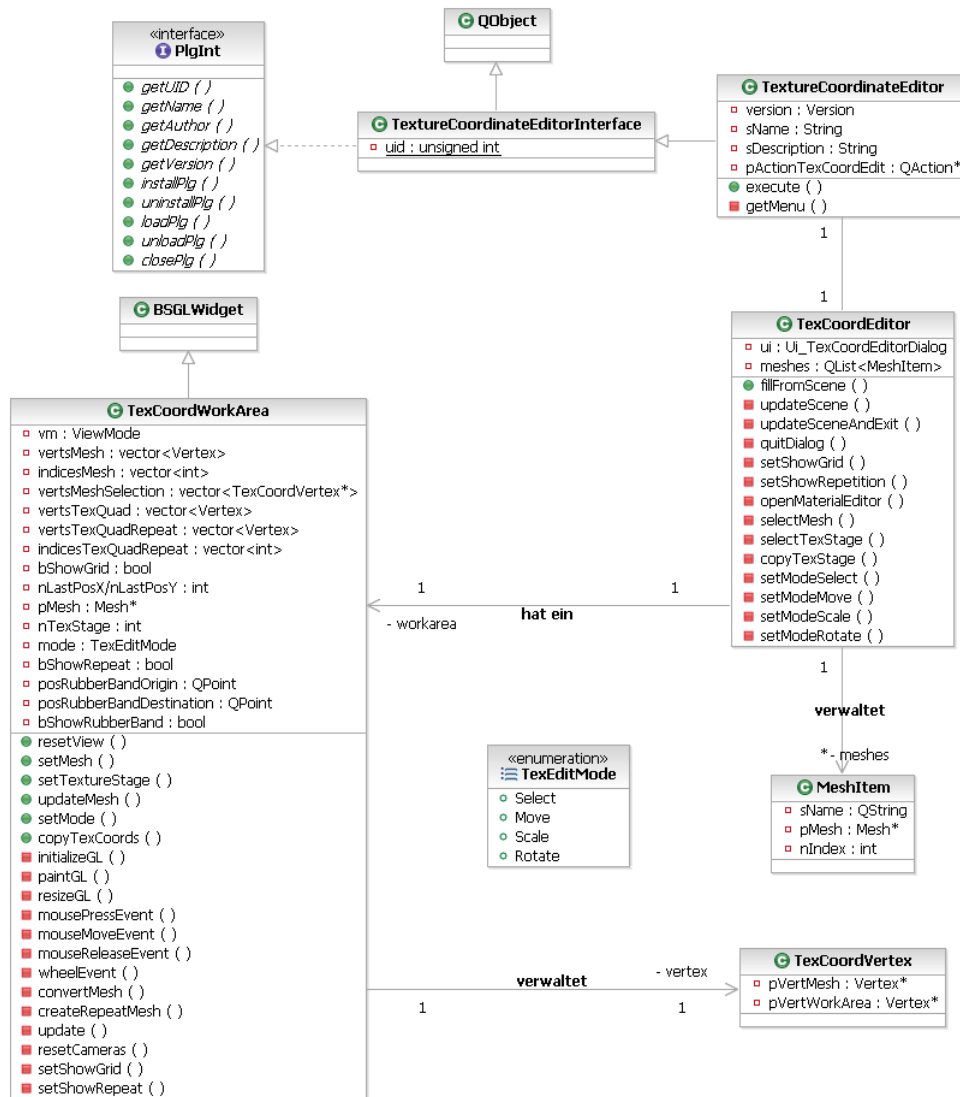
12.2.8 Globale Entwurfsentscheidungen

- Um das Laden mehrerer Texturen zu erleichtern, können im Dateiauswahldialog mehrere Dateien gleichzeitig selektiert werden. Diese werden dann nacheinander geladen und sind dann (bei erfolgreichem Ladevorgang) in der Liste verfügbar.
- Es können nur vom User definierte Materialien bearbeitet werden. Standardmaterialien (z.B. Gittermaterial) und die sog. Pluginmaterialien sind nicht sichtbar. Siehe dazu Kapitel Renderer->Textur- und Material-System.

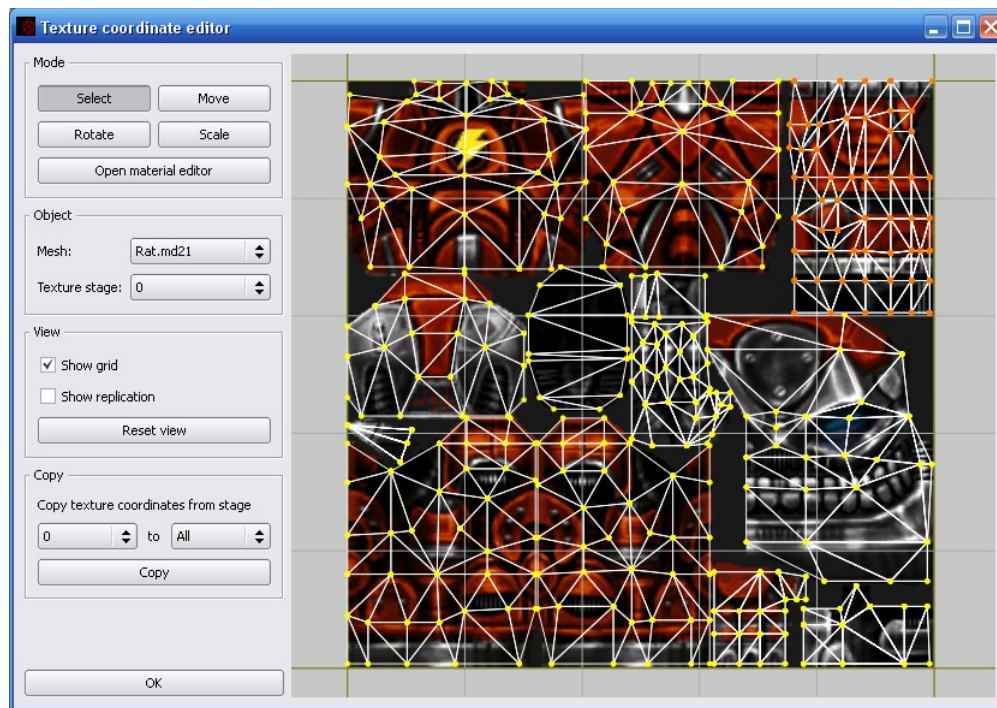
12.3 Texturcoordinate-Editor

Um die Texturkoordinaten eines Models komfortabel bearbeiten zu können, wird dieses Plugin eingesetzt. Es stellt einen Dialog zur Verfügung, mit dessen Hilfe die Koordinaten bewegt, rotiert und skaliert werden können.

12.3.1 UML



12.3.2 Screenshots



Der Texturkoordinaten-Editor

12.3.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es sollen alle Texturschichten eines Meshes bearbeitet werden können
- Zur besseren Übersicht über die Koordinaten soll ein Gitter verwendet werden. Es soll dabei die gleichen Einstellungen besitzen wie das Gitter im 'eigentlichen' Editor
- Optional soll die Texturwiederholung dargestellt werden (Show repetition)
- Die Texturkoordinaten sollen selektiert und dann bewegt, rotiert oder skaliert werden können
- Eine komplette Texturschicht mit allen Texturkoordinaten soll kopiert werden können (z.B. Schicht 1 nach 3)

12.3.4 Mengengerüste

Es wird bei der Darstellung der Texturkoordinaten eines Modells eine Liste gespeichert, die eine Zuordnung zwischen dem Originalvertex (woher die Texturkoordinate stammt) und dem Vertex im Arbeitsbereich speichert. Ansonsten werden keine weiteren Daten gespeichert, die Zuordnungsliste lässt eine direkte Bearbeitung der Koordinaten zu (sowie eine komfortable Art und Weise sie leicht darzustellen).

12.3.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **TextureCoordinateEditorInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **TextureCoordinateEditor:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins wird ein Menüeintrag in das 'Material'-Menü eingefügt, mit dessen Hilfe auf das Texturkoordinaten-Editor-Dialog zugegriffen werden kann.
- **TexCoordEditor:** Diese Klasse stellt das Dialogfenster mithilfe von Qt dar. In diesem Dialog ist außerdem ein Arbeitsbereich vorhanden, indem die Koordinaten per Maus manipuliert werden können (siehe TexCoordWorkArea). Alle Aktionen, die sich auf den Arbeitsbereich auswirken, werden von hier an das Fenster weitergeleitet.

- **TexCoordWorkArea:** Dies ist die Klasse für den Arbeitsbereich, er verwendet den OpenGL-Renderer zur Darstellung. Hier wird immer die aktuell gewählte Texturschicht eines bestimmten Meshes dargestellt. Dabei wird jeder Texturkoordinate einem neuen Vertex zugeordnet (von $(u,v) \rightarrow (x,y,z)$). Bearbeitet werden die Koordinaten über die Maus. Je nach Modus führt die Mausinteraktion zur Selektion, Bewegung, Rotation oder Skalierung der selektierten Daten.
- **MeshItem:** Eine Struktur die einem bestimmten ComboBox-Element (bzw. Index) ein Mesh-Objekt zuordnet.
- **TexCoordVertex:** Eine Struktur zur Zuordnung von Originalvertex im ausgewählten Mesh und dem Vertex das im Arbeitsbereich (siehe TexCoordWorkArea).

12.3.6 Architektursicht – Laufzeit

Falls ein Mesh und Texturschicht ausgewählt wird, werden die Texturkoordinaten der Vertices des Meshes einem Vertex im Arbeitsbereich zugeordnet. Wird der Editor geschlossen, wird dieser Zuordnung das Mesh manipuliert. Pro Vertex des Mesh werden für die Zuordnung je zwei Pointer verwendet.

12.3.7 Einflussfaktoren und Randbedingungen

- Eine Manipulation soll aus Komfortgründen immer gleich sichtbar sein. So befinden sich z.B. im Modus 'Move' die selektierten Vertices immer unter Maus (bis sie losgelassen wird). Die Bearbeitung wird damit vereinfacht, da das Ergebnis sofort sichtbar ist.

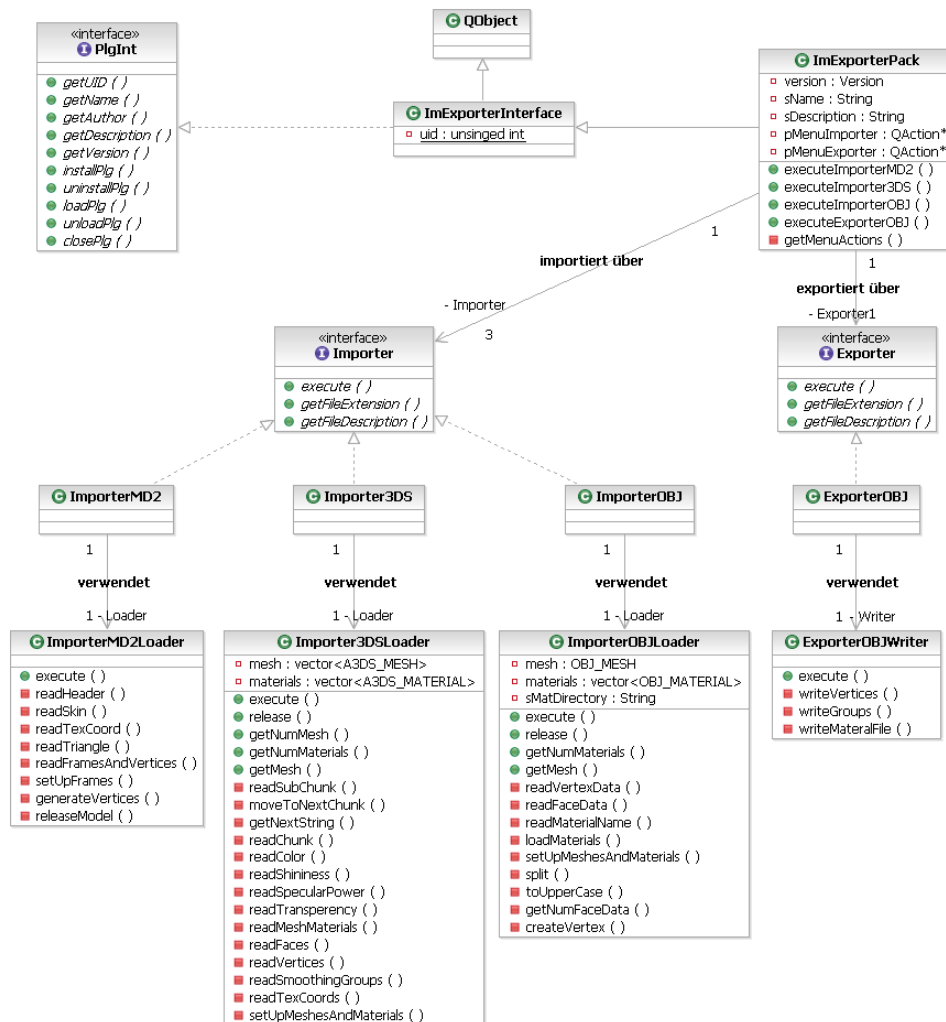
12.3.8 Globale Entwurfsentscheidungen

- Sollte ein Material fehlen, kann dies über einen einfachen Klick auf 'Open material editor' nachgeholt werden. Eine kurzzeitige Bearbeitung des Materials kann nötig sein, wenn die Farbe des Materials und die Linienfarbe annähernd die gleiche Farbe haben und damit sehr schwer voneinander zu unterscheiden sind.

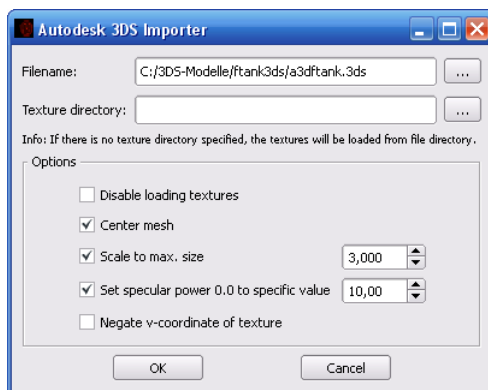
12.4 Im- und Exporter-Pack

Im- und Exporter dienen dazu, Dateiformate von anderen Programmen/Spielen/... zu laden bzw. in dieses Format zu speichern. Einige dieser Im- und Exporter zu den bekanntesten Formaten werden daher in diesem Plugin gesammelt angeboten.

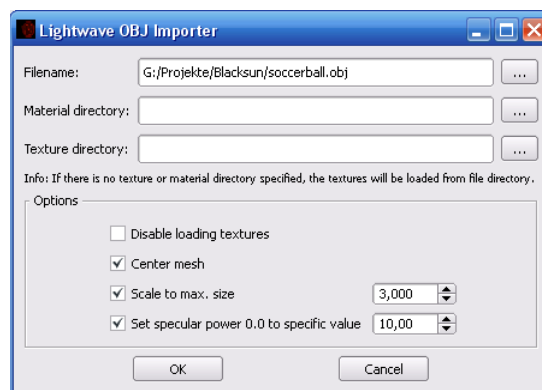
12.4.1 UML



12.4.2 Screenshots



3DS Importer



OBJ Importer

12.4.3 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es sollen die bekanntesten Modellformate geladen werden können. Dies sind 3DS (Autodesk 3D Studio), MD2 (Quake II Model) und OBJ (Lightwave Model)
- Gespeichert werden soll in das OBJ-Format (Lightwave Model)
- Die Importer sollen alle Daten aus der zu ladenden Datei verwenden, bis auf animationspezifische Elemente (z.B. Keyframes) und von BlackSun nicht unterstützte Features (z.B. Splines).
- Alle Exporter sollen alle im Editor gesammelten Daten in das Format speichern, sofern es das Format unterstützt

12.4.4 Mengengerüste

Beim importieren eines Modells wird das komplette Modell geladen. Danach werden die geladenen Daten in ein oder mehrere Meshes überführt, die dann in die Szene eingefügt werden. Im Format gespeicherte Materialien werden ebenfalls in die Szene aufgenommen.

12.4.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- **ImExporterPackInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **ImExporterPack:** Hier wird das Interface des Plugins implementiert. In der Ladephase des Plugins werden alle Im- und Exporter in das 'File'-Menü unter 'Importer' und 'Exporter' eingetragen. Von dort kann auf alle verfügbaren Im- und Exporter zugegriffen werden.
- **Importer:** Interface für alle Importer
- **Exporter:** Interface für alle Exporter
- **ImporterMD2:** Ruft den Loader (siehe ImporterMD2Loader) auf und fügt das vom Loader generierte Mesh in die Szene ein.
- **ImporterMD2Loader:** Die Klasse zum Auslesen aller Daten aus einer MD2-Datei (QuakeII-Model). Zuerst werden komplett alle Daten gelesen und anschließend diese Daten in ein Mesh umgewandelt, die dann in die Szene eingefügt werden können.
- **Importer3DS:** Stellt das Dialog zum importieren der 3DS-Modelle dar. Danach wird der Loader aufgerufen (siehe Importer3DSLoader) und die davon generierten Meshes in die Szene eingefügt.
- **Importer3DSLoader:** Die Klasse zum Auslesen aller Daten aus einer 3DS-Datei (Autodesk 3D Studio). Zuerst werden komplett alle Daten gelesen und anschließend diese Daten in ein bzw. mehrere Meshes umgewandelt, die dann in die Szene eingefügt werden können.
- **ImporterOBJ:** Stellt das Dialog zum importieren der OBJ-Modelle dar. Danach wird der Loader aufgerufen (siehe ImporterOBJLoader) und die davon generierten Meshes in die Szene eingefügt.
- **ImporterOBJLoader:** Die Klasse zum Auslesen aller Daten aus einer OBJ-Datei (Lightwave Model). Zuerst werden komplett alle Daten gelesen und anschließend diese Daten in ein bzw. mehrere Meshes umgewandelt, die dann in die Szene eingefügt werden können.
- **ExporterOBJ:** Diese Klasse bereitet die Daten der Szene und die Materialien auf und stellt das Speicherdialog. Danach wird der Writer(ExporterOBJWriter) gestartet, der die aufbereiteten Daten in das OBJ-Format speichert.
- **ExporterOBJWriter:** Die Klasse zum Schreiben der Daten in eine OBJ-Datei (Lightwave Model). Es speichert zuerst alle Vertexdaten in die OBJ-Datei, danach alle Materialien in die MTL-Datei.

12.4.6 Architektursicht – Laufzeit

Falls ein Model importiert wird, werden kurzzeitig Strukturen angelegt, die zum Auslesen der entsprechenden Datei dienen. Danach werden daraus ein oder mehrere Meshes und Materialien gebildet, die in die Szene eingefügt wird. Der Speicherbedarf ist abhängig von der Größe des zu ladenen Modells.

Beim Exportieren werden auch kurzzeitig Strukturen angelegt, um das Speichern zu erleichtern. Diese werden nach dem Speichervorgang wieder gelöscht.

12.4.7 Einflussfaktoren und Randbedingungen

- Falls das Modellformat, das geladen werden soll, Animationen unterstützt, soll die Standardanimation der Datei genommen und in ein statisches Modell umgewandelt werden.
- Beim Importieren eines MD2-Modells wird keine Textur geladen. Diese soll nachträglich eingetragen werden, z.B. im Material-Editor. Der Grund ist, dass in einer MD2-Datei kein Texturname gespeichert ist.

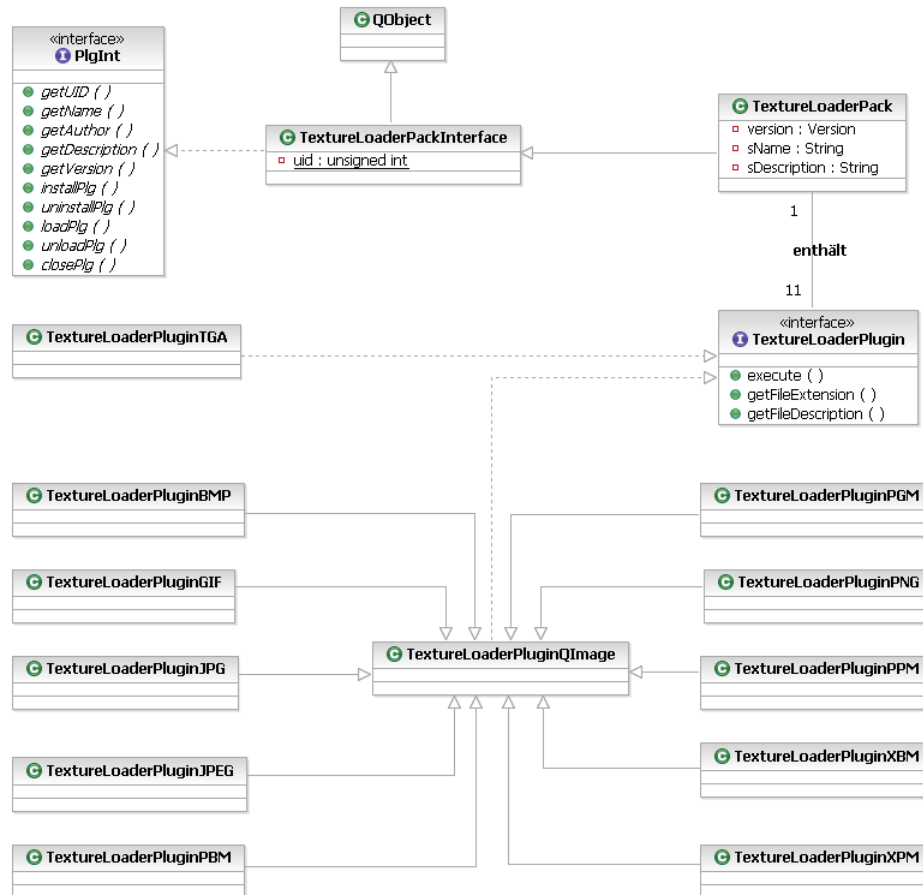
12.4.8 Globale Entwurfsentscheidungen

- Da viele Objekte in einem anderen Größenverhältnis erstellt wurden, enthalten die Import-Dialoge eine Option zum skalieren des geladenen Modells auf eine bestimmte Größe.
- Texturen und Materialdateien können in einem anderen Ordner liegen als das zu ladenende Modell. Um diese trotzdem laden zu können, kann der Ordner ausgewählt werden, wo sich diese befinden.

12.5 TextureLoaderPack

Um Texturen von bestimmten Dateiformaten laden zu können, wird dieses Plugin eingesetzt. Es enthält eine Reihe von TextureLoaderPlugin-Klassen, die es dem TexturManager des Renderers ermöglichen, Texturen zu laden.

12.5.1 UML



12.5.2 Funktionale und nicht funktionale Anforderungen

Das Plugin muss folgende Anforderungen erfüllen:

- Es sollen die bekanntesten Grafikformate geladen werden können. Darunter zählen: BMP, TGA, PNG, JPG, JPEG und GIF
- Falls das Laden einer Textur fehlschlägt, soll eine Meldung in der Log hinterlegt werden. Bei erfolgreichem Laden soll ebenfalls eine Nachricht eingetragen werden
- Die TextureLoaderPlugins sollen die wichtigsten Daten (siehe `Renderer::TextureInfo`) korrekt ablegen
- Sollte ein Grafikformat Transparenz oder Kompression unterstützen, sollte dies auch korrekt geladen werden
- Die Bilddaten sollen in ein unkomprimiertem Byte-Array gespeichert werden

12.5.3 Mengengerüste

Die Textur wird in einen übergebenem Byte-Array gespeichert, das dort allokiert und befüllt wird. Ein Pixel in der Bilddatei hat im Normalfall 3 Byte, im Falle von zusätzlich gespeichertem Alphawert 4 Byte. Eine Textur mit der Dimension $Width \times Height$ hat daher folgenden Speicherplatzbedarf:

Ohne Transparenz: $3 * Width * Height$ Byte

Mit Transparenz: $4 * Width * Height$ Byte

12.5.4 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

- ***TextureLoaderPackInterface***: Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- ***TextureLoaderPack***: Hier wird das Interface des Plugins implementiert. In der Ladephase werden alle TextureLoaderPlugins in den TextureManager hinzugefügt. Ab dann ist der TextureManager bereit, diese zu Laden. Sollte bereits ein TextureLoaderPlugin vorhanden sein, der den gleichen Dateityp lädt, so wird dieser nicht eingefügt. Wird das Plugin entladen, werden nur die neu hinzugefügten TextureLoaderPlugins entfernt.
- ***TextureLoaderPluginQImage***: Diese Klasse lädt die meisten der Bildformate mit der QT-Klasse 'QImage'. Dies hat den Vorteil, das die Laderoutinen dort plattformunabhängig und sehr performant sind.
- ***TextureLoaderPluginBMP***: Lädt 'Windows Bitmap'-Dateien mithilfe von QImage
- ***TextureLoaderPluginGIF***: Lädt 'Graphic Interchange Format'-Dateien mithilfe von QImage
- ***TextureLoaderPluginJPEG***: Lädt 'Joint Photographic Experts Group'-Dateien mithilfe von QImage
- ***TextureLoaderPluginJPG***: Lädt 'Joint Photographic Experts Group'-Dateien mithilfe von QImage
- ***TextureLoaderPluginPBM***: Lädt 'Portable Bitmap'-Dateien mithilfe von QImage
- ***TextureLoaderPluginPGM***: Lädt 'Portable Graymap'-Dateien mithilfe von QImage
- ***TextureLoaderPluginPNG***: Lädt 'Portable Network Graphics'-Dateien mithilfe von QImage
- ***TextureLoaderPluginPPM***: Lädt 'Portable Pixmap'-Dateien mithilfe von QImage
- ***TextureLoaderPluginTGA***: Lädt 'Targa Image'-Dateien
- ***TextureLoaderPluginXBM***: Lädt 'X11 Bitmap'-Dateien mithilfe von QImage
- ***TextureLoaderPluginXPM***: Lädt 'X11 Pixmap'-Dateien mithilfe von QImage

12.5.5 Architektursicht – Laufzeit

Die TextureLoaderPlugins werden aufgerufen, wenn der User eine Bilddatei lädt (beispielsweise im MaterialEditor). Sollte die geeignete Laderoutine vorhanden sind, wird versucht, die Datei zu laden. Sollte keine Laderoutine gefunden worden sein, wird eine Meldung in der Log eingetragen, die auf das Fehlen hinweist. Siehe dazu nähere Informationen im Kapitel zum Renderer->Textur- und Material-System.

12.5.6 Einflussfaktoren und Randbedingungen

- Für die meisten Laderoutinen wurde QImage verwendet, da es sehr performant und vielseitig einsetzbar ist. Außerdem unterstützt er viele Formate, die sonst sehr aufwändig zu implementieren wären (z.B. JPG/JPEG, ...).
- Da TGA ebenfalls ein wichtiges Bildformat ist, von QImage allerdings nicht geladen werden kann, wurde diese Laderoutine 'von Hand' implementiert. Es können sowohl komprimierte und unkomprimierte Bilder mit und ohne Transparenz geladen werden. Aufgrund der verschiedensten TGA-Typen können aber nicht alle Dateien abgedeckt und somit nicht geladen werden.

12.5.7 Globale Entwurfsentscheidungen

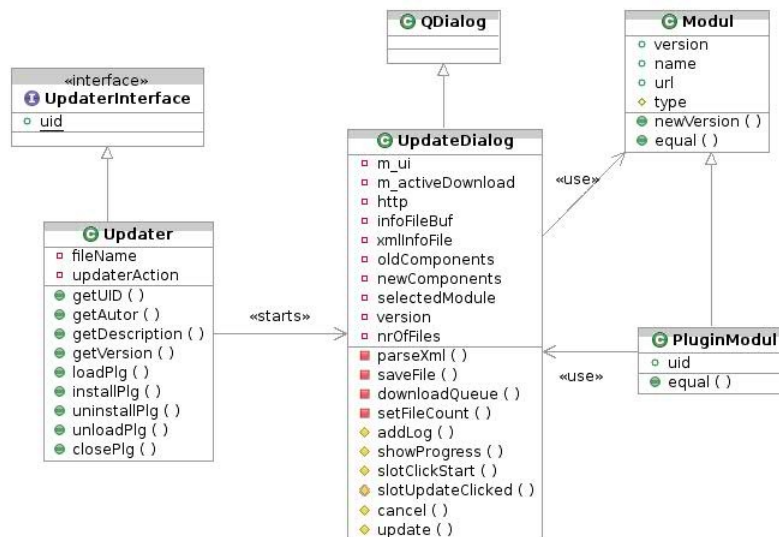
- Um dem User ein Feedback zu geben, ob das Laden erfolgreich war, werden Meldungen in die Log eingetragen. Diese geben auch bei einem Fehlschlagen einen Hinweis auf eine mögliche Fehlerquelle.

12.6 Updater

Der Updater besteht aus einem Plugin, das die im Editor vorhandenen Informationen (lokale Version, installierte Plugins, ...) sammelt und diese an die UpdaterApp, eine eigenständige Anwendung, weitergibt. Diese überprüft dann, ob ein Update nötig ist und lädt dieses, falls gewünscht, herunter und aktualisiert den Editor.

Der Updater ist als **Experimentell** eingestuft, da noch nicht ausreichend getestet werden konnte, wie die Verflechtung der Module mit einbezogen werden muss.

12.6.1 UML



12.6.2 Funktionale und nicht funktionale Anforderungen

Der Updater muss folgende Anforderungen erfüllen:

- Es muss von jeder Datei, die aktualisiert wird, ein Backup angelegt werden
- Der Benutzer braucht nicht die installierten Komponenten und ihre Versionen kennen, der Updater sucht sich die benötigten Daten selber
- Der Benutzer braucht nicht die Adressen der Updates kennen. Die Updates müssen auch nicht auf dem selben Server liegen.

12.6.3 Mengengerüste

Das Plugin kommt bis auf den Pfad zu der UpdaterApp ganz ohne Attribute aus. Die UpdaterApp hingegen speichert für jedes installiertes Modul ein Modul-Objekt, für Plugins ein PluginModul-Objekt. Des weiteren wird für jedes weitere, in der Update – Datei gefundene Modul, ein Modul oder ein PluginModul angelegt. Daneben besitzt die UpdaterApp noch Objekte um das Anwendungsfenster darzustellen, für den Dateidownload und ein XML-Objekt für die Verarbeitung der Update-Datei.

12.6.4 Architektursicht – Gesamtsicht

UpdateApp ist eine Eigenständige Anwendung. Es muss weder vom Editor aufgerufen werden, noch greift es auf Daten des Editors zu. Alle erforderlichen Daten werden beim Start über die Kommandozeile übergeben. Das Plugin Update hingegen, das diese Daten sammelt und dann damit die UpdateApp aufruft, ruft die Kernkomponentendaten von den jeweiligen Komponenten und die Plugindaten vom Pluginsystem ab.

12.6.5 Architektursicht – Klassensicht

Folgende Klassen/Interfaces werden verwendet:

Plugin Updater:

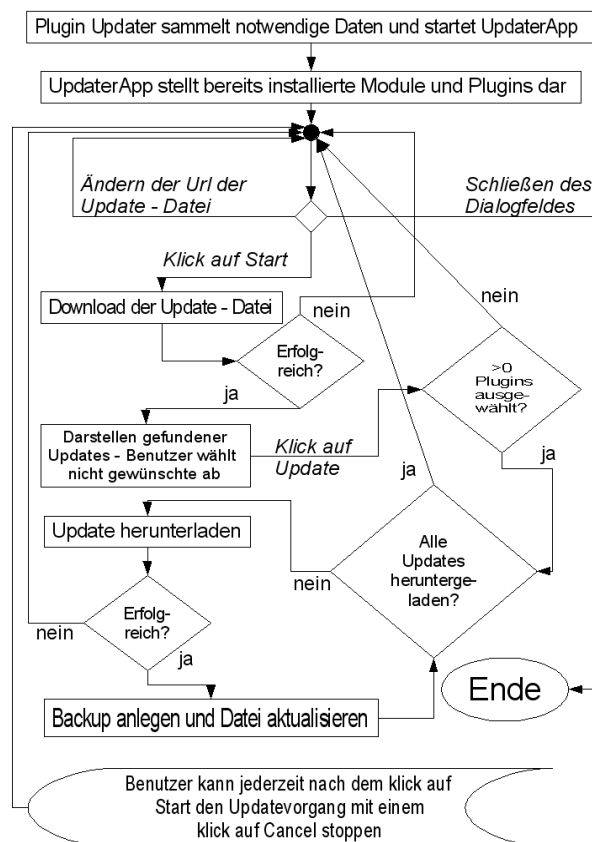
- **UpdaterInterface:** Dies ist das Interface für das Plugin. Es vereint die UID (die eindeutige Identifikationsnummer), das Plugin-Interface des Pluginsystems und die Eigenschaften von QObject. QObject wird benötigt, um die Signale und Slots verwenden zu können, die von Qt angeboten werden und die Handhabung der GUI (insbesondere die Änderung von Widget-Eigenschaften und deren Folge in der GUI) stark erleichtern. Damit bleibt auch die GUI vom diesem konkreten Plugin unabhängig.
- **Updater:** Diese Klasse bildet die Schnittstelle zwischen dem Editor und der eigentlichen UpdateAnwendung. Beim laden des Plugins stellt es diese unter dem Menü 'Help->Check for new version' für den Benutzer zur Verfügung. Wird dieser Menüpunkt dann ausgewählt, dann sammelt Update alle erforderlichen Daten und startet die Update – Anwendung. Des weiteren implementiert diese Klasse das Interface UpdateInterface.

Anwendung UpdateApp:

- **UpdateDialog:** Diese Klasse repräsentiert das Anwendungsfenster, das nach dem Start dem Benutzer gezeigt wird. Außerdem beinhaltet sie die Objekte für den Dateidownload, das Parsen der Update-Datei und das Backup der aktualisierten Dateien.
- **Modul:** In dieser Klasse werden alle für den Updatevorgang nötigen Daten für eine Kernkomponenten gespeichert (Name, Url, Version). Außerdem besitzt sie eine Funktion, um zwei Module miteinander zu vergleichen und eine, die feststellt, ob ein Modul eine neuere Version hat als ein anderes.
- **PluginModul:** Diese Klasse ist abgeleitet von Modul. Sie repräsentiert Plugins und hat als zusätzliches Attribut noch die UID.

12.6.6 Architektursicht – Laufzeit

Das folgende Diagramm beschreibt die Arbeitsweise des Updaters zur Laufzeit:



12.6.7 Einflussfaktoren und Randbedingungen

- Es wurde das HTML-Protokoll zum Download der Dateien gewählt, da dieses einfacher zum implementieren ist und auch mehr an HTML-Server zur Verfügung stehen. So liegen zur Zeit die Dateien auf unserem Homeverzeichnis der FH.
- Die Daten in der Update-Datei werden im XML-Format gespeichert. Dieses wurde gewählt, weil es die Inhalte übersichtlich darstellt und plattformunabhängig ist. Entscheidend war zudem, das im QT-Framework bereits XML-Module integriert sind.

12.6.8 Globale Entwurfsentscheidungen

- Der eigentliche Updater wurde als eigenständige Anwendung realisiert, da dieser eventuell auf die Programmdateien des Editors zugreifen muss und diese ändert.

13 Ergänzungen

13.1 Das Makefile-Konzept

Das hier angewandte Konzept ist so gestaltet, das es ein einziges, zentrales Makefile im Projektverzeichnis gibt, von dem aus alle anderen Makefiles der Komponenten aufgerufen werden. Die Makefiles der Komponenten werden nicht selber geschrieben, sondern von dem Tool *qmake* erstellt. Diese Weg wurde deshalb gewählt, da alle Komponenten, die die QT – GUI benutzen, dies sowieso machen müssen (da die Quelldateien zuvor noch mit dem so genannten *moc* – Compiler verarbeitet werden müssen), und sich somit ein einheitlicheres Bild zeigt, aber auch, da die *qmake* Projektdateien verständlicher sind als die Makefiles. So ruft das zentrale Makefile zuerst bei einer Komponente *qmake* auf um aus der Projektdatei ein Makefile zu erstellen, dann *make* das dann das Makefile abarbeitet und die Komponente kompiliert.

Das Makefile ist nicht ganz Plattformunabhängig. Mit *make debugwin, debuglinux, releasewin, releaselinux, cleanwin* oder *cleanlinux* wird die gewünschte Aktion ausgewählt. Die 'Targets' legen dann die jeweiligen Plattformspezifischen Parameter (z.b. der Verzeichnisname oder die Endung der Shared Library) fest. Somit kann der größte Teil der Datei Plattformunabhängig gestaltet werden.

13.2 Modul-Konzept

Jede Komponente wird separat in eine eigene dynamische Bibliothek kompiliert (.so / .dll). Diese werden dann beim Start des Programmes automatisch geladen. Dies hat die Vorteile:

- das wenn eine Komponente verändert wurde, nur diese eine neu kompiliert werden muss (außer man ändert was an den Schnittstellen).
- das das ausführbare Programm kleiner ist.

Nachteile:

- Dynamische Bibliotheken werden in jedem Betriebssystem anders behandelt. So benötigt das Programm z.B. unter Linux einen Startscript, der den Pfad der Bibliotheken dem Betriebssystem bekannt gibt.
- Wurde nach einem Ändern der Schnittstellen nicht jedes Modul neu kompiliert, so konnte dies zu unverständlichen Programmabstürzen führen.
- Des Debuggen war Anfangs unmöglich, dieses Problem konnte aber inzwischen gelöst werden.

13.3 Singletons

Alle Kernkomponenten zeichnen sich durch das Singleton – Konzept aus. Die einzige Instanz dieser Klasse wird mit *getInstance* erzeugt und auch darauf zugegriffen. In der main – Funktion werden zu Beginn eine Instanz der Klasse *Mainwindow* erzeugt. Diese ruft ihrerseits wieder Funktionen anderer Module auf und erzeugt sie somit. Durch dieses Konzept wird sichergestellt, das es im gesamten Programm nur eine einzige Instanz dieser Klasse gibt. Der Aufruf des Destruktors wird über eine sogenannte Wächterklasse sichergestellt. Beispielcode:

Komponente.h:

```
class Komponente
{
//...
private:
    static Komponente* m_instance;
//...
    Komponente();
    virtual ~Komponente();

    class Waechter
    {
    public:
        virtual ~Waechter()
        {
            if(Komponente::m_instance != NULL)
                delete Komponente::m_instance;
        }
    };
    friend class Waechter;
};
```

Komponente.cpp:

```
Komponente* Komponente::m_instance = NULL;

void Komponente::createInstance()
{
    static Waechter g;
    if(m_instance == NULL)
        m_instance = new Komponente;
}

Komponente* Komponente::getInstance()
{
    return m_instance;
}
```

Nachteile: Die Wächterklasse wird als statisches Objekt angelegt. Da die statische Objekte beim beenden des Programms in zufälliger Reihenfolge gelöscht werden, werden die Destruktoren der Wächterklassen, die ja das Singleton löschen, in unvorhersehbarer Reihenfolge gelöscht. Somit kann es zu Problemen kommen, wenn ein Modul im Destruktor auf ein anderes Modul zugreift, das evtl. schon gelöscht ist. Am anfälligsten dafür sind die Plugins, weil wenn diese z.B. in *close()* oder in ihrem Destruktor auf ein Modul zugreifen das schon gelöscht wurde, dann kann dies zu Programmfehlern führen.

13.4 Namespace-Konzept

Zu beginn stellte sich die Frage, wie wir 'unsere' Klassen von anderen abgrenzen um Namenskonflikte zu vermeiden. Dazu gibt es zwei Techniken: Entweder das verwenden von Prefixe (z.B. QT, da beginnen alle Klassennamen mit Q. Analog dazu hätten unsere z.B. mit BS beginnen können) oder das verwenden von von Namensräumen (z.B. die C++ Standard Library mit ihren Namensraum std.) Wir entschieden uns für die Namensräume, da sie flexibler sind und einzelne Module besser voneinander abgegrenzt werden (jedes Modul hat seinen eigenen Namensraum).

Nachteil: Das zugreifen auf Objekte innerhalb des Namensraums gestaltet sich ziemlich aufwendig (vor allem in den Plugins). Abhilfe schafft hier die using – Deklaration. Allerdings gibt man damit auch die Vorteile eines Namensraums auf. Probleme hatten wir hier mit der Klasse Polygon unter Verwendung der using - Deklaration. Das bereitete unter Windows Probleme. Daran könnte eventuell die Windows – API – Funktion *int Polygon(HDD__*,tagPOINT const*,int)* schuld sein.