

§1. Programmiersprache

(1) Wir programmieren in C/C++. Die Pointer-Deklarationen stehen immer am Typen zu dem sie gehören.

Negativbeispiel:

```
virtual float * getPointNormalArray( size_t *n = NULL);
```

geht so gar nicht! C++-Version:

```
virtual float* getPointNormalArray(size_t &n);
```

(2) Referenzen sind Pointerübergaben wann immer es Sinn macht zu bevorzugen.

(3) Pointer-Variablen werden mit 0 bzw. NULL initialisiert.

```
float* foo = 0;
```

(4) Der Stern für den Pointer steht hinter dem Typ. Nicht vor dem Variablennamen (ich gebe zu das ist Geschmackssache, da ich mir die Mühe mache das hier aufzuschreiben entscheide ich mich für die Variante die ich aus verschiedenen Gründen schöner finde (alter Pascal-Programmierer ;-)). Auf gar keinen Fall wird der Typ mit dem Variablennamen multipliziert:

```
float * foo = 0;
```

(5) Jede Variable wird in einer eigenen Zeile deklariert (wegen (4)).

(6) In C++-Programmen erfolgt die Ausgabe auf der Konsole mit `std::cout`. In C-Programmen wird `printf(...)` verwendet. Auf keinen Fall beide Varianten mischen.

§2. Dateistruktur

(1) Alle Dateinamen beginnen mit einem Großbuchstaben. Enthalten Sie Klassen-Code heißt die Datei wie die Implementierte Klasse mit entsprechender Endung.

(2) Header haben die Dateiendung `.hpp`, nicht-template-Implementierungen haben die Endung `.cpp` und werden jeweils in ein eigenes Objectfile compiliert und hinterher zur lssr-Bibliothek zusammengelinkt.

(3) Templatecode wird in `.tcc`-Dateien ausgelagert.

(4) In der Regel wird für jede Klasse ein Satz `.hpp / .cpp / .tcc`-Dateien angelegt.

(5) Die Benennung ist CamelCased. Beinhaltet der Klassenname ein Akronym, wird dieses in Großbuchstaben geschrieben.

Beispiele:

```
class MyFirstClass;  
class PCLPointCloudManager;
```

entsprechend würden im schlimmsten Fall folgende Dateien benötigt

```
PCLPointCloudManager.hpp  
PCLPointCloudManager.cpp  
PCLPointCloudManager.tcc
```

(6) Danach folgt eine Angabe des / der Autoren, die an der Datei gearbeitet haben (im Doxygen-Style)

Beispiel:

```
/**  
 * MeshIO.hpp  
 *  
 * @date 04.08.2011  
 * @author Thomas Wiemann (twiemann@uni-osnabrueck.de)  
 * @author Lars Kiesow (lkiesow@uos.de)  
 */
```

§3. Code Style

(1) Code Style ist Allman / BSD.

Beispiel:

```
int f(int x, int y, int z)  
{  
    if (x < foo(y, z))  
    {  
        haha = bar[4] + 5;  
    }  
    else  
    {  
        while (z)  
        {  
            haha += foo(z, z);  
            z--;  
        }  
        return ++x + bar();  
    }  
}
```

(2) Man beachte, dass einzeilige Code-Blöcke ebenfalls geklammert werden!

(3) Der Rückgabewert einer Funktion steht in der selben Zeile wie der Methodenkopf.

(4) Zwischen mindestens Binären Operatoren und Variablen sind ausnahmslos Leerzeichen zu setzen. Bei unären müssen keine Leerzeichen gesetzt werden, es sei denn diese erleichtern die Lesbarkeit (siehe Abs. (6)).

Ja:

```
int x = 5 + y;
```

Nein:

```
int x=5+y;
c ++
```

(5) Namespaces werden nicht eingerückt. Das Ende eines Namespaces ist zu markieren:

```
namespace lssr
{
    class MyClass
    {
        ...
    };
} // namespace lssr
```

(6) Werden Blöcke von Variablen oder sehr ähnlichen Aufrufen geschrieben, sind Tabs so zu setzen, dass man die Struktur des Zugriffs gut erkennen kann.

Beispiel:

```
if(m_faceNormals != 0) delete[] m_faceNormals;
if(m_vertices    != 0) delete[] m_vertices;
if(m_colors      != 0) delete[] m_colors;
if(m_indices     != 0) delete[] m_indices;
```

Oder

```
m_faceNormals = new float[3 * o.m_numVertices];
m_vertices    = new float[3 * o.m_numVertices];
m_colors      = new float[3 * o.m_numVertices];
```

Oder

```
glVertex3f(v[a], v[a + 1], v[a + 2])
glVertex3f(v[b], v[b + 1], v[b + 2]);
glVertex3f(v[c], v[c + 1], v[c + 2]);
```

Das hilft, Fehler bei der Nummerierung der Indizes zu finden. Das "Gliedernde Element" (z.B. der Zuweisungsoperator wird dazu auf die minimale Tab-Position eingerückt, die benötigt wird, um eine saubere Ausrichtung des Blockes zu erreichen.

(7) Tabs sind 4-Stellen weit und werden durch Leerzeichen ersetzt (Kraft autoritärer Willkür).

(8) Die 80-Zeichen-Grenze wird bei uns nicht streng eingehalten, solange der Ausdruck des Codes noch lesbar ist. Umbrüche bei Funktionen mit langer Parameterliste erfolgen folgendermaßen:

```
return getIndexeArrayf(
    n, m_num_points,
    &m_points,
    &m_indexed_points );
```

§4.. Verwaltung von Template Code

(1) Kurze Template-Funktionen < 5 Zeilen dürfen im Header geinlined werden. Alles andere wird in .tcc-Dateien ausgelagert.

(2) Die .tcc-Dateien werden jeweils in der letzten Zeile der .hpp-Datei included.

(3) Template Deklarationen stehen in einer eigenen Zeile vor der Template-Funktion:

Beispiel:

```
template <typename T> void
Foo<T>::bar ( )
{
    ...
}
```

(4) Template-Variablen haben das Postfix T: CoordT, VertexT etc.

§5. Variablennamen

(1) Variablennamen für Interfaces (also in den Deklarationen) sind CamelCased (s. Klassennamen).

(2) Klassenmember beginnen mit dem Präfix m_.

(3) Auch sie sind CamelCased.

(4) Variablen innerhalb Methoden / Funktionen können nach Geschmack des Programierers benannt werden.

(5) Using-Deklarationen dürfen sich nur auf einzelne Klasse beziehen. Komplette Namespaces werden nicht inkludiert: