Softwarearchitektur

Programmieren eines 3D-Editors

-- Blacksun --

Mitglieder:	Philipp Gruber	pgruber@fh-landshut.de
	Reinhard Jeschull	rjeschu@fh-landshut.de
	Thomas Kuhndörfer	tkuhndo@fh-landshut.de
	Thomas Tischler	ttischl1@fh-landshut.de
	Stefan Zeltner	szeltne@fh-landshut.de
Betreuer:	Prof. Peter Hartmann	
Homepage:	http://sourceforge.net/projects/blacksun/	
letzte Änderung:	Mittwoch, 10. Januar 2007	

Inhaltsverzeichnis

1 Übersicht

1.1 Funktionale und nicht funktionale Anforderungen

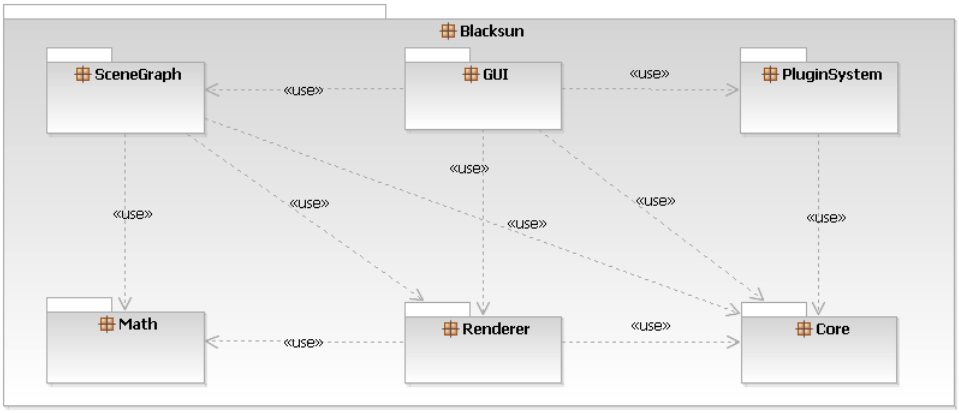
Alle funktionalen Anforderungen sind im Pflichten-/Lastenheft aufgelistet.

1.2 Aufteilung der bisherigen Arbeit

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

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 komforabel zur Verfügung zu stellen.
- ***Renderer:*** Die Vermittlungsstelle zwischen OpenGl und den anderen Blacksun-Modulen. Die Hauptaufgabe ist es die Renderaufträge an OpenGl weiter zu leiten und wichtige Daten wie Renderaufträge, Texturen und Materialien zu verwalten.
- ***PluginSystem:*** Verantwortlich für die Installation, Verwaltung und Kommunikation der Plugins.
- ***SceneGraph:*** Verantwaltung 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 individuel 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.

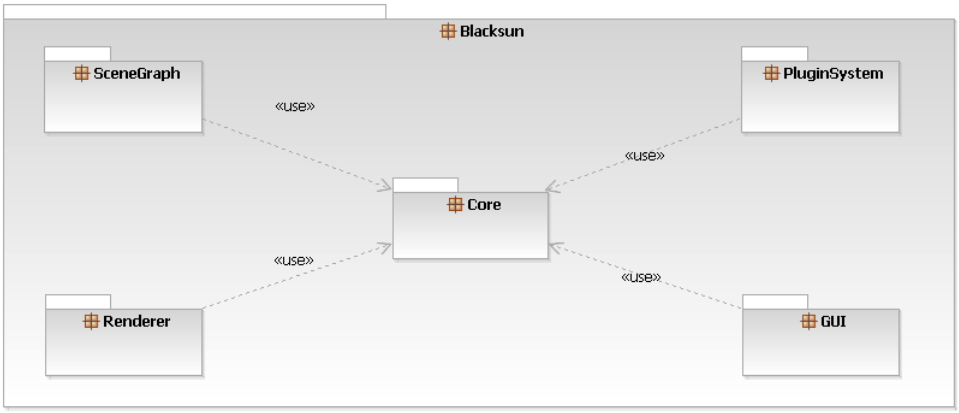
2.3 Mengengerüste

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 includiert.

Das folgende Paketdiagramm zeigt die Integration des Core-Moduls:



2.5 Architektursicht – Klassensicht

Die Log-Klasse entält 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“-Funtionen 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__)

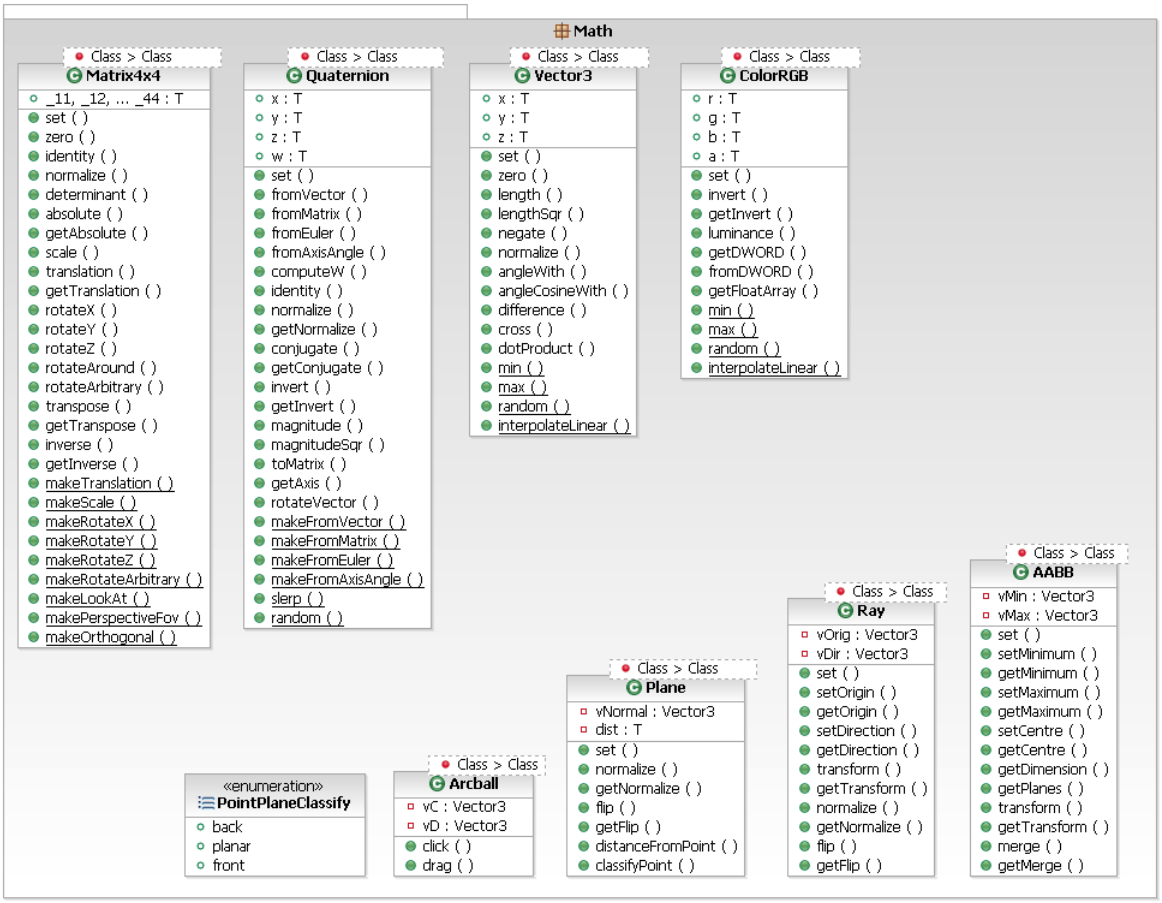
2.9 Globale Entwurfsentscheidungen

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 enhält 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.

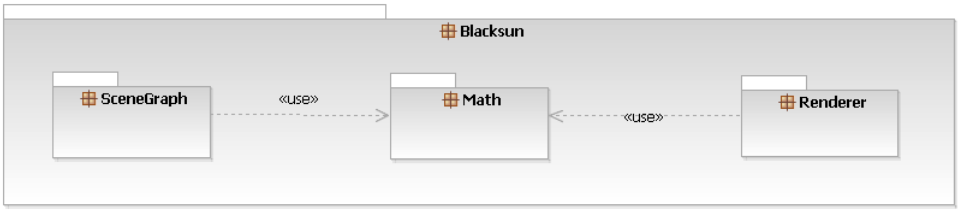
3.3 Mengengerüste

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, includiert 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 (Vector*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 werde kann. Wie die Matrix-Klasse kann auch diese Klasse als direkter Eingabeparameter für OpenGl-Funktionen genutzt werden.
- ***ColorRGB***: Farbwert im RGB-Raum. Enhält 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 euklidsche 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 beispielweise verwendet, um nicht sichtbare Objekte vom Rendern auszuschließen.
- ***Arcball***: Wandelt eine Bewegung der Maus in eine Rotations-Quaternion um.

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>

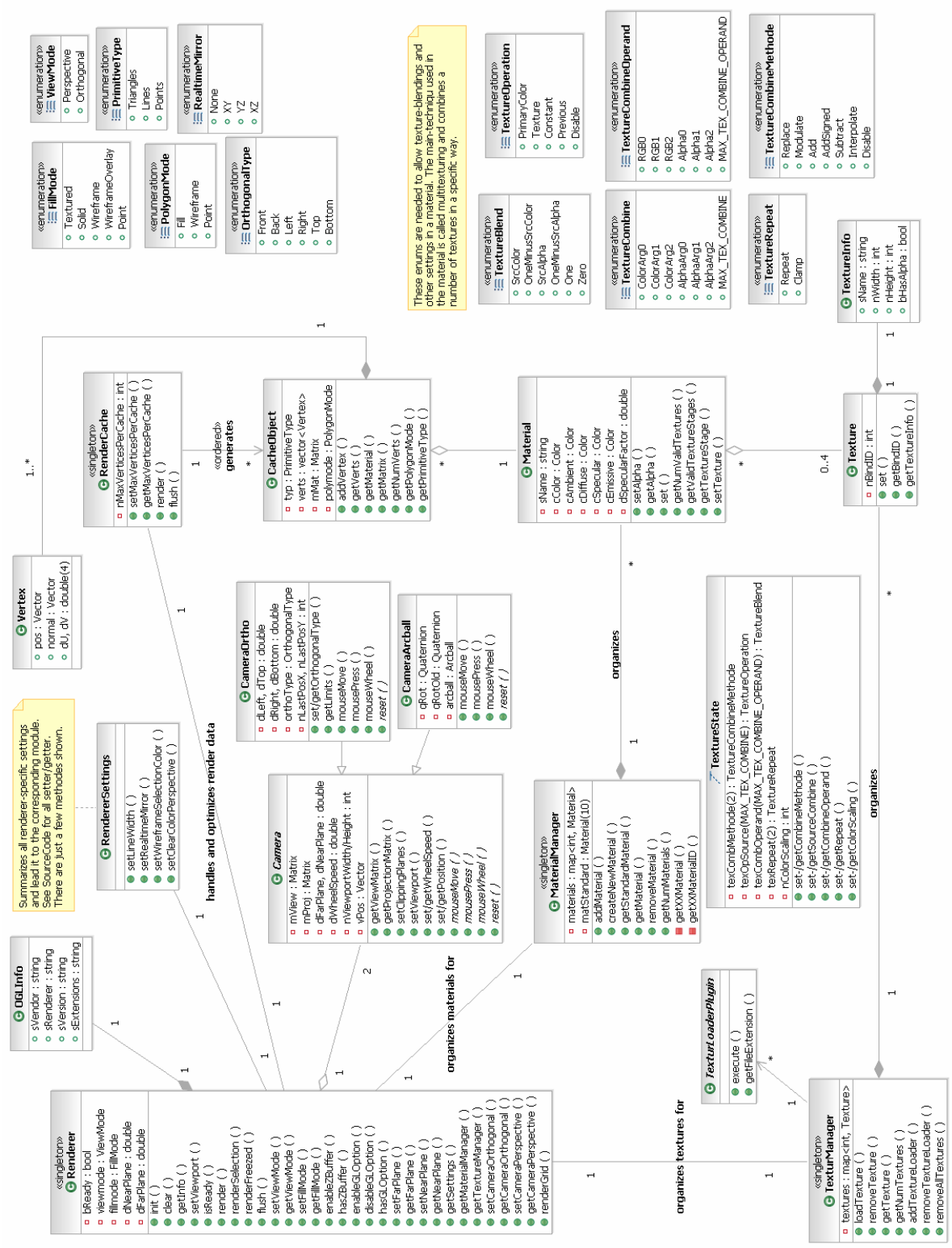
3.9 Globale Entwurfsentscheidungen

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 Renderer

Das Renderermodul ist die Vermittlungsstelle zwischen OpenGl und den anderen Blacksun-Modulen. Die Hauptaufgabe ist es die Renderaufträge an OpenGl weiter zu leiten und wichtige Daten wie Renderaufträge, Texturen und Materialien zu verwalten.

4.1 UML



4.2 Funktionale und nicht funktionale Anforderungen

Der Renderer muss folgende Anforderungen erfüllen:

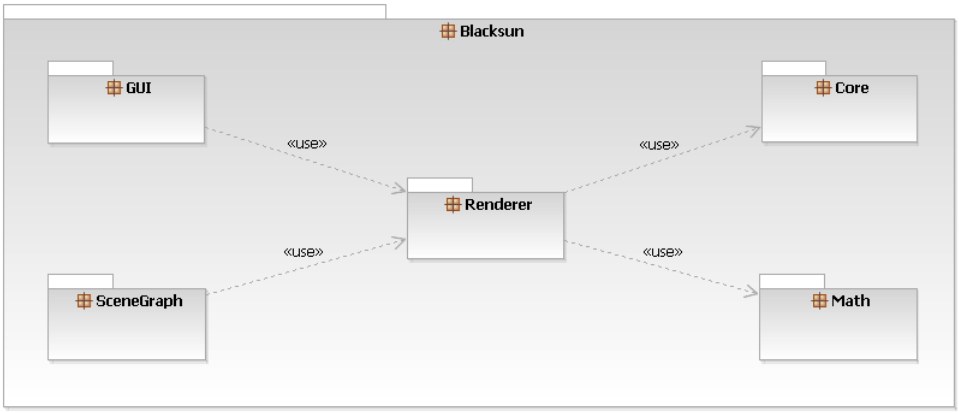
- Es müssen Renderaufträge angenommen werden können. Diese können Triangles (Dreiecke), Linien und Punkte sein.
- Um die Performance beim Rendern zu steigern, werden die Renderaufträge zu möglichst großen Paketen zusammengefasst. Der Grund ist, das das Senden der 3D-Daten an die Grafikkarte über OpenGl immer über den Grafikbus 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. Außerdem soll der Renderer nicht sichtbare Szenenobjekte gar nicht erst zur Grafikkarte senden (das sogenannte Culling).
- 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 Material erstellen, löschen und ändern können. Ein Material beschreibt die Oberflächeneigenschaften und vereint materialspezifische Eigenschaften (z.B. Farben), eine Menge von Texturen und die dazugehörigen Kombinationeinstellungen derjenigen Texturen.
- Eine konfortable 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, das 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 all diese Daten zur Grafikkarte geschickt.
- Je nach Szene und Einstellungen werdem im Worst-Case alle Rohdaten bis zum entgültigen 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:



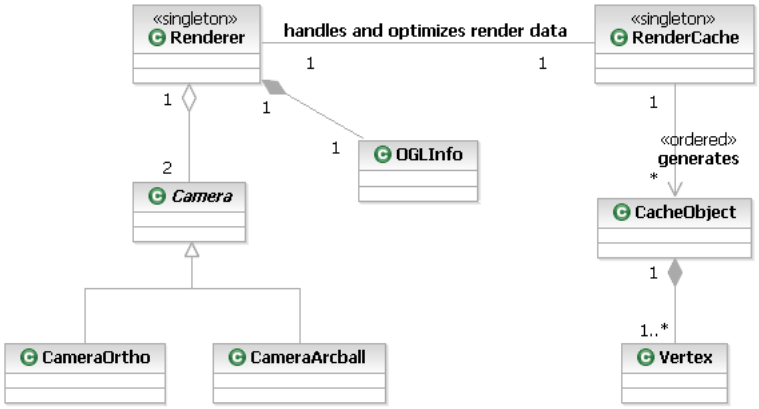
4.5 Architektursicht – Klassensicht

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.

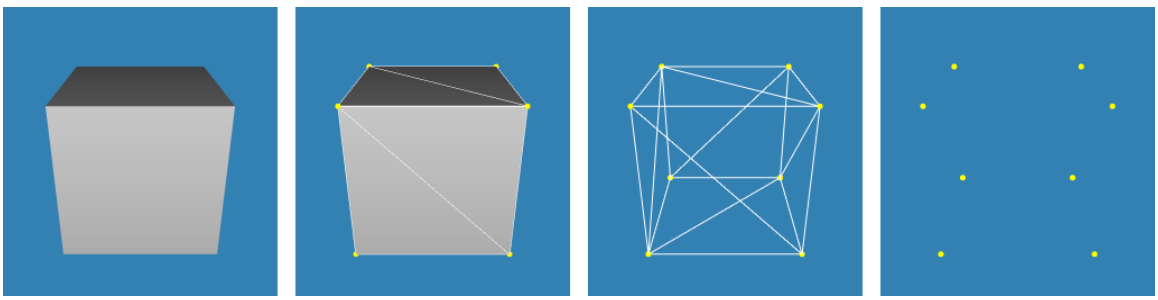
Diese drei Untersystme sind im Folgenden im Detail beschrieben:

4.5.1 Eigentliches Renderer-System



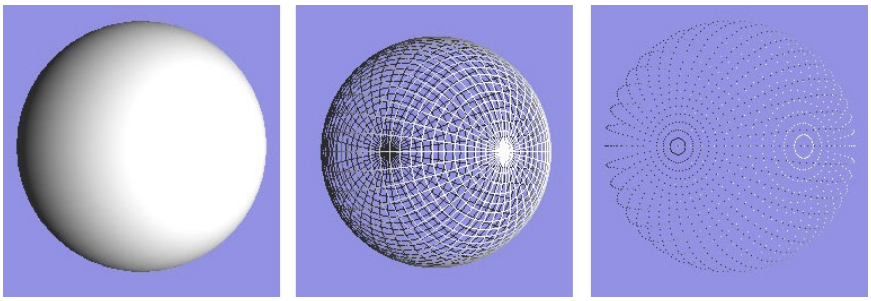
Folgende Klassen /Strukturen sind beteiligt:

- ***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 TexutreManager, MaterialManager und die zentrale Konfigurations-Klasse des Renderer-Moduls.



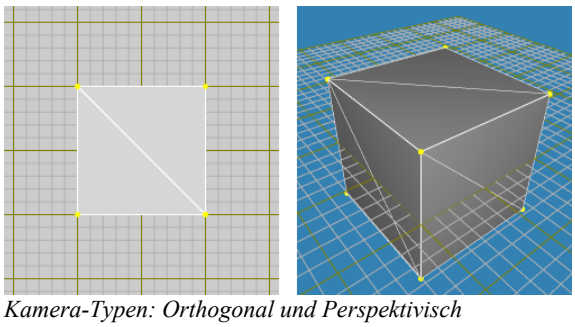
Füllmodi am Beispiel einer Box: Solid, WireframeOverlay, Wireframe, Point

- ***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.
- ***RendererCache:*** Ist zuständig für das Verarbeiten der Renderaufträge 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 entgültigen 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.
- ***CacheObject:*** Bildet ein Paket des RendererCache 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 Primtive-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.



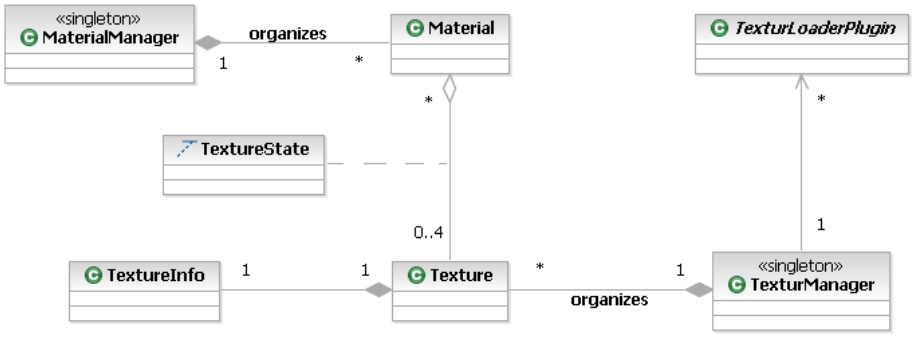
Polygon-Typ am Beispiel einer Kugel: Filled, Wireframe, Point

- ***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-Bildschirmebene 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 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-Sytem



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 Dublikate 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 Eingeschaften 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 berets 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 Laderountine, 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 TexturManager eine Methode zum Hinzufügen neuer Laderoutinen. Eine solche Laderoutine lädt alle Dateien einer bestimmten Dateiendung. Im TexturManager 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 TexturManager 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 sich die Kombinationsart (z.B. Modulate, Add, ...), die Quellen für die Farbargumente 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 Texturkoordinate)
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).

4.8 Einflussfaktoren und Randbedingungen

- Alle wichtigen Manager-Klasse und der Renderer sind Singletons, da nur eine Instanz bestehen darf.
- Die Initialisierung des Renderers muss separat erfolgen. Über den Konstruktor kann nicht garantiert werden, dass ein Fenster vorhanden ist, auf dem OpenGl intialisieren ist (z.B. OpenGl-Widget).
- Die maximale Anzahl von Texturen pro Material und Vertex ist auf 4 limitiert. Ein höheres Limit würde bei großen Szenen zu viel Speicherplatz benötigen.
- Zusätzlich zu den vom Benutzer definierten Materialien werden im Material-Manger noch Standardmaterialien gespeichert, wie das Material für Linien oder selektierte Objekte. Dies hat den Vorteil, das auch die Standardmaterialien über eine eindeutige ID identifiziert werden können. Allerdings können diese Material nicht wie die benutzerdefinierten Materialien gelöscht werden.

4.9 Globale Entwurfsentscheidungen

- Der Renderer enthält viele Parameter die eingestellt werden können, wie z.B. Standardmaterialien. Um den anderen Modulen eine zentrale Stelle für alle Einstellungen zu bietet, enthält der Renderer eine Klasse RendererSettings, die über das Renderer-Singleton zugegriffen werden kann. Sie leitet die Einstellungen an die richtigen Klassen weiter.
- Alle wichtigen Operationen und Instanzen werden im Renderer-Singleton bereitgestellt. Das interne Singleton RenderCache ist aber nicht über das Renderer-Singleton erreichbar, da der Benutzer des Renderer-Moduls nicht drauf zugreifen sollte. Der Grund ist, das der User bei einem Missbrauch des RenderCache-Singletons die Performance verschlechtern würde/könnte.

5 Pluginsystem

Dieses Modul ist verantwortlich für die Installation, Verwaltung und Kommunikation der Plugins.

5.1 UML

5.2 Funktionale und nicht funktionale Anforderungen

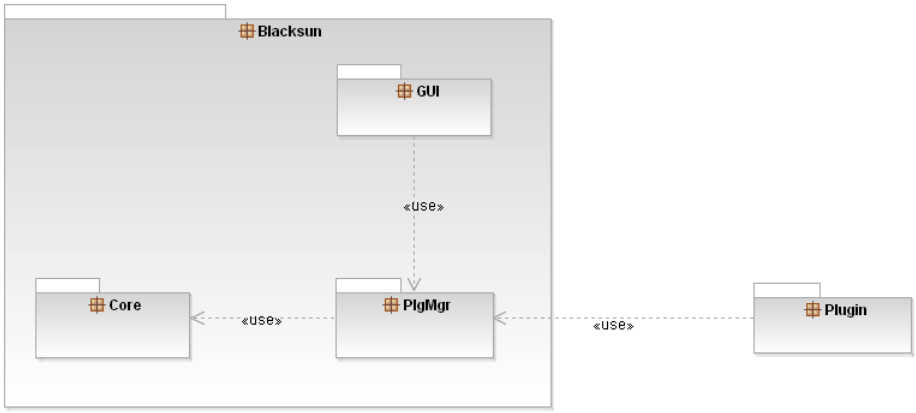
- Plugins bringen nachträglich Funktionen in das Programm ein, ohne das dieses neu kompiliert werden muss.
- Plugins sind dynamische Bibliotheken (.so unter Linux, .dll unter Windows).
- Sie können auf die Funktionalität von anderen Plugins zugreifen.
- Das Pluginsystem kann neue Plugins installieren und auch wieder deinstallieren.
- Desweiteren 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.

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 und einer auf die Bibliotheken, der Status 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:

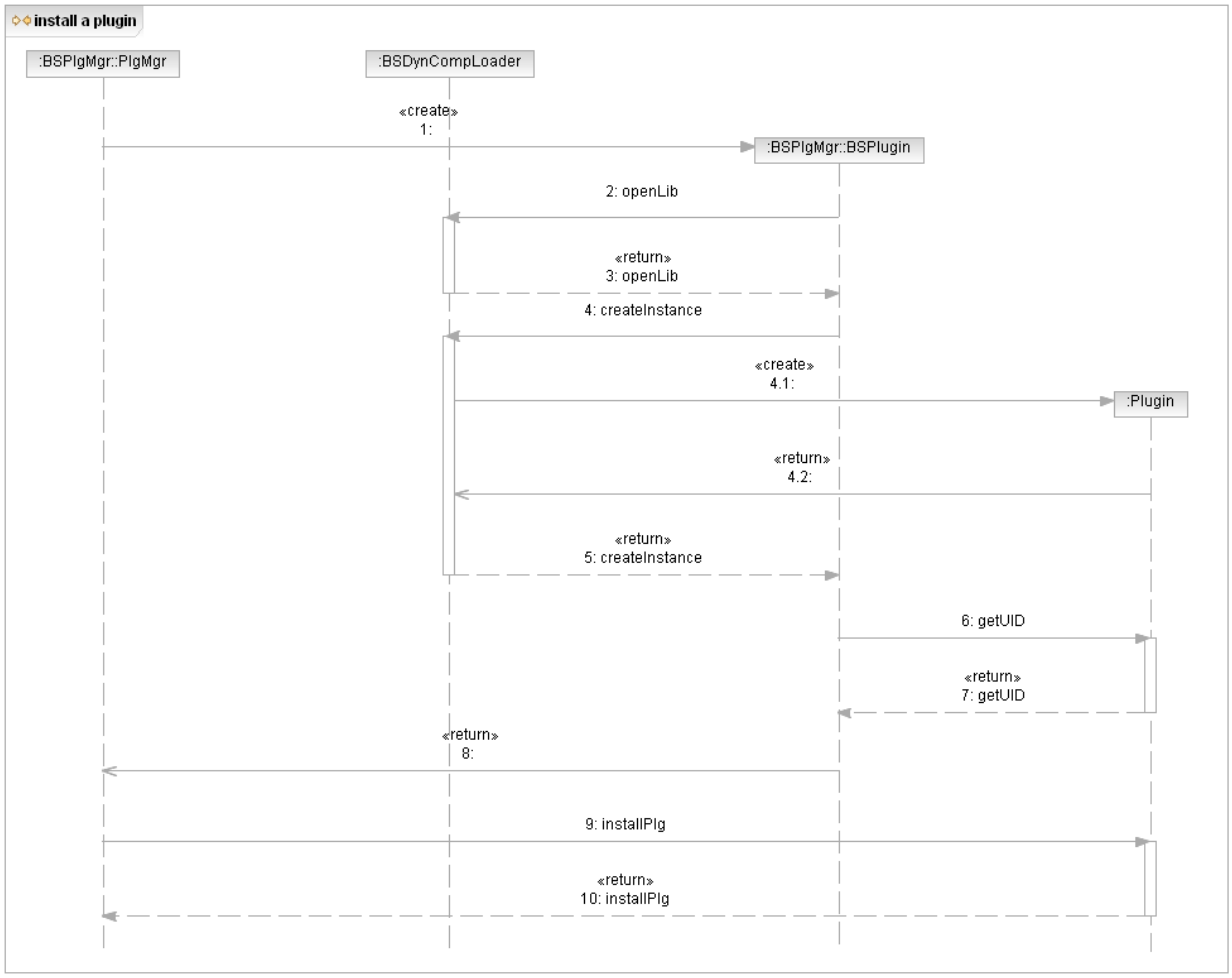
- ***PlgMgr:*** Diese Klasse ist für die Verwaltung der Plugins zuständig. Für jedes installierte Plugin wird ein - das jeweilige Plugin repräsentierende - *BSPlugin* gespeichert. Alle Plugins können über ihre eindeutige UID (siehe 8.1) und ihre Version bestimmt werden. Zudem ist diese Klasse die Schnittstelle 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 eine Bibliothek. Da es in jedem Betriebssystem unterschiedliche Funktionen gibt, prüft der Prekompiler über `#ifdef WIN32` bzw. `#ifndef WIN32` das jeweilige Betriebssystem und setzt dann die jeweils richtigen Funktionen ein. Übersicht:

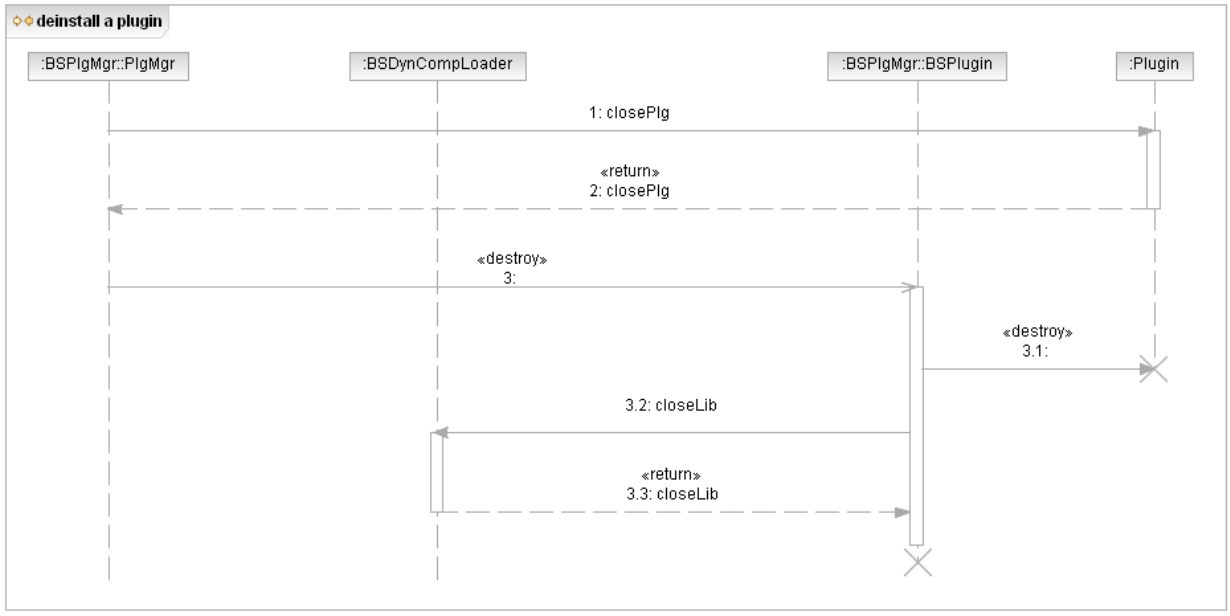
	Windows	**Linux**
	`#include <windows.h>`	`#include <dlfcn.h>`
`#define DCL_OPEN_LIB(a)`	`LoadLibrary((const WCHAR*) a)`	`dlopen(a, RTLD_LAZY)`
`#define DCL_GET_OBJECT`	`GetProcAddress`	`dlsym`
`#define DCL_CLOSE_LIB`	`FreeLibrary`	`dlclose`
`#define DCL_INSTANCE`	`HINSTANCE`	`void*`

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, das die Funktion extern "C" PlgInt* createInstance() in der Bibliothek nicht gefunden wurde.

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(UID) sind Getter und liefern die gewünschten Informationen über das mit der UID 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 9.3)

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, das viele Funktionen mit Plugins realisiert werden sollte, war es klar, das es eine flexible Kommunikation zwischen zwei Plugins geben muss. Am besten währe 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 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, das das andere Plugin noch nicht entladen/deinstalliert wurde (s.o.). Das heißt, Plugins müssen eindeutig sein. Das beinhaltet aber auch, da Plugins ja auch weiterentwickelt werden, das es eine Art Versionskontrolle geben muss.
  Ansatz: *UID (unique ID):*
  - Vorgehen: Jedes Plugin hat eine ein eindeutige UID und eine Version (siehe 8.1)

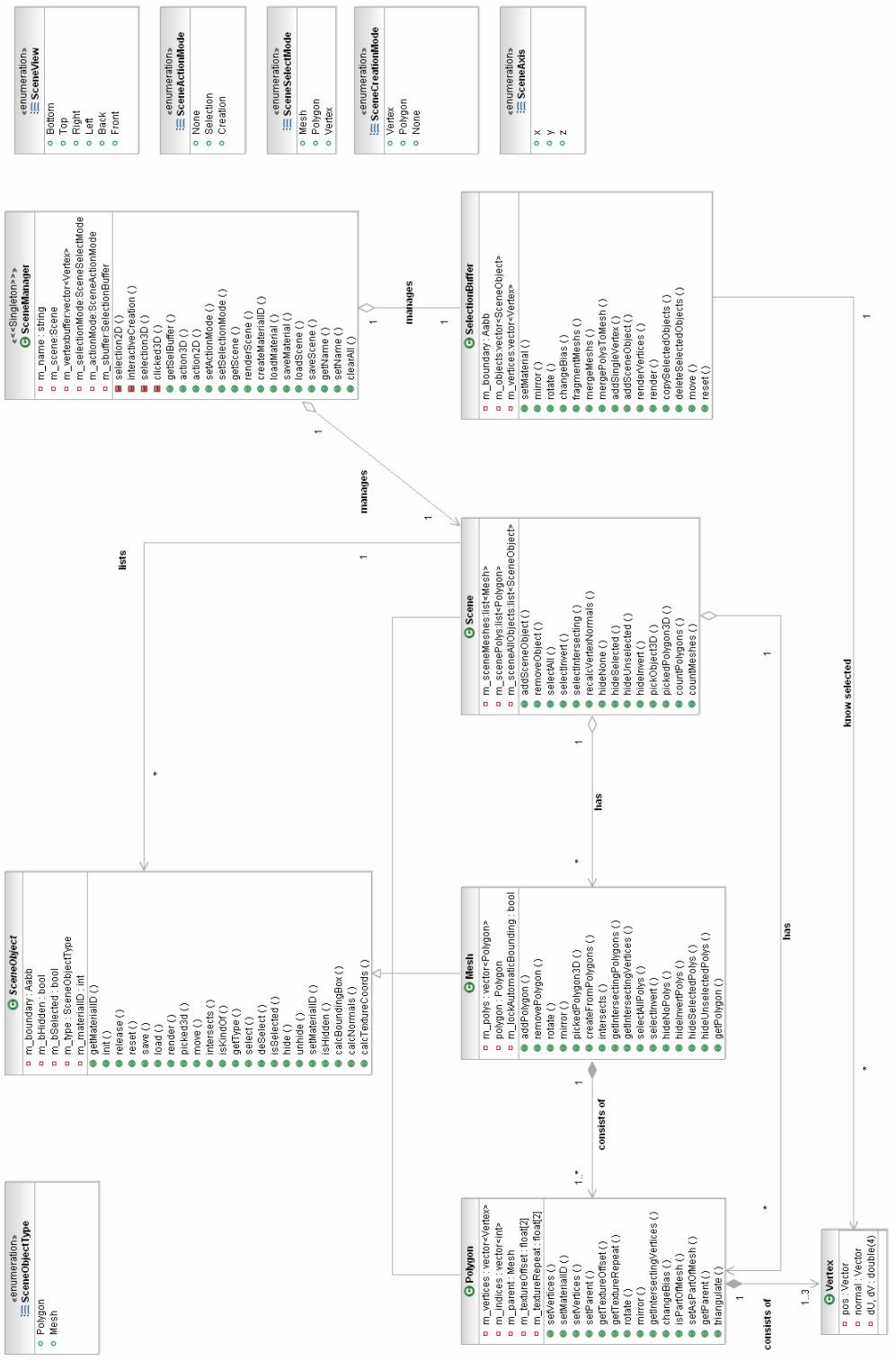
5.9 Globale Entwurfsentscheidungen

- 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 Szenengraph 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 Pluginsystem 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
  - 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 Szenelementen 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 Anforderugen 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::TextureManager gehalten, der Scenegraph kennt dagegen nur die vom TexturManager vergebene ID. Um jedoch Materialänderungen durch den Benutzer zu erfassen

1.2.3 Anforderugen betreffend Komponente Renderer

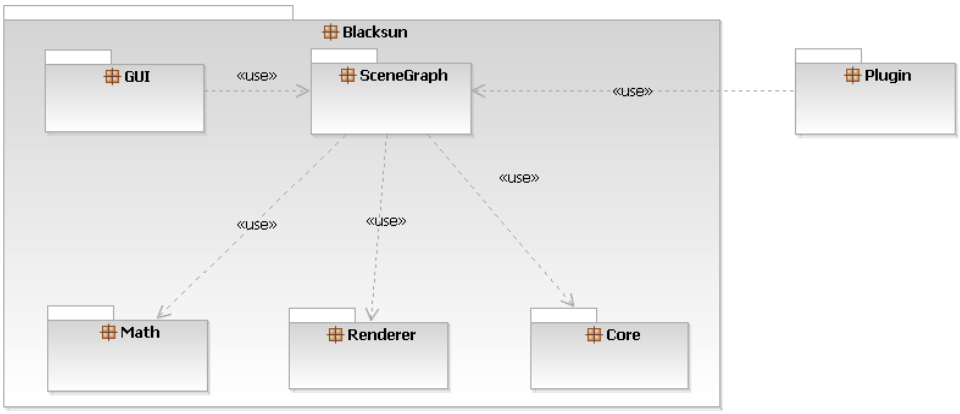
- Wenn sich ein beliebiges Detail am Szenenzustand geändert hat, ist die Funktion zur Neudarstellung des Renderers aufzurufen.
- Jedes Szenenobjekt muß über eine ID für das Material verfügen. Physikalisch werden Materialien im Renderer::TextureManager gehalten, der Scenegraph kennt dagegen nur die vom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

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 der durch den Anwender stattfand, ruft die GUI den SceneManger 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 Hauptwerkzeugleiste 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. Wichtig ist dabei nur zu wissen, ob eine Mausaktion in einem 2D- oder einem 3D-Ansichtsfenster stattgefunden hat.

  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.

- ***SelectionBuffer:*** Der SelectionBuffer managed die Selektion von Szenenobjekten. Er merkt sich dazu die Verweise auf die tatsächlichlen 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.

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

  Auch kapselt der SelectionBuffer die Funktionalität zur Umwandlung von mehreren Polygonen zu einem Mesh. Der Anwender selektiert zum Bespiel 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.

  Wichtig ist der SelectionBuffer für die Auswahlmöglichkeiten in der Editor-GUI. Es ist beispielsweise möglich, eine Selektion beliebig oft zu duplizieren.

- ***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.

  Bei der Selektion aus der GUI heraus ist es vonnöten eine Unterscheidung zwischen einem Mausklick in der orthogonalen (2D-Ansicht) und der perspektivischen (3D-Ansicht) Sicht zu treffen. Während bei der 2D-Ansicht eine rechteckige Auswahfläche über eine Szene gelegt wird, muß bei einer 3D-Anwahl ein sogenannter Strahl (Klasse Ray im Paket Math) für die Kollisionserkennung benutzt werden. Aus diesem Grund gibt es für orthogonale Auswahl die Funktion *intersects* und für perspektivische Auswahl die Funktion *picked3D*.

  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 sozusagen 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*.

- ***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 mindestens einem. 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 Polyonge berechnet wird, was beim einzelnen Einfügen nicht der Fall wäre. Neben dem Einfügen ist natürlich aus das spätere Entfernen einzelner Polygone aus einem erstellten Mesh möglich.

  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 zahlt sich die Designentscheidung zur Selbstverwaltung der Szenenobjekte somit durch Einfachheit aus.

  Zur Kollisionsabfrage steht neben denen von SceneObject übernommenen Methoden für orthogonale und perspektivische Auswahl noch weitere Arten von Kollisionsabfragen zur Verfügung. Da sich der Anwender entschließen kann, anstatt ganzer Meshes auch nur einzelne Polygone auszuwählen, wurden spezielle Funktionen hinzugefügt: *getIntersectingPolygons* (2D) und *pickedPolygon3D* (3D).

- ***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 3 Vertices.
  Eine Szene setzt sich genaugenommen nur aus einzelnen Polygonen oder Gruppen von Polygonen (siehe Klasse Mesh) zusammen. Der Anwender kann zwar einzelne Vertices editieren, aber es ist nicht möglich einzelne Vertices in eine Szene einzufügen oder aus ihr heraus zu löschen. Vertices ohne Zugehörigkeit zu einem Polygon sind per Definition ausgeschlossen und exisitieren daher nicht autark.

  Neben den reinen Vertexdaten muß ein Polygon sich auch noch Indize zu den Verticen 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 vorgefertige *triangulate*-Funktion verwenden, welche eine sequentielle Indexreihenfolge erzeugt.

  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.

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).

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->getScene().addSceneObject(box);

// Eventuell vorhandene Selektionen im SelectionBuffer verwerfen
sm->getSelBuffer().reset();
// Neues Objekt als Selektion im SelectionBuffer hinzufügen
sm->getSelBuffer().addSceneObject(tischplatte);
```

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.

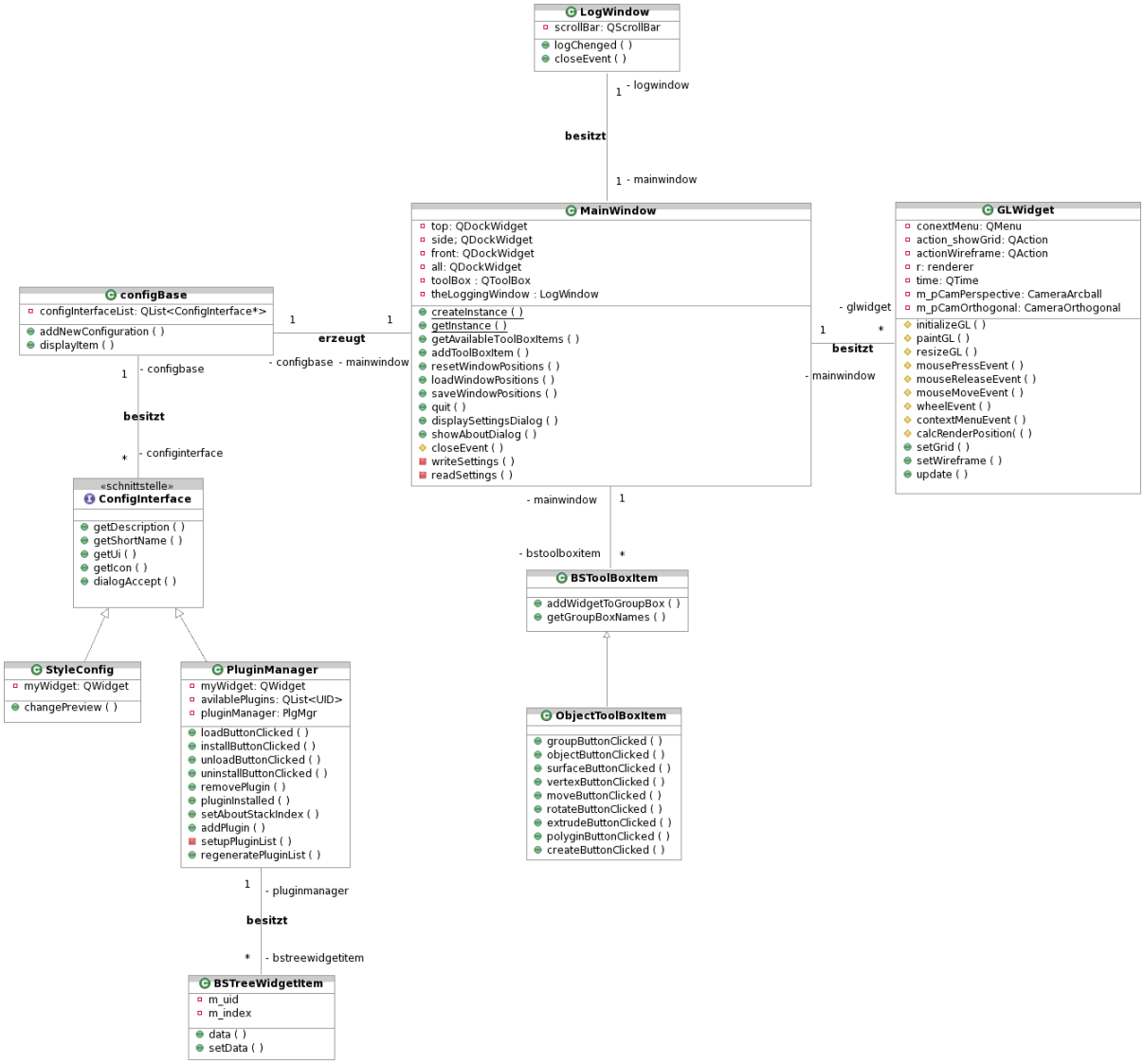
6.8 Globale Entwurfsentscheidungen

- 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 verlorengehende 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

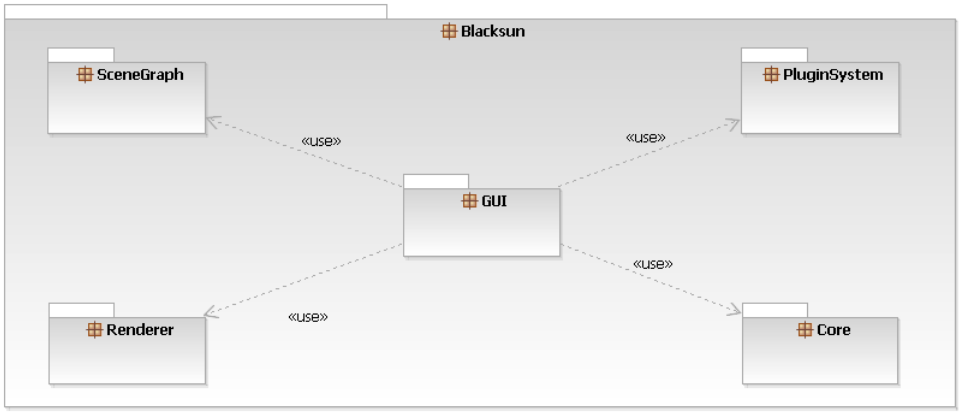


7.2 Funktionale und nicht funktionale Anforderungen

- **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.
- **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. Ausserdem frägt dieses Widget für jedes Plugin ab, ob es eventuell ein eigenes Konfigurationswidget hat.
- ***ConfigInterface:*** Dieses ist eine abstrakte Basisklasse die implementiert werden muss wenn ein eigenes Konfigurationswidget im Konfigurationsfenster darzustellen ist.
- ***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.

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 dynamsichen 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 Kode 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 zu anderen Plugins 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 Vereinbahrung 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 Funkton:
AnderesPlugin* ap = PlgMgr::getInstance()
 ->getPlgInstance(AnderesPlugin::uid,Version(1,0,0));

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

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

9 Ergänzungen

10.1 Das Makefile-Konzept

Das hier angewante 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. Dies wurde gemacht, da alle die die QT – GUI benutzen, dies sowieso machen müssen (da diese zuvor noch mit dem sogenannten *moc* – Compiler verarbeitet werden müssen), sich somit ein einheitlicheres Bild zeigt, aber auch, da die *qmake* Projektdateien verständlicher sind als 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.

10.2 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 ausführbare Programm kleiner ist.

10.3 Singletons

Alle Kernkomponenten zeichnen sich durch das Singleton – Konzept aus. Die einzige Instanz dieser Klasse wird über *createInstance* erzeugt und über *getInstance* kann darauf zugegriffen werden. In der main – Funkton werden zu Beginn alle Komponenten mit *createInstance* erzeugt. 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;
}
```