1. SOFTWARE CODING STANDARD

1.1. Naming

Rule

- All identifiers in UpperCamelCase: class names, method names, function names, members of structures, namespaces, global (static) variables, etc.
- Exceptions:
  - local variable names in lowerCamelCase
  - macro names in capitals with underscores
- Private/protected members of classes: prefix the UpperCamelCase name with “m_”

Don’t use Hungarian notation like pszString, bBusy, etc.

```
#include "viewport.h"
#include <cmath>

#define SOURCE_INFO (std::to_string(__LINE__) + std::string(__FILE__))

namespace ImageGenerationControl { namespace Igc { namespace Proxylayer {

const unsigned int MaxNrButtons = 20;

MyClass::MyClass(Control& parent)
  : Control(parent),
    m_Scroll(*this)
{
    for (int i = 0; i < n; i++)
        m_Buttons.emplace_back(std::make_shared<Button>(*this));

    m_Scroll.Scroll = [this]
    {
        for (auto i : m_Buttons)
            i->Position = Point(x, y);

        if (m_Scrolled) m_Scrolled(SOURCE_INFO);
    }
}

} } }
```

Rationale: this rule is a combination of more modern naming and some legacy elements.

1.2. Private member variables

Rule: Member variables of classes are private
Rationale: OO information hiding principle

1.3. “using namespace” constraints

- Don’t use “using namespace” in header files.
- Place “using namespace” statement inside a namespace
- Don’t use “using namespace std”

Rationale: keep the (global) namespace clean

1.4. Only one private, protected and public section

Rule: Define at most one public section, same for protected and private
Rationale: Prevent confusion

1.5. Use std::function instead of a callback interface

Rule: Use std::function instead of a callback interface

Rationale: defining a std::function and doing a callback on it requires less code than defining a callback interface, deriving from it and implementing it

1.6. Use C++ types and functions instead of OS/DEVENV specific functions

Rule: use std::mutex, std::sleep_for, std::condition_variable, std::string etc. instead of Win32 API or MFC.
Rationale: C++ types/functions are platform independent and more commonly known to developers.

1.7. Forward declarations

Rule: Use forward declarations in header files
Rationale: Decrease build time dependencies; avoid circular references.

1.8. Use const keyword if possible

Rule: use the const keyword if a value, function or object is constant
Rationale: it prevents run time errors by checking compile time if functions are able to change an object

1.9. Arguments of non-primitive types

Rule: Pass arguments of non-primitive types as (const) references
Rationale: only a reference of the type will be passed saving memory and CPU resources

1.10. Static objects

Rule: Static objects are not allowed to use other static objects
Rationale: it is undefined in C++ in which order the objects defined in different files are created and will be deleted. Static/global objects should never use each other.

Note:
It is however defined that all primitive types are initialized before objects.

```
std::string s("Hello access violation");
```

```
static MyClass mc;

MyClass::MyClass()
{
   std::cout << s;
}
```

Because s and mc are both objects, and mc uses s, this will probably lead to a runtime memory access violation (if mc is created first, then std::cout is called with a argument that doesn't exist yet)
To resolve this you could use a const char* instead of a string. This will always be created first. Or you could make a function returning an std::string: *std::string s() {return "no access violation"; }*

1.11. Use preferred types inside the unit

Rule: use preferred types (often less-specific) inside the unit.
Rationale: prevent type conversions at multiple places, localization of impact of type changes of external interfaces and libraries

If libraries/interfaces use more specific types, then convert them as soon as possible near the interface.

Examples of preferred types: char, (byte), bool, int, std::string, double, and if applicable the variants of them. Also std containers are preferred.
Examples of more-specific types: schar_t, int16_t, int32_t, int64_t …

1.12. Member variable preference: objects, references and pointers

Rule:

- Use containment (full object member) if the lifetime of the object is the same as its parents lifetime
- Use a reference to an object if this reference can be injected on creation of the parent
- Use a unique_ptr in other cases
- Use only a shared_ptr if the owner of the object can't be defined

Rationale:
Simplify run-time control of objects:

- containment makes it easy: an object will exist as long as it's parent, and because of information hiding, access of the object is guaranteed within the lifetime.
- A reference has the disadvantage that another object controls the lifetime. But at least it has a value at creation of the parent, and it can be used without check to "nullptr". If no pointers would be used in code, references would be no problem. Unique_ptr is a good alternative to a reference.
- Shared_ptrs can keep each other alive. It's possible to combine them with weak_ptr but the result is more complex than the other variable types.

1.13. Order of initializer list

Rule: Order of member variables in initializer list = order in header file
Rationale: the compiler will initialize the member variables in the order of the header file, which is not intuitive for the programmer.

1.14. Lambda expression caption

Rule: Use "[this]" as default caption in lambda expressions, mention local variables explicitly
Rationale: don't use [=] because this would copy all members to a temporary object.

1.15. Resource claims/guards

Rule: Use scoped versions of resource claims/guards
Rationale: It prevents that the application hangs/crashes if the function scope is left before the resource is unlocked/freed
e.g. unique_lock<std::mutex>

1.16. nullptr

Rule: Use nullptr instread of NULL
Rationale: NULL is defined as 0. This makes it impossible for the compiler to distinguish between the types of an literal being 0 or a ptr literal being NULL. NULL can always be replaced by nullptr.

```
void f(void* ptr) { std::cout << "Pointer!"; }
void f(int i) { std::cout << "Integer!"; }

void main
{
  f(NULL);
}
```

The output will be "Integer!". If NULL is replaced by nullptr, then "Pointer!" will be displayed.

1.17. Declare variables where they are used

Rule: If possible, declare the variable in the same line as the assignment
Rationale: readability and maintainability: code becomes more compact and readable.

1.18. Keyword 'auto'

Rule: use the 'auto' keyword instead of explicitly mentioning variable type
Rationale: maintainability. Changing the type of a variable has less impact on the code.

Exception: this rule does not apply to basic types (int, bool) where using 'auto' would make the code less readable.

1.19. Tracing

Rule: Use scoped tracing for entry/exit tracing
Rule: Trace only functions that change the state of an object.

1.20. Use override keyword if a method is overridden (omit virtual keyword then)

Rule: Use override keyword if a method is overridden (omit virtual keyword then)
Rationale: this will prevent accidental overloading in case it should have been overridden.

1.21. Use the "using" keyword when overriding or "overloading" methods which have overloads in the base class,

Rule:

- C++ compiler: If you overload a method of the base class, the method in the derived class will hide all the overloads in the base class
- C++ compiler: The same rule applies for an override of a method: all overloads of the base class will be hidden
- Resolve the problem by overriding all overloads or by adding a "using" statement in the header file of the derived class

Rationale:
Overload resolution is not applied across different class scopes in C++. As a result, ambiguities between functions from base class and derived class are not resolved based on argument types. In other words, if a function is defined in a base class and overloaded in the derived, then the function in the derived class will be called if the argument can be converted, even if the base class defines a function with the correct type of argument. A compiler error will be generated If the argument(s) can't be converted or the number of arguments don't match in the derived class.
Example:

```
// Incorrect!!
class Base
{
public:
    virtual void DoSomething(int i) { std::cout << 'i'; }
};

class Derived: public Base
{
public:
    void DoSomething(char c) { std::cout << 'c'; }
};

static void f()
{
    char c = 'a';
    int i = 100;
    Derived d;

    d.DoSomething(i);     // will unexpectedly call Derived::DoSomething(char)
    d.DoSomething(c);     // will call Derived::DoSomething(char)
}
```

Calling f results in "cc"

Change the Derived class definition by adding a "using" statement:

```
class Derived: public Base
{
```

```
public:
    using Base::DoSomething;
    void DoSomething(char c) { std::cout << 'c'; }
};
```

Calling f results in “ic”

1.22. Case label ending

Rule: End the implementation of a case label with “break” or “return”
Rationale: other options will make the program flow more complex to follow.
Note: This means simple fall through to use the same implementation for multiple cases is allowed.

1.23. Arguments of sizeof

Rule: Argument within sizeof(..) shall not have side-effects.
Rationale: sizeof is a precompiler directive; it evaluates before compiling. The argument will never be evaluated run-time.

1.24. Declare method as deleted function if it must not be used

Rule: Declare method as deleted function if it must not be used
Rationale: Readability, maintainability
Restriction: no support in VS2012
Note: Most common examples are the copy constructor and assignment operator if an object is not allowed to be copied

1.25. Variable and type scoping

Rule: Declare variables and types with the smallest possible scope
Rationale: maintainability

1.26. Each file must be self-contained

Rule: Each header file must be compilable by including it in a source file which only contains the precompiled header include.
Rationale: Maintainability. Header files don’t have to be included in a specific order and the source file doesn’t have to be aware of all the dependencies of the used classes.

1.27. stl container classes

Rule: Use stl container classes instead of C-style arrays
Rationale: stl containers are more usable and safer than C-style arrays

1.28. Use enum classes instead of old style enums

Rationale: enum classes are stronger typed, do not convert implicitly to int and make forward declarations possible

2. SOFTWARE CODING GUIDELINE

2.1. Macro usage

Guideline: Avoid the use of macros
Rationale: use strongly typed variables or functions

2.2. Implementation in the header file

Guideline: Avoid implementation in the header file
Rationale: This increases build-time

2.3. Spacing and brackets / styling

Rationale: uniform code is easier to read and sometimes easier to merge

- Each statement on a separate line
- Use spaces, not tabs (just a choice, but team must make a choice)
- Use spaces
  - around keywords like if, while, …
  - around binary operators like + = == << < …
  - around : ?
  - after , ;
- Don't use spaces
  - behind open brackets ([< {
  - before close brackets)] > }
  - before , ;
  - between function-name and (
- Enhance readability
  - Use (condition == literal) in a condition, e.g. (fluo == enableEnum::enabled) is much more readable than (enableEnum::enabled == fluo)

Example:

```
#include "stdafx.h"
#include <limits>
#include <iostream>

template<class T>
typename std::enable_if<!std::numeric_limits<T>::is_integer, bool>::type
AlmostEqual(T x, T y, int ulp)
{
    // the machine epsilon has to be scaled to the magnitude of the larger value
    // and multiplied by the desired precision in ULPs (units in the last place)
    return std::abs(x - y) <= std::numeric_limits<T>::epsilon() *
                              std::max(std::abs(x), std::abs(y)) * ulp;
}

int main()
{
    std::cout << "Epsilon double: " << std::numeric_limits<double>::epsilon() << std::endl;
    // 2.22045e-016
    double d1 = 1E-200;
    double d2 = (1.0 + 1E-015) * 1E-200;

    if (d1 == d2)
        std::cout << "d1 == d2\n";
    else
        std::cout << "d1 != d2\n";

    std::cout << ((d1 < d2) ? "d1 < d2\n" : "d1 > d2\n");

    const int ulp = 6;
    if (AlmostEqual(d1, d2, ulp))
        std::cout << "d1 almost equals d2\n";
    else
```

```
        std::cout << "d1 isn't almost equal to d2\n";

    return 0;
}
```