Coding Conventions for LLGL

Lukas Hermanns

May 13, 2018

Introduction

This is the document to describe the major C++ coding conventions for the LLGL (Low Level Graphics Library) project. While the coding conventions in programming languages such as Java and C# are globally defined and commonly acknowledged, the coding conventions in C++ vary from project to project.

If you want to contribute to this project, please comply to the coding style which is described in this document. Of course there are exceptions, but they should be well documented to avoid confusion. In the following sections there is always an example of a desired and an undesired coding style.

Syntax

Names

We start with the syntax styles in LLGL. This is similar to C#. All **classes**, **structures**, **enumerations**, **functions**, **type-aliases**, and **namespaces** begin with an upper-case letter, underscores are not allowed, and each sub name (or acronym) begins again with an upper-case letter (except for data type acronyms like ‘f’ for float):

```
// Desired
class RenderSystem
struct RenderContextDescriptor
enum class BlendOp
void Foo()
using ColorRGBf
namespace LLGL
```

```
// Undesired
class renderSystem
struct RenderContext_Descriptor
enum class blend_op
void foo()
using ColorRgbf
namespace llgl
```

All **variables** and **parameters** begin with a lower-case letter, underscores are not allowed, and each sub name (or acronym) begins again with an upper-case letter:

```
// Desired
float viewSize
ColorRGBf borderColor
Vector2f viewportSize
```

```
// Undesired
float ViewSize
ColorRGBf bordercolor
Vector2f viewport_Size
```

Moreover, the **naming convention** should be uniform as well. For a graphics object `Foo` there is almost always a descriptor structure named `FooDescriptor` for instance, and the respective parameter name is `fooDesc`:

```
// Desired
class Buffer
struct BufferDescriptor
Buffer* CreateBuffer(const BufferDescriptor& bufferDesc)
```

```
// Undesired
class Buffer
struct BufferDesc
Buffer* AllocBuffer(const BufferDesc& descriptor)
```

However, the name of a descriptor does not explicity appear in the names of inner structures:

```
// Desired
struct TextureDescriptor
{
    struct Texture1D;
    struct Texture2D;
    /* ... */
};
TextureDescriptor::Texture2D texture2DDesc;
```

```
// Undesired
struct TextureDescriptor
{
    struct Texture1DDescriptor;
    struct Texture2DDescriptor;
    /* ... */
};
TextureDescriptor::Texture2DDescriptor texture2DDesc;
```

Indentation and Braces

For **indentation** 4 spaces are used. Do not use tab characters because their size vary between the platforms and IDE settings. For all embraced code blocks a new indentation level is added, except for namespaces, because they typically always embrace the entire code file. The **braces** are written like in C#, too. An open brace has its own line (except for very small lambda functions):

```
// Desired
namespace LLGL
{

namespace InnerNamespace
{


void Func(int& i)
{
    auto verySmallLambda = [](int x) { return x*x; };
    if (i > 10)
    {
        Foo();
        Bar();
    }
    else if (i == 5)
        i = verySmallLambda(2);
}


} // /namespace InnerNamespace

} // /namespace LLGL
```

```
// Undesired
namespace LLGL {
    namespace InnerNamespace {
        void Func(int& i)
        {
            auto verySmallLambda = [](int x) {
                return x*x;
            };
            if (i > 10) { Foo();
                          Bar(); }
            else if (i == 5)
                i = verySmallLambda(2);
        }
    }
}
```

The `switch`-statement is an exception for each case block:

```
// Desired
switch (x)
{
    case 1:
        Foo();
        break;
    case 2:
        Bar();
        break;
}

switch (y)
{
    case 1:
    {
        int x = y*2;
        Foo(x);
    }
    break;

    case 2:
    {
        Bar();
    }
    break;
}
```

```
switch (z)
{
    case GL_INT:    return Types::Int;
    case GL_FLOAT:  return Types::Float;
    case GL_DOUBLE: return Types::Double;
}
```

Code Documentation

For the code documentation *doxygen* is used, i.e. the doxygen syntax is required in the commentaries of the interfaces. The following order and syntax for the doxygen commands is used:

```
// Desired
/**
\brief Does something useful.
\param[in] count Specifies the number of foos.
\param[in,out] bar Specifies the resulting bars.
\param[in] ptr Specifies an optional pointer.
\return A meaningful result.
\throw std::runtime_error If something went wrong.
\remarks This function does something useful, and returns something even more useful.
\note Only supported with: OpenGL.
\see Bar
*/
int Foo(int count, int& bar, int* ptr = nullptr);
```

to be continued . . .