

PixelLight Conventions Documentation



February 23, 2012
PixelLight 0.9.11-R1



The content of this PixelLight document is published under the Creative Commons
Attribution-NonCommercial-ShareAlike 3.0 Unported
Copyright © 2002-2012 by The PixelLight Team

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 5 |
| 2 | General | 7 |
| 2.1 | Keep it Simple | 7 |
| 2.2 | Encoding | 7 |
| 2.3 | File Extension | 7 |
| 2.4 | C++11 Language Features | 7 |
| 2.5 | For-Scope | 8 |
| 2.6 | Locality and Scope | 8 |
| 2.7 | C runtime stack Vs. C heap | 8 |
| 2.8 | Global Variables and Global Functions | 9 |
| 2.9 | Null Pointer | 9 |
| 2.10 | Overwriting Methods | 9 |
| 2.11 | Casting | 10 |
| 2.12 | Const Correctness | 10 |
| 2.13 | static const Vs. const static | 11 |
| 2.14 | Namespaces | 12 |
| 2.15 | Inline | 12 |
| 2.16 | Dynamic Parameters | 13 |
| 2.17 | Names | 13 |
| 2.18 | Prefix | 14 |
| 2.19 | Postfix | 15 |
| 2.20 | Events and Signals | 15 |
| 2.21 | Event Handlers and Slots | 15 |
| 2.22 | Run-Time Type Information (RTTI) Interface | 16 |
| 2.23 | Reuseability and adding new Stuff | 17 |
| 3 | Documentation | 19 |
| 3.1 | Files | 19 |
| 3.2 | Functions | 19 |
| 3.3 | Tags | 20 |
| 3.4 | Semantic Documentation | 21 |
| 4 | Headers | 23 |
| 4.1 | Including Headers | 23 |
| 4.2 | Header Guard and <code>#pragma once</code> | 24 |

Contents

| | | |
|----------|---|-----------|
| 4.3 | Layout | 24 |
| 4.4 | Operating System (OS) and Application Programming Interface (API) . | 26 |
| 4.5 | Forward Declarations | 26 |
| 4.6 | Completeness | 27 |
| 4.7 | Namespaces | 27 |
| 4.8 | Number of Classes | 27 |
| 4.9 | Template Implementations | 28 |
| 4.10 | Definitions | 28 |
| 5 | Contributing | 29 |
| 5.1 | Welcome | 31 |
| 5.2 | Contributor License Agreement | 32 |
| 5.3 | Commit Checklist | 32 |
| 6 | Contact | 35 |
| | Abbreviations | 37 |

1 Introduction

Target Audience This document is meant for C++ programmers.

Motivation C++ is a mighty programming language, and has evolved over the time - as a result, there are many ways how certain goals can be archived and a lot of pitfalls. This huge freedom often leads to confusion which way is the *best* - and often, the answer is that there's no best way, just widely used practices or literature from C++ gurus and accepted design patterns.

If a project has a certain size, it's usually a good idea to write a document about the used *coding conventions*. So, here's the *PixelLight coding conventions* document.

This documentation can't describe everything¹ - because then you would spend more time in reading and rereading this document instead of actual programming. In here, we especially mention things that are important to us or important for cross-platform compatible source code. The rest should be pretty self explanatory and it should be possible to *find out* other style conventions by self. We strongly recommend you to spend time in writing clean code yourself to support the *work flow*. It's not important *how fast* you wrote the code, it's important that this code is well designed, well commented and cleanly written because you may use this code for a loonng time.²

¹Although it has grown over the years...

²Belief us, at the time of writing we're working already 9 years on PixelLight and are always happy when there's no need to touch code again because it's just fine the way it is

2 General

2.1 Keep it Simple

In case you're one of the persons really hating this phrase and laughing about people using it, stop reading this document because you're probably don't like the rest of it either. For the rest, go on reading.

The world is complicated enough - if there are multiple solutions, prefer the simplest over the most complicated one. This way, the chances are high that other will understand the solution as well as you when looking at the code some years later.

2.2 Encoding

We work with multiple operation systems so we have to take into account *how* text files are saved. All *Diary*, *Readme*, *Todo* and *Plan* text files saved as "Unicode (UTF-8 with signature) - Codepage 65001". All other text files like code or make files saved in classic *ANSI*.

2.3 File Extension

First of all, we use the old fashion *.h*-file extension to mark header files - *hpp* would be the *correct* extension for C++, but it's not widely used. Usually, we outsource inline implementations into files with an *inl*-extension to keep the header files good readable. For source codes, we use the file extension *cpp*.

2.4 C++11 Language Features

Using C++11 (previously known as C++0x) language features is fine as long as

- It's possible to emulate, or at least deactivate the feature within compilers don't supporting it (yet)
- There's no comfortable/acceptable way to solve a task without using the feature, but those situations have to be discussed within the team and community

Currently the following C++11 language features are used:

- *nullptr* - a null pointer literal

- *override* - gives the compiler a chance to detect and blame errors related to overwriting methods

Extern Templates Don't use *extern templates*¹ in order to avoid template instantiation in other modules. This is a feature we can't emulate and there are still legacy compilers like GNU Compiler Collection (GCC) 4.2.1 used on Mac OS X 10.6 actively used around the world. So, at least for now, don't use this useful feature.

2.5 For-Scope

Within the PixelLight projects, the *for-scope* is active by default. The following for instance will produce a compiler error:

```
1 for (int i=0; i<1; i++) {  
2     // Do anything  
3 }  
4 i = 20; // i has already gone out of scope
```

Listing 2.1: for-scope

2.6 Locality and Scope

Locality means that one should keep things together whenever possible. A good example is the *for-scope* from section 2.5. Do not define all variables you're using within a function right at the top of the function. Define and initialize local variables when you first need them. If reasonable, it's also fine to add a new scope by using brackets in order to restrict the life of a local variable which is only required within a tiny section of your implementation. Consider moving this inner bracket into an own function if it makes sense in some way, but this is no requirement if you can't see any point in adding another tiny function.

Having only a hand full of local variables with a well defined scope makes it easier to understand and debug the code. It also avoids trashing the valuable C runtime stack. If there's only a limited number of local variables, a good compiler might also have the chance to produce more effective code. Another benefit of this approach is that this local variable may never be created and touched at all in case the part of the implementation is never executed.

2.7 C runtime stack Vs. C heap

Prefer allocating variables on the C runtime stack over the C heap whenever possible. Dynamic memory allocation and deallocation via the C heap is considered to be slow

¹<http://www2.research.att.com/~bs/C++0xFAQ.html#extern-templates>

compared to using the C runtime stack. The C heap introduces the risk of memory leaks while the C runtime stack is cleaned up automatically when leaving a scope. Also the C heap suffers from something called fragmentation and caching may be an issue. In concurrent programming the C runtime stack is less painful because every thread has it's own C runtime stack, while all threads share the same C heap which is a source of really nasty bugs in case one is sloppy when dealing with dynamically allocated memory.

Be aware of the difference between C runtime stack and C heap. If you're not familiar or unsafe about this terminology and topic, please refer to the widely available literature about C/C++. For example, adding a class member variable instance directly to a class, meaning without using a pointer, doesn't necessarily mean that this will be super fast because this member variable will be automatically allocated on the C runtime stack. If this class is allocated on the C heap, the member variable will be allocated on the C heap as well.

2.8 Global Variables and Global Functions

Whenever possible, try to avoid using global variables and global functions. If you really have to, because e.g. everything else would be overcomplex for a tiny application, document at least why you had to make it global. In general, especially global variables lead to hard to understand, debuggable and maintainable code.

2.9 Null Pointer

For null pointers, use `nullptr`² from *C++11* and not for example the legacy, but traditional `NULL` definition or even directly a integer 0.

```
1 char *pszMyString = nullptr;
```

Listing 2.2: Null pointer

2.10 Overwriting Methods

Put the methods within the base class, which are allowed to be overwritten, into a separate code block so everyone is able to find them at once.

```
1 // [-----]
2 // [ Public virtual MyClass functions ]
3 // [-----]
4 virtual void MyMethod() = 0;
```

Listing 2.3: Virtual methods within a base class

²For example *Microsoft Visual Studio 2010* and *GCC 4.6* have native support for `nullptr`

When overwriting virtual methods within a derived class, put the overwritten methods into a code block telling where those methods originally came from.

```

1 // [-----]
2 // [ Public virtual MyClass functions ]
3 // [-----]
4 virtual void MyMethod() override;

```

Listing 2.4: Overwriting virtual methods within a derived class

Although technically not required, do also add a *virtual* to make it absolutely clear that this is a virtual method. To give the compiler a chance to find and blame possible errors like a signature change within the base class, use the C++11 language keyword *override*.

2.11 Casting

PixelLight is using *C++ style casts* (`int i = static_cast<int>(42.21f)`), not *C style casts* (`int i = (int)42.21f`). It's much easier to search for *C++ style casts*³ and they are less vulnerable to unintended effects as well - and because they are not that compact as the *C style casts*, one may think about it a second time why there's a need for a cast.

GCC GCC, offers an option called `-Wold-style-cast` to let the compiler warn if an old-style (C-style) cast to a non-void type is used within a C++ program.

2.12 Const Correctness

Define functions, variables etc. whenever possible to be constant. By giving the compiler this hint, it may be possible to use special optimizations or uncover bugs within the implementation.

Const Example Have a look at the following example function:

```

1 void MyFunction(PLMath::Vector3 &vPosition)
2 {
3     // ...
4 }

```

Listing 2.5: Non-constant function parameter

In case the function is considered to manipulate *vPosition*, all's fine. Let's continue with another usage example:

```

1 MyFunction(PLMath::Vector3(0.0f, 1.0f, 2.0f));

```

Listing 2.6: Using a temporary variable instance as non-constant function parameter

³`static_cast`, `reinterpret_cast`, `const_cast`, and `dynamic_cast`

The compiler creates a temporary `PLMath::Vector3` instance on the runtime stack and is passing a reference into the function. Looks fine as well? In case the function manipulates the variable passed by reference, the temporary instance is changed. In principle that's no problem because it's thrown away after the function call anyway. In practice, this situation is considered to be evil. Microsoft Visual C++ (MSVC) 2010 will shy tell you via a warning

```
warning C4239: nonstandard extension used : 'argument' : conversion from
'PLMath::Vector3' to 'PLMath::Vector3 &' A non-const reference may only
be bound to an lvalue
```

So, it's possible to just ignore or even deactivate this warning... isn't it? In fact, it isn't because other compilers might not be that tollerant and will throw an error message at you. To sum this up: In the example above, you really want to write

```
1 void MyFunction(const PLMath::Vector3 &vPosition)
2 {
3     // ...
4 }
```

Listing 2.7: Constant function parameter

in order to be cross-platform safe.

Const Exception There's one situation were we do not use `const` - when dealing with function parameters because

```
1 void MyFunction(int nVariable1, int nVariable2);
```

Listing 2.8: Function parameters

is inside headers better readable than

```
1 void MyFunction(const int nVariable1, const int nVariable2);
```

Listing 2.9: Constant function parameters

In this situation, the readability is more important for us. This rule does not apply for pointer or reference parameters like

```
1 void MyFunction(const String &sVariable);
```

Listing 2.10: Constant function pointer/reference parameter

because the user should be able to see whether or not a function is going to manipulate the parameter variable.

2.13 static const Vs. const static

Use `static const` instead of `const static`. Have a look at e.g. the GCC option `Wold-style-declaration` resulting in the warning

‘static’ is not at beginning of declaration

or into chapter 6.11 of ISO C99 (*Future Language Directions -> Storage-class specifiers*).

2.14 Namespaces

PixelLight is using multiple namespaces, one for each sub-project. If you want to use for instance the string class which is defined in *PLCore* you need to do this:

```
1 PLCore::String sMyString;
```

Listing 2.11: Explicit namespace

Or this:

```
1 using namespace PLCore;  
2 ...  
3 String sMyString;
```

Listing 2.12: Using namespace

Try to avoid using *using namespace* too often or this will result in name conflicts which you then have to resolve by hand by adding for instance *PLCore::*. We recommend to never use *using namespace* within header files.

2.15 Inline

Whenever reasonable, mark methods as inline in order to give the compiler a chance to optimize out function calls. This may not always result in a huge performance improvement, but we have to use features like inline to get a decent overall performance and restricting the performance impact introduced by heavily using Object-Oriented Programming (OOP). This is especially true when considering devices like Smartphones which have more restricted resources as a powerful workstation. For example, by marking a getter-method as inline, requesting the value of a member variable may not introduce any performance drawback compared to making the attribute public and accessing it directly.

Potential candidates for inline are one-line-methods which don't require to include more headers within the class headers, meaning that the header complexity is not significantly increasing by using inline. Other candidates are methods which require only to include really lightweight headers or headers which have usually already been included. Beside the mentioned one-line-methods, methods with multiple instructions and lines can also be marked as inline as long as you consider it as reasonable after thinking it thru again. Complex methods should not be marked as inline to avoid increasing the resulting binary size and therefore making the life of the cache harder. Do only mark complex methods as inline if there are extremely good, and especially documented reasons for doing so. Do not inline methods if this requires to include system or so called

standard-headers, except there's a really good and heavily documented reason for doing so.

Do not directly write the inline implementation into the class header. The class headers have to stay human readable and shouldn't disclose to many implementation details - not because they are top secret, but because the class has to be usable by only looking at it's interface. So, move the inline implementations into a separate *inl* file as described within section 2.3. Of course, the inline implementation is still visible to the user, but not when looking at the class header directly.

2.16 Dynamic Parameters

When dynamic parameters are used and the name of the parameters inside a string is irrelevant, as this is the case for *PLCore::Params::FromString*, the parameters are named using *Param<x>* where x starts with 0 (example: *Param0=1 Param1="Hello"*).

2.17 Names

In general, names of classes, functions, variables and so on must have human readable names. The name has to tell as much as possible about the usage - if the user can guess correctly the usage of for example a variable by just looking at it's name, the name is perfect.

General rules:

- A single character as name for local (really, only local) control variables like *i* within a for-loop is acceptable as long as there are not too much of those at once (else use reasonable names to avoid confusion)
- Short cuts should be avoided whenever possible because they may lead to confusion⁴ (no stuff like *stricmp()*)
- If there's a *commonly used* name for something, use this name instead of creating a totally new one. Example: Don't call an array class *Bob* just because this name is cool - use the name *Array* for an array class instead and the users might be able to find it.
- Avoid long names, if there's an expressive shorter name it's the preferred one... but keep the short cut rule in mind

Classes, structures and so on have a upper case letter at the beginning. Example:

⁴True story: When using *Rot* as short cut for *Rotation*, we once had the situation that a German speaking programmer asked confused what the color *Rot* should do inside the scene node... in German, *Rot* is the word for *red*...

```

1 class Player {
2 };
3 struct Info {
4 };

```

Listing 2.13: Name convention

2.18 Prefix

Because the readability of code is extremely important when working in a team and/or using code from others, one of our goals was to make the PixelLight code as readable and well structured as possible. We are using a name style convention⁵.

Variable prefixes for standard types:

| | | | |
|----|--|------------------|--|
| 1 | Type | Prefix | Example |
| 2 | <code>bool</code> | <code>b</code> | <code>bool bActive</code> |
| 3 | | | |
| 4 | (n for all none standard floating point types) | | |
| 5 | <code>int</code> | <code>n</code> | <code>int nNumber</code> |
| 6 | <code>char</code> | <code>n</code> | <code>char nCharacter</code> |
| 7 | <code>long</code> | <code>n</code> | <code>long nHuge</code> |
| 8 | | | |
| 9 | <code>float</code> | <code>f</code> | |
| 10 | <code>double</code> | <code>d</code> | |
| 11 | | | |
| 12 | (Character arrays -> strings) | | |
| 13 | <code>char[]</code> | <code>sz</code> | <code>char szName[64]</code> |
| 14 | <code>char*</code> | <code>psz</code> | <code>char *pszName</code> |
| 15 | | | |
| 16 | (Pointers) | | |
| 17 | <code>*</code> | <code>p</code> | <code>Player *pPlayer</code> |
| 18 | | | |
| 19 | (References) | | |
| 20 | <code>&</code> | <code>-</code> | <code>char &nTest = nTest2;</code> |
| 21 | | | |
| 22 | <code>struct</code> instance | <code>s</code> | <code>Info sPlayer (struct Info)</code> |
| 23 | | | |
| 24 | <code>class</code> instance | <code>c</code> | <code>Player cPlayer (class Player)</code> |

Listing 2.14: Variable prefixes for standard types

⁵We know that there are a lot of discussions around the internet whether or not prefixes should be used. In the year 2002 we decided to use them and we don't change it - due to the dimension of the PixelLight project, it would be a huge effort to change it anyway.

General variable prefix for class variables: `m_` (m for member)

Example: `char *m_pszName`

Variable prefixes for PixelLight types:

| Type | Prefix | Example |
|---|--------|---|
| String | s | String sName |
| Container | lst | List lstNames |
| Map | map | HashMap mapNames |
| VectorX (X for dimension: 2, 3 or 4) | v | Vector3 vPosition |
| MatrixXxX (X for dimension: 3 or 4) | m | Matrix4x4 mRotation |
| Quaternion | q | Quaternion qRotation |
| ColorX (X for dimension: 3 or 4) | c | Color3 cColor (same as <code>class</code>) |

Listing 2.15: Variable prefixes for PixelLight types

2.19 Postfix

We recommend you to use the PixelLight name convention and marking debug versions with a *D* at the end of the filename. Example: *MyPlugins.dll* = release version, *MyPluginsD.dll* = debug version.

2.20 Events and Signals

As soon as an event is inside a class, we refer to it as *signal*. As such, the prefix *Event* like within *EventKeyDown* is used outside classes while prefix *Signal* like within *SignalKeyDown* is used inside classes.

2.21 Event Handlers and Slots

Within our name convention for event handlers and RTTI slot names, there's a *On* within for example *OnMyEvent* indicating that this is a handler/slot method. The other part of the name consists of the event/signal - for *OnMyEvent* this would be an event/signal with the name *MyEvent*.

2.22 RTTI Interface

Within PixelLight, the RTTI class properties and members are always defined in the following order:

- Properties
- Attributes
- Constructors
- Methods
- Signals
- Slots

This way one knows exactly where to look for something. Further, within the RTTI class properties and members definitions, only tabs and no spaces are used to make it easier to write the definitions like a table. This makes it more comfortable for the eyes and brain to navigate to certain definition parts without too much searching around.

Here's an example source code showing the common RTTI interface layout (without the word wrap):

```

1 // [-----]
2 // [ RTTI interface ]
3 // [-----]
4 pl_class(pl_rtti_export, MyRTTIClass, "", PLCore::Object, "Sample RTTI
   class, don't take it to serious")
5 // Properties
6 pl_properties
7   pl_property("MyProperty", "This is a property value")
8 pl_properties_end
9 // Attributes
10 pl_attribute(Name, PLCore::String, "Bob", ReadWrite, GetSet, "A
   name, emits MySignal after the name was changed", "")
11 pl_attribute(Level, int, 1, ReadWrite, DirectValue, "Level,
   automatically increased on get/set name and OnMyEvent", "")
12 // Constructors
13 pl_constructor_0(DefaultConstructor, "Default constructor", "")
14 // Methods
15 pl_method_0(Return42, int, "Returns 42", "")
16 pl_method_1(IgnoreTheParameter, void, float, "Ignores the
   provided parameter", "")
17 pl_method_0(SaySomethingWise, void, "Says something wise",
   "")

```

```

18 pl_method_0(GetSelf,      MyRTTIClass*,      "Returns a pointer to this
    instance", "")
19 // Signals
20 pl_signal_1(MySignal, PLCore::String, "My signal, automatically
    emitted after the name was changed", "")
21 // Slots
22 pl_slot_0(OnMyEvent,    "My slot",    "")
23 pl_class_end

```

Listing 2.16: RTTI interface (without the word wrap)

2.23 Reuseability and adding new Stuff

Before you add new classes, functions and so on - check first whether there's already something similar within PixelLight. If there's something you can already use directly, use it instead of writing new stuff. If there's something quite similar, have a more detailed look at it and contact your team colleagues to discuss whether a refactoring is possible and reasonable to update and/or to enhance existing stuff.

This is also true for adding new precompiler definitions. Look deeply whether or not there's already a definition (or better method) existing which can be used. If new precompiler definitions are really required we should discuss them first because adding new of those may make it harder to compile PixelLight in the correct way.

Reuseability is one of the most important concepts when creating frameworks like PixelLight... and reuseability does not mean that it's possible to copy'n'past it and then hacking around for a certain project. Reuseability means that it's possible to directly reuse, to share something between multiple projects in a quite universal way without the need to enhance and hack around constantly.

3 Documentation

Source code comments are extremely important in order to make it possible to work with the source code in an efficient way. Also, comments can be considered as a kind of bug uncover mechanism. If a source code doesn't match what the comment is describing, it's either a source code bug or a comment bug. In either way, this situation is suspicious and has to be reviewed. The PixelLight code is extensively documented via comments and everything is in English. The interfaces are documented using a style Doxygen¹ understands and therefore can create code documentations automatically. Further, we are using a comment style convention and order.

3.1 Files

At the top of the files you will find a short comment header:

```
1 /*****\
2 * File: <filename> *
3 * <Optional file comment if useful (not within the
4 * source like 'cpp'-files itself)>
5 \*****/
```

Listing 3.1: Header comment

3.2 Functions

Above each class/function declaration there's a comment block which give some hints about the class/function.

```
1 /**
2 * @brief
3 * <Brief function description>
4 *
5 * @param[in] <InputVariable>
6 * <Input variable description, the 'original' value
7 * from 'outside' is not manipulated>
8 * @param[out] <OutputVariable>
9 * <Output variable description, the 'original' value
```

¹Doxygen can be downloaded from <http://www.doxygen.org/>

```

10 *   from 'outside' may be manipulated>
11 * @param[in, out] <InputOutputVariable>
12 *   <Input and output variable description, the 'original'
13 *   value from 'outside' may be manipulated>
14 *
15 * @return
16 *   <Return value description>
17 *
18 * @remarks
19 *   <Detailed description>
20 *
21 * @note
22 *   - <Some hints and good to know information>
23 *
24 * @see
25 *   - <Reference to another documentation for further
26 *     reading>
27 */

```

Listing 3.2: Class/function comment block

The *@brief* part should consist of one phrase only. If you can't describe a class, function etc. within a single phrase you should consider to refactor your code...

3.3 Tags

Within the code documentation we use multiple tags to *mark* points within the code so we can find all marks simply by searching for the tags. Do also don't forget to write a comment to the tag even if you think a comment would be useless there... especially then.

```

1 // [TODO] Here's still something that must be done
2
3 // [HACK] Really terrible solution for a problem, it works but
4 // we are not happy with the solution -> if you have to use
5 // this tag frequently, something must be totally wrong...
6
7 // [DEBUG] Code that is just there for debugging purposes and
8 // may be removed later within the development process
9
10 // [DEPRECATED] Code that exists only for backward
11 // compatibility and may be removed within a next version

```

Listing 3.3: Comment tags

3.4 Semantic Documentation

You can also add documentation with semantic our *PLProject* code post processing tool *knowns*.

The following adds information to *Microsoft Visual Studio* configuration files so we don't jump *into* methods of the string class during debugging:

```
1 // [-----]
2 // [ Debugging options for Microsoft Visual Studio ]
3 // [-----]
4 // <<nostepinto>> PLCore::String::.*
```

Listing 3.4: Semantic documentation

For more information look at the *Microsoft Visual Studio* documentation on *nostepinto* and so on.

4 Headers

During the development of PixelLight we take especially care of the headers because the time the projects need to compile depends heavily on how the headers are build up and how platform and API independent the project is.

We have several rules for our header files. See below. At the first look some or all may look extreme, maybe even crazy to you - but it works really well and solves many problems occurring when using such *nasty* headers directly.

4.1 Including Headers

Order When including files, the order should be

- System headers like *windows.h* on MS Windows
- External library headers
- Project headers

In theory, the order of including headers shouldn't matter. In practice, especially when using system headers there might be situations where one has to play around with the header inclusion order to get it compiled. Those situations are really nasty and have to be avoided whenever possible, but hacking around in system headers is no option. That's the reason why system headers have to come first. If there's need for a certain, fixed header order it's required to document that it's required and why. If it's not documented, another person might change the order in the believe that the header inclusion order doesn't matter... and it might work for the used platform, but now breaks it for another one.

Slash Vs. Backslash When including files, use slashes (/), not backslashes (\). Meaning

```
1 #include <PLCore/String/String.h>
```

Listing 4.1: Slash Vs. Backslash - Right

and not

```
1 #include <PLCore\String\String.h>
```

Listing 4.2: Slash Vs. Backslash - Wrong

While MSVC is accepting both versions, other compilers like GCC only accept slashes (/) and are creating an error for the backslash (\) version.

4.2 Header Guard and `#pragma once`

Each header has to use definitions and `#pragma once` to prevent multiple inclusion - the last one is enough and the preferred way to go, but not each compiler does support it¹.

```

1 #ifndef __PLCORE_STRING_H__
2 #define __PLCORE_STRING_H__
3 #pragma once
4 // ...
5 #endif // __PLCORE_STRING_H__

```

Listing 4.3: Header guard

4.3 Layout

We have a strict layout for the header files so someone now's exactly were to look for something. Additionally we use blocks² with a width of 60 characters to subdivide the header into sections: (includes, classes, functions)

```

1 //[-----]
2 //[ <Section name> ]
3 //[-----]

```

Listing 4.4: Code section comment blocks

This is also used inside class definitions to mark private, public, etc. class parts. Further we tried to make the code as clean and well designed as possible because when working with that many lines of codes it would break your neck if the code is ugly because you will need longer find certain sections and understand the code itself.

Below you can see the complete header layout. If a block is not required, do not use it in a header.

```

1 /*****\
2 * File: <Filename>.h *
3 \*****/
4
5
6 #ifndef __<PROJECT_HEADERNAME>_H__
7 #define __<PROJECT_HEADERNAME>_H__
8 #pragma once
9
10
11 //[-----]
12 //[ Includes ]

```

¹ `#pragma once` is a widely supported preprocessor directive, but non-standard

² You can also call them *eye catcher* if you want to

```

13 // [-----]
14 <Includes required by this header>
15
16
17 // [-----]
18 // [ Forward declarations ]
19 // [-----]
20 <All your forward declarations required by this header>
21
22
23 // [-----]
24 // [ Definitions ]
25 // [-----]
26 <Definitions. Because they are namespace independent they are put in
    right here. You can also call them 'Macros', 'Macro definitions' and
    so on - but it has to be clear that this are '#define' things that
    are best avoided if possible.>
27
28
29 // [-----]
30 // [ Namespace ]
31 // [-----]
32 namespace <Project name> {
33
34
35 // [-----]
36 // [ Forward declarations ]
37 // [-----]
38 <If there are only forward declarations within the same namespace you
    can also put them in here, else put them into the forward
    declarations block above>
39
40
41 // [-----]
42 // [ Template instance ]
43 // [-----]
44 <Template instance(s)>
45
46
47 // [-----]
48 // [ Global functions ]
49 // [-----]
50 <Global functions - normally there's no reason to use those>
51

```

```

52
53 // [-----]
54 // [ Classes ]
55 // [-----]
56 <All your classes>
57
58
59 // [-----]
60 // [ Namespace ]
61 // [-----]
62 } // <Project name>
63
64
65 // [-----]
66 // [ Includes ]
67 // [-----]
68 <Internal includes like for instance template implementations>
69
70
71 #endif // __PROJECT_HEADERNAME_H__

```

Listing 4.5: Complete header layout

4.4 OS and API

To be as platform and API independent as possible, we keep the usage of OS and API headers - even so called *standard headers* to a minimum. As a nice side effect, the compiler time becomes much shorter because this mentioned headers often come with an extreme overhead and we reduce unsolvable name conflicts - there are many headers out there using ton's of definitions that don't work with namespaces and introducing annoying name problems. A general rule for our PixelLight headers is to never ever include such headers in own public headers the application programmer will work with when it's possible to avoid this usage.

4.5 Forward Declarations

Use *forward declarations* whenever possible instead of includes when something is just referred without the need of knowing exactly what it. Here's a compact example class:

```

1 class MyInterface {
2     void SetName(const PLCore::String &sName);
3 };

```

Listing 4.6: Using forward declaration

Someone can now just throw in an include like:

```
1 #include <PLCore/String/String.h>
```

Listing 4.7: Include

But now, your header depends on another one and *PLCore/String/String.h* is always processed if your header is used, even if the string class is never used. Even worse, if *PLCore/String/String.h* is changed a lot of other codes need to be recompiled, if really required or not. Long compiler times are definitively not productive.

Now someone may cry out "*Use cool precompiled headers!*", but this is no option for us to use additional stuff just to fix messy and not well profound headers. Keep it simple. We grab the problem on it's root by doing for example the following:

```
1 namespace PLCore {
2   class String;
3 }
```

Listing 4.8: Forward declaration

4.6 Completeness

Each header must provide all information the compiler needs to compile it. So, add includes or if possible the preferred *forward declarations*.

4.7 Namespaces

Except for headers for the final application, each header has to put it's content into a appropriate namespace.

```
1 namespace PLCore {
2   class MyClass {
3   };
4 } // PLCore
```

Listing 4.9: Namespace definition

It's not allowed to use *using namespace* - this may introduce ugly name conflicts in other projects.

4.8 Number of Classes

Whenever advisable, do only put one class into one header. The name of the header and the name of the class have to be equal. Within seldom situations when it's really a

good idea, you can put multiple classes into one header, but as mentioned, do this only if you are sure this is a good way to go.

4.9 Template Implementations

When defining templates within a header, do not put its implementation into the same file, too. Normally, templates are not really good readable, the implementation directly in the same file would not increase the readability. (maybe it would even scare of the reader :)

Put the template implementation in special files with the same filename as the header, but with the file extension *.inl*. Include this file at the bottom of the template header.

4.10 Definitions

Do only use *#define* within a header if there's no other way to do it. Normally *enum*, *typedef* or something like *static const int MyConstant* will also do the job and beside will respect namespaces.

5 Contributing

Sorry in advance for the following quite long introduction. Don't take everything too serious, it should just point out what can go wrong within a free open-source project and hints how this can be avoided. PixelLight is in development since the year 2002 and experience tells that it's required to mention a few things, first. The intent of this chapter is to make the collaboration with and within the PixelLight developer team more comfortable, relaxed and to reduce time and energy consuming pointless detail discussions to a minimum. We should spend most of our time for developing, not arguing. It should be fun to work on PixelLight, not frustrating due to fighting inside the team. Please note that we are no lawyers nor do we have money to spend on lawyers in order to check this text. In case you love to sue other people, please stay away from this free open-source project. There's no money to get out of it.

Tiny Contributions We hope that a lot of people will provide patches and small contributions, but we hope you understand that, for organisation purpose, we can't accept every tiny contribution. As in every open-source project you will be mentioned in the repository as contributor but it's not possible to list everybody within the *AUTHORS*-file. Nevertheless we will thank you very much for your efforts to make PixelLight a better and more stable software. For sending bug reports or providing a prebuild external package we are really thankful and everyone who is using PixelLight will profit from this helpful contribution.

Warning about Convention Before you contribute something please note that we take coding conventions and proper behaviour within the development team seriously. In order to manage a big project like PixelLight, rules, discipline, respecting each other and especially structure and order are important. That's why this document exists in the first place¹. If the laws of the game are known, one can focus on the development itself without, or at least reduced constant and time consuming discussions about for example code style or whether or not to use a certain C++ feature. Additionally, the team members are frequently making code reviews. This means that code you contribute may be constantly changed or even completely removed if it's no longer required or there are other serious reasons for doing so. We will also change pointless and stupid stuff like spaces and tabs, moving around code, adding nonsense comments for code where it should be clear to everyone what it's doing and why. In case you're hating it when other developers are touching your code, or you demand that before someone in the team changes anything should first have to contact you - maybe you should think again

¹... or maybe because we German guys are over exact by nature...

about contributing to this project. The development of a technology like PixelLight is an iterative and evolutionary process in which we already refactored or even dropped code we spend a lot of time in writing it for the greater good. So, even if this sounds extreme, we don't accept contributions blocking the further development of the technology. Sorry. In case you're open minded, willing to share and learn - please read on.

Warning about Copyright and Patents We take copyright and patents seriously in order to avoid conflicts. In case we become aware that you contributed something which is not from you and you haven't mentioned the source which has to be compatible with PixelLight's license, the contribution will be removed. If you follow the rules applying to scientific work, there shouldn't be any problems. Any contribution will become part of PixelLight, meaning that it also will be released under the same license. If the PixelLight licence is changed for example to a less restrictive licence at some point of time, you agree that this is ok for you.

Errors and Bugs We expect that the contribution is free of any errors... just kidding. As you might now, it's impossible to prove that, at least a little bit longer code, is free of any errors or undesired side effects. But we expect that you're taking responsible measurements and best practices in order to at least reduce the number of errors which can be avoided. If possible, do also provide unit tests so we can ensure that the quality stays high over the time by performing automatic unit tests. If we notice any errors, breaking stuff and so on no one will yell at you because something like this happens even to the most experienced developer. In case we're noticing the same error over and over again, we might discuss how we can avoid, or at least reduce it in the future to keep the productivity at a decent level. But to be fair, you also have to accept that the other team members are not perfect and will make errors on a regular basis. We have to work together to keep up the quality as high as possible.

Behaviour Be as polite as possible to each other. Criticism is always good in order to get improvement, but do never get personal or take something personal. Be careful with something like jokes, because a person from another culture may totally misunderstand it. Even persons from one and the same culture may think totally different, take this into account. Experience shows that a smooth teamwork is really hard to achieve, but please, show your best side.

Language Do only write in English. English isn't the native language of the project founders, it's German and that's the only reason why the previous diary entries are in German. To be able to work on a global scale with an international team, everything must be in English so that everyone has at least a real chance to be able to understand it without using an automatic translator.

Documentation and Diaries Source code has to be documented. Thrown in undocumented source code is worthless. Additionally we're writing diaries right from the

beginning of the project in the year 2002. This makes it possible to figure out who has written certain parts, background information and thoughts can be followed as well. The diaries are also a security measurement in order to detect frequent change forth and change back of certain parts over the years. Although it appears that this diary approach is not standard in software development, when working on PixelLight it has several times proven to be quite useful. As a result it's required to add diary entries describing what you've done, why, helpful insight into your thoughts and everything else that helps to understand and maintain your work. You don't need to write a complete book each time, depending on the made change keywords might also be fine. In general, writing more detailed diary entries has also proven to be a quality enhancement. While explaining what you've done, from time to time one's noticing that there's something totally wrong with the made change. So, we motivate all team members to try to write at least a little bit more detailed diary entries.

Trying to take over PixelLight Sorry to mention this point at all. The team will not accept over dominant persons trying to take over the control of PixelLight and constantly telling us what's best and how things should be handled the right way. Even if it sounds stupid and totally selfishness, a project like PixelLight requires a certain hierarchy within the team in order to get progress and quality. Everything else would end in anarchy, poor results and endless discussions without any sense. In extreme it would lead to the dead of the project due to constant fighting inside the team. We respect the freedom of speech and are trying to give everybody the chance to get more control. But this has to be done by constant contributing and showing the rest of the team that you really know what you're talking about. If there are open questions what's the best for the project in general, the order of the *AUTHORS*-file listed persons should give a hint who has the last word when there's a total disagreement inside the team. Important stuff, like changing the licence PixelLight is released under, is suspect to the project founders working on the project since the beginning. No one else. Right now, on the PixelLight homepage at <http://www.pixellight.org> only the founders are listed to know which are the contact persons. They started the project and worked for it on their own in private for nearly a decade.

5.1 Welcome

In case you're still here reading this and have not closed the document either shocked or in total disagreement about what you've read in the introduction of this chapter... we're always totally happy to get new contributors. The PixelLight technology covers a wide area of topics and it's always wonderful to have developers willing to focus to improve certain parts of the technology. In general, it will make the organisation much easier if you concentrate on a certain part. Managing persons working on everything at one and the same time is hard to manage and also requires these persons to have a deep inside and understanding of the complete project.

5.2 Contributor License Agreement

Before we can accept contributions, you have to sign a contributor license agreement. Our contributor license agreement is derived from The Apache Software Foundation² in order to have something commonly used across open source software. In case you are not familiar with such kind of agreement we ask you to make yourself familiar with this concept so that you know what this is all about.

Individual Contributor License Agreement You'll find the *PixelLight 3D Engine Individual Contributor License Agreement ("Agreement")* online at http://pixellight.sourceforge.net/docs/PixelLight_CLA_Individual.pdf.

Software Grant and Corporate Contributor License Agreement You'll find the *PixelLight 3D Engine Software Grant and Corporate Contributor License Agreement ("Agreement")* online at http://pixellight.sourceforge.net/docs/PixelLight_CLA_Corporate.pdf.

5.3 Commit Checklist

Here are some best practices which may help to reduce errors within commits.

- Is your commit only about one single topic/task/bugfix or have you mixed up multiple stuff into one huge commit?
- Have you reviewed your commit?³
- Are you sure you haven't touched parts of the source code which had nothing to do with the topic/task/bugfix?
- Have you tested your change?
- Have you added extensive source code documentation?
- Have you respected the coding style conventions?
- Is your work seamlessly blending into the already existing source code?
- Have you added a descriptive log message, not too long, not too short?⁴
- Have you added a diary entry explaining in English what your work was about in order to make it possible for other people to follow your thoughts?

²<http://www.apache.org/licenses/>

³e.g. TortoiseGit comes with graphical diff-tools making it easy to click through made changes, this way you can also ensure that you haven't added accidentally spaces, tabs, new lines and so on

⁴Usually one or two sentences are fine

- Is your commit only about one single topic/task/bugfix or have you mixed up multiple stuff into one huge commit? Yes, this point was already there, but it's important, really.

In case you're insecure how commits should look like, please have a look into the Git log and review already made commits.

6 Contact

contact@pixellight.org

<http://www.pixellight.org>

Abbreviations

API Application Programming Interface

OS Operating System

RTTI Run-Time Type Information, or Run-Time Type Identification

MSVC Microsoft Visual C++, also known as *Visual C++* or *VC++*

GCC GNU Compiler Collection, formerly the "GNU C Compiler"

OOP Object-Oriented Programming