



Open CASCADE Technology
7.3.0

Coding Rules

May 26, 2018

Contents

1	Introduction	2
1.1	Scope of the document	2
2	Naming Conventions	3
2.1	General naming rules	3
2.2	Names of development units	3
2.3	Names of variables	5
3	Formatting rules	7
4	Documentation rules	11
5	Application design	12
6	General C/C++ rules	13
7	Portability issues	15
8	Stability issues	16
9	Performance issues	19
10	Draw Harness command	20
11	Examples	22

1 Introduction

The purpose of this document is to define a common programming style for Open CASCADE Technology.

The common style facilitates understanding and maintaining a code developed cooperatively by several programmers. In addition, it enables construction of tools that incorporate knowledge of these standards to help in the programming.

OCCT programming style follows common and appropriate best practices, so some guidelines have been excerpted from the public domain.

The guide can be improved in the future as new ideas and enhancements are added.

1.1 Scope of the document

Rules in this document refer to C++ code. However, with minor exceptions due to language restrictions, they are applicable to any sources in Open CASCADE Technology framework, including:

- C/C++
- GLSL programs
- OpenCL kernels
- TCL scripts and test cases

2 Naming Conventions

2.1 General naming rules

The names considered in this section mainly refer to the interface of Open CASCADE Technology libraries or source code itself.

International language [MANDATORY]

Open CASCADE Technology is an open source platform available for an international community, thus all names need to be composed of English words or their abbreviations.

Meaningful names

Names should be meaningful or, at least, contain a meaningful part. To better understand this requirement, let us examine the existing names of toolkits, packages, classes and methods:

- Packages containing words *Geom* or *Geom2d* in their names are related to geometrical data and operations.
- Packages containing words *TopoDS* or *BRep* in their names are related to topological data and operations.
- Packages ending with *...Test* define Draw Harness plugins.
- Methods starting with *Get...* and *Set...* are usually responsible for correspondingly retrieving and storing data.

Related names

Names related to a logically connected functionality should have the same prefix (start with the same letters) or, at least, have any other common part. For example, method *GetCoord* returns a triple of real values and is defined for directions, vectors and points. The logical connection is obvious.

Camel Case style

Camel Case style is preferred for names. For example:

```
Standard_Integer awidthofbox; // this is bad
Standard_Integer width_of_box; // this is bad
Standard_Integer aWidthOfBox; // this is OK
```

2.2 Names of development units

Usually a unit (e.g. a package) is a set of classes, methods, enumerations or any other sources implementing a common functionality, which is self-contained and independent from other parts of the library.

No underscores in unit names [MANDATORY]

Names of units should not contain underscores, unless the use of underscores is allowed explicitly.

File name extensions [MANDATORY]

The following extensions should be used for source files, depending on their type:

- *.cxx* – C++ source files

- *.hxx* – C++ header files
- *.lxx* – additional headers containing definitions of inline methods and auxiliary code

Note that *.lxx* files should be avoided in most cases - inline method should be placed in header file instead.

Prefix for toolkit names [MANDATORY]

Toolkit names are prefixed by *TK*, followed by a meaningful part of the name explaining the domain of functionality covered by the toolkit (e.g. *TKOpenG*).

Names of public types

Names of public classes and other types (structures, enums, typedefs) should match the common pattern: name of the package followed by underscore and suffix (the own name of the type):

```
<package-name>_<class-name>
```

Static methods related to the whole package are defined in the class with the same name as package (without suffix).

Each type should be defined in its own header file with the name of the type and extension ".hxx". Implementation should be placed in the file with the same name and extension ".cxx"; for large classes it is possible to split implementation into multiple source files with additional suffixes in the names (usually numerical, e.g. *BSplCLib_1.cxx*).

For example, class *Adaptor2d_Curve2d* belongs to the package *Adaptor2d*; it is defined in header file *Adaptor2d_Curve2d.hxx* and implemented in source file *Adaptor2d_Curve2d.cxx*.

This rule also applies to complex types constructed by instantiation of templates. Such types should be given own names using *typedef* statement, located in same-named header file.

For example, see definition in the file *TColStd_IndexedDataMapOfStringString.hxx*:

```
typedef NCollection_IndexedDataMap<TCollection_AsciiString, TCollection_AsciiString, TCollection_AsciiString>
    TColStd_IndexedDataMapOfStringString;
```

Names of functions

The term **function** here is defined as:

- Any class method
- Any package method
- Any non-member procedure or function

It is preferred to start names of public methods from an upper case character and to start names of protected and private methods from a lower case character.

```
class MyPackage_MyClass
{
public:
    Standard_Integer Value() const;
    void SetValue (const Standard_Integer theValue);

private:
    void setIntegerValue (const Standard_Integer theValue);
};
```

2.3 Names of variables

There are several rules that describe currently accepted practices for naming variables.

Naming of variables

Name of a variable should not conflict with the existing or possible global names (for packages, macros, functions, global variables, etc.).

The name of a variable should not start with an underscore.

See the following examples:

```
Standard_Integer Elapsed_Time = 0; // this is bad - possible class name
Standard_Integer gp = 0;          // this is bad - existing package name
Standard_Integer aGp = 0;        // this is OK
Standard_Integer _KERNEL = 0;    // this is bad
Standard_Integer THE_KERNEL = 0; // this is OK
```

Names of function parameters

The name of a function (procedure, class method) parameter should start with prefix *the* followed by the meaningful part of the name starting with a capital letter.

See the following examples:

```
void Package_MyClass::MyFunction (const gp_Pnt& p);           // this is bad
void Package_MyClass::MyFunction (const gp_Pnt& theP);      // this is OK
void Package_MyClass::MyFunction (const gp_Pnt& thePoint); // this is preferred
```

Names of class member variables

The name of a class member variable should start with prefix *my* followed by the meaningful of the name starting with a capital letter.

See the following examples:

```
Standard_Integer counter; // This is bad
Standard_Integer myC;     // This is OK
Standard_Integer myCounter; // This is preferred
```

Names of global variables

It is strongly recommended to avoid defining any global variables. However, as soon as a global variable is necessary, its name should be prefixed by the name of a class or a package where it is defined followed with *_my*.

See the following examples:

```
Standard_Integer MyPackage_myGlobalVariable = 0;
Standard_Integer MyPackage_MyClass_myGlobalVariable = 0;
```

Static constants within the file should be written in upper-case and begin with prefix *THE_*:

```
namespace
{
    static const Standard_Real THE_CONSTANT_COEF = 3.14;
};
```

Names of local variables

The name of a local variable should be distinguishable from the name of a function parameter, a class member variable and a global variable.

It is preferred to prefix local variable names with *a* and *an* (or *is*, *to* and *has* for Boolean variables).

See the following example:

```
Standard_Integer theI;    // this is bad
Standard_Integer i;      // this is bad
Standard_Integer index;  // this is bad
Standard_Integer anIndex; // this is OK
```

Avoid dummy names

Avoid dummy names, such as *i*, *j*, *k*. Such names are meaningless and easy to mix up.

The code becomes more and more complicated when such dummy names are used there multiple times with different meanings, or in cycles with different iteration ranges, etc.

See the following examples for preferred style:

```
void Average (const Standard_Real** theArray,
              Standard_Integer      theRowsNb,
              Standard_Integer      theRowLen,
              Standard_Real&        theResult)
{
    theResult = 0.0;
    for (Standard_Integer aRow = 0; aRow < aRowsNb; ++aRow)
    {
        for (Standard_Integer aCol = 0; aCol < aRowLen; ++aCol)
        {
            theResult += theArray[aRow][aCol];
        }
        theResult /= Standard_Real(aRowsNb * aRowLen);
    }
}
```

3 Formatting rules

To improve the open source readability and, consequently, maintainability, the following set of rules is applied.

International language [MANDATORY]

All comments in all sources must be in English.

Line length

Try to stay within the limit of 120 characters per line in all sources.

C++ style comments

Prefer C++ style comments in C++ sources.

Commenting out unused code

Delete unused code instead of commenting it or using `#define`.

Indentation in sources [MANDATORY]

Indentation in all sources should be set to two space characters. Use of tabulation characters for indentation is disallowed.

Separating spaces

Punctuation rules follow the rules of the English language.

- C/C++ reserved words, commas, colons and semicolons should be followed by a space character if they are not at the end of a line.
- There should be no space characters after '(' and before ')'. Closing and opening brackets should be separated by a space character.
- For better readability it is also recommended to surround conventional operators by a space character. Examples:

```
while (true)                                // NOT: while( true ) ...
{
  DoSomething (theA, theB, theC, theD); // NOT: DoSomething(theA,theB,theC,theD);
}
for (anIter = 0; anIter < 10; ++anIter) // NOT: for (anIter=0;anIter<10;++anIter){
{
  theA = (theB + theC) * theD;           // NOT: theA=(theB+theC)*theD
}
```

Declaration of pointers and references

In declarations of simple pointers and references put asterisk (*) or ampersand (&) right after the type without extra space.

Since declaration of several variables with mixed pointer types contradicts this rule, it should be avoided. Instead, declare each variable independently with fully qualified type.

Examples:

```

Standard_Integer *theVariable; // not recommended
Standard_Integer * theVariable; // not recommended
Standard_Integer* theVariable; // this is OK

Standard_Integer *&theVariable; // not recommended
Standard_Integer *& theVariable; // not recommended
Standard_Integer*& theVariable; // this is OK

Standard_Integer **theVariable; // not recommended
Standard_Integer ** theVariable; // not recommended
Standard_Integer** theVariable; // this is OK

Standard_Integer *theA, theB, **theC; // not recommended (declare each variable independently)

```

Separate logical blocks

Separate logical blocks of code with one blank line and comments.

See the following example:

```

// check arguments
Standard_Integer anArgsNb = argCount();
if (anArgsNb < 3 || isSmthInvalid)
{
    return THE_ARG_INVALID;
}

// read and check header
...
...

// do our job
...
...

```

Notice that multiple blank lines should be avoided.

Separate function bodies [MANDATORY]

Use function descriptive blocks to separate function bodies from each other. Each descriptive block should contain at least a function name and purpose description.

See the following example:

```

// =====
// function : TellMeSmthGood
// purpose  : Gives me good news
// =====
void TellMeSmthGood()
{
    ...
}

// =====
// function : TellMeSmthBad
// purpose  : Gives me bad news
// =====
void TellMeSmthBad()
{
    ...
}

```

Block layout [MANDATORY]

Figure brackets `{ }` and each operator (*for*, *if*, *else*, *try*, *catch*) should be written on a dedicated line.

In general, the layout should be as follows:

```

while (expression)
{
    ...
}

```

Entering a block increases and leaving a block decreases the indentation by one tabulation.

Single-line operators

Single-line conditional operators (*if*, *while*, *for*, etc.) can be written without brackets on the following line.

```
if (!myIsInit) return Standard_False; // bad
if (thePtr == NULL)           // OK
    return Standard_False;
if (!theAlgo.IsNull())        // preferred
{
    DoSomething();
}
```

Having all code in the same line is less convenient for debugging.

Comparison expressions with constants

In comparisons, put the variable (in the current context) on the left side and constant on the right side of expression. That is, the so called "Yoda style" is to be avoided.

```
if (NULL != thePointer) // Yoda style, not recommended
if (thePointer != NULL) // OK

if (34 < anIter)        // Yoda style, not recommended
if (anIter > 34)        // OK

if (theNbValues >= anIter) // bad style (constant function argument vs. local variable)
if (anIter <= theNbValues) // OK

if (THE_LIMIT == theValue) // bad style (global constant vs. variable)
if (theValue == THE_LIMIT) // OK
```

Alignment

Use alignment wherever it enhances the readability. See the following example:

```
MyPackage_MyClass anObject;
Standard_Real     aMinimum = 0.0;
Standard_Integer  aVal      = theVal;
switch (aVal)
{
    case 0: computeSomething();           break;
    case 12: computeSomethingElse (aMinimum); break;
    case 3:
    default: computeSomethingElseYet ();   break;
}
```

Indentation of comments

Comments should be indented in the same way as the code to which they refer or they can be in the same line if they are short.

The text of the comment should be separated from the slash character by a single space character.

See the following example:

```
while (expression) //bad comment
{
    // this is a long multi-line comment
    // which is really required
    DoSomething(); // maybe, enough
    DoSomethingMore(); // again
}
```

Early return statement

Use an early return condition rather than collect indentations.

Write like this:

```
Standard_Integer ComputeSumm (const Standard_Integer* theArray,
                             const Standard_Size   theSize)
{
  Standard_Integer aSumm = 0;
  if (theArray == NULL || theSize == 0)
  {
    return 0;
  }

  ... computing summ ...
  return aSumm;
}
```

Rather than:

```
Standard_Integer ComputeSumm (const Standard_Integer* theArray,
                             const Standard_Size   theSize)
{
  Standard_Integer aSumm = 0;
  if (theArray != NULL && theSize != 0)
  {
    ... computing summ ...
  }
  return aSumm;
}
```

This helps to improve readability and reduce the unnecessary indentation depth.

Trailing spaces

Trailing spaces should be removed whenever possible. Spaces at the end of a line are useless and do not affect functionality.

Headers order

Split headers into groups: system headers, headers per each framework, project headers; sort the list of includes alphabetically. Within the class source file, the class header file should be included first.

This rule improves readability, allows detecting useless multiple header inclusions and makes 3rd-party dependencies clearly visible. Inclusion of class header on top verifies consistency of the header (e.g. that header file does not use any undefined declarations due to missing includes of dependencies).

An exception to the rule is ordering system headers generating a macros declaration conflicts (like "windows.h" or "X11/Xlib.h") - these headers should be placed in the way solving the conflict.

The source or header file should include only minimal set of headers necessary for compilation, without duplicates (considering nested includes).

```
// the header file of implemented class
#include <PackageName_ClassName.hxx>

// OCCT headers
#include <gp_Pnt.hxx>
#include <gp_Vec.hxx>
#include <NCollection_List.hxx>

// Qt headers
#include <QDataStream>
#include <QString>

// system headers
#include <iostream>
#include <windows.h>
```

4 Documentation rules

The source code is one of the most important references for documentation. The comments in the source code should be complete enough to allow understanding the corresponding code and to serve as basis for other documents.

The main reasons why the comments are regarded as documentation and should be maintained are:

- The comments are easy to reach – they are always together with the source code;
- It is easy to update a description in the comment when the source is modified;
- The source by itself is a good context to describe various details that would require much more explanations in a separate document;
- As a summary, this is the most cost-effective documentation.

The comments should be compatible with Doxygen tool for automatic documentation generation (thus should use compatible tags).

Documenting classes [MANDATORY]

Each class should be documented in its header file (.hxx). The comment should give enough details for the reader to understand the purpose of the class and the main way of work with it.

Documenting class methods [MANDATORY]

Each class or package method should be documented in the header file (.hxx).

The comment should explain the purpose of the method, its parameters, and returned value(s). Accepted style is:

```
//! Method computes the square value.  
//! @param theValue the input value  
//! @return squared value  
Standard_Export Standard_Real Square (Standard_Real theValue);
```

Documenting C/C++ sources

It is very desirable to put comments in the C/C++ sources of the package/class.

They should be detailed enough to allow any person to understand what each part of code does.

It is recommended to comment all static functions (like methods in headers), and to insert at least one comment per each 10-100 lines in the function body.

There are also some rules that define how comments should be formatted, see [Formatting Rules](#).

Following these rules is important for good comprehension of the comments. Moreover, this approach allows automatically generating user-oriented documentation directly from the commented sources.

5 Application design

The following rules define the common style, which should be applied by any developer contributing to the open source.

Allow possible inheritance

Try to design general classes (objects) keeping possible inheritance in mind. This rule means that the user who makes possible extensions of your class should not encounter problems of private implementation. Try to use protected members and virtual methods wherever you expect extensions in the future.

Avoid friend declarations

Avoid using 'friend' classes or functions except for some specific cases (for example, iteration) 'Friend' declarations increase coupling.

Set/get methods

Avoid providing set/get methods for all fields of the class. Intensive set/get functions break down encapsulation.

Hiding virtual functions [MANDATORY]

Avoid hiding a base class virtual function by a redefined function with a different signature. Most of the compilers issue warning on this.

Avoid mixing error reporting strategies

Try not to mix different error indication/handling strategies (exceptions or returned values) on the same application level.

Minimize compiler warnings [MANDATORY]

When compiling the source pay attention to and try to minimize compiler warnings.

Avoid unnecessary inclusions

Try to minimize compilation dependencies by removing unnecessary inclusions.

6 General C/C++ rules

This section defines the rules for writing a portable and maintainable C/C++ source code.

Wrapping of global variables [MANDATORY]

Use package or class methods returning reference to wrap global variables to reduce possible name space conflicts.

Avoid private members

Use *protected* members instead of *private* wherever reasonable to enable future extensions. Use *private* fields if future extensions should be disabled.

Constants and inlines over defines [MANDATORY]

Use constant variables (`const`) and inline functions instead of defines (`#define`).

Avoid explicit numerical values [MANDATORY]

Avoid usage of explicit numeric values. Use named constants and enumerations instead. Numbers produce difficulties for reading and maintenance.

Three mandatory methods

If a class has a destructor, an assignment operator or a copy constructor, it usually needs the other two methods.

Virtual destructor

A class with virtual function(s) ought to have a virtual destructor.

Overriding virtual methods

Declaration of overriding method should contains specifiers "virtual" and "override" (using `Standard_OVERRIDE` alias for compatibility with old compilers).

```
class MyPackage_BaseClass
{
public:
    Standard_EXPORT virtual Standard_Boolean Perform();
};

class MyPackage_MyClass : public MyPackage_BaseClass
{
public:
    Standard_EXPORT virtual Standard_Boolean Perform() Standard_OVERRIDE;
};
```

This makes class definition more clear (virtual methods become highlighted).

Declaration of interface using pure virtual functions protects against incomplete inheritance at first level, but does not help when method is overridden multiple times within nested inheritance or when method in base class is intended to be optional.

And here "override" specifier introduces additional protection against situations when interface changes might be missed (class might contain old methods which will be never called).

Default parameter value

Do not redefine a default parameter value in an inherited function.

Use const modifier

Use *const* modifier wherever possible (functions parameters, return values, etc.)

Usage of goto [MANDATORY]

Avoid *goto* statement unless it is really needed.

Declaring variable in for() header

Declare a cycle variable in the header of the *for()* statement if not used out of cycle.

```
Standard_Real aMinDist = Precision::Infinite();
for (NCollection_Sequence<gp_Pnt>::Iterator aPntIter (theSequence);
     aPntIter.More(); aPntIter.Next())
{
    aMinDist = Min (aMinDist, theOrigin.Distance (aPntIter.Value()));
}
```

Condition statements within zero

Avoid usage of C-style comparison for non-boolean variables:

```
void Function (Standard_Integer theValue,
              Standard_Real*   thePointer)
{
    if (!theValue)           // bad style - ambiguous logic
    {
        DoSome();
    }

    if (theValue == 0)       // OK
    {
        DoSome();
    }

    if (thePointer != NULL) // OK, predefined NULL makes pointer comparison cleaner to reader
    {
        DoSome2();         // (nullptr should be used instead as soon as C++11 will be available)
    }
}
```

7 Portability issues

This chapter contains rules that are critical for cross-platform portability.

Provide code portability [MANDATORY]

The source code must be portable to all platforms listed in the official 'Technical Requirements'. The term 'portable' here means 'able to be built from source'.

The C++ source code should meet C++03 standard. Any usage of compiler-specific features or further language versions (for example, C++11, until all major compilers on all supported platforms implement all its features) should be optional (used only with appropriate preprocessor checks) and non-exclusive (an alternative implementation compatible with other compilers should be provided).

Avoid usage of global variables [MANDATORY]

Avoid usage of global variables. Usage of global variables may cause problems when accessed from another shared library.

Use global (package or class) functions that return reference to static variable local to this function instead of global variables.

Another possible problem is the order of initialization of global variables defined in various libraries that may differ depending on platform, compiler and environment.

Avoid explicit basic types

Avoid explicit usage of basic types (*int*, *float*, *double*, etc.), use Open CASCADE Technology types from package *Standard*: *Standard_Integer*, *Standard_Real*, *Standard_ShortReal*, *Standard_Boolean*, *Standard_CString* and others or a specific *typedef* instead.

Use *sizeof()* to calculate sizes [MANDATORY]

Do not assume sizes of types. Use *sizeof()* instead to calculate sizes.

Empty line at the end of file [MANDATORY]

In accordance with C++03 standard source files should be trailed by an empty line. It is recommended to follow this rule for any plain text files for consistency and for correct work of git difference tools.

8 Stability issues

The rules listed in this chapter are important for stability of the programs that use Open CASCADE Technology libraries.

Use `OSD::SetSignal()` to catch exceptions

When using Open CASCADE Technology in an application, call `OSD::SetSignal()` function when the application is initialized.

This will install C handlers for run-time interrupt signals and exceptions, so that low-level exceptions (such as access violation, division by zero, etc.) will be redirected to C++ exceptions that use `try {...} catch (Standard_Failure) {...}` blocks.

The above rule is especially important for robustness of modeling algorithms.

Cross-referenced handles

Take care about cycling of handled references to avoid chains, which will never be freed. For this purpose, use a pointer at one (subordinate) side.

See the following example:

```
class Slave;

class Master : public Standard_Transient
{
...
void SetSlave (const Handle(Slave)& theSlave)
{
    mySlave = theSlave;
}
...
private:
    Handle(Slave) theSlave; // smart pointer
...
}

class Slave : public Standard_Transient
{
...
void SetMaster (const Handle(Master)& theMaster)
{
    myMaster = theMaster.get();
}
...
private:
    Master* theMaster; // simple pointer
...
}
```

C++ memory allocation

In C++ use `new` and `delete` operators instead of `malloc()` and `free()`. Try not to mix different memory allocation techniques.

Match `new` and `delete` [MANDATORY]

Use the same form of `new` and `delete`.

```
aPtr1 = new TypeA[n];           ... ; delete[]      aPtr1;
aPtr2 = new TypeB();           ... ; delete        aPtr2;
aPtr3 = Standard::Allocate (4096); ... ; Standard::Free (aPtr3);
```

Methods managing dynamical allocation [MANDATORY]

Define a destructor, a copy constructor and an assignment operator for classes with dynamically allocated memory.

Uninitialized variables [MANDATORY]

Every variable should be initialized.

```
Standard_Integer aTmpVar1;    // bad
Standard_Integer aTmpVar2 = 0; // OK
```

Uninitialized variables might be kept only within performance-sensitive code blocks and only when their initialization is guaranteed by subsequent code.

Do not hide global *new*

Avoid hiding the global *new* operator.

Assignment operator

In *operator=()* assign to all data members and check for assignment to self.

Float comparison

Don't check floats for equality or non-equality; check for GT, GE, LT or LE.

```
if (Abs (theFloat1 - theFloat2) < theTolerance)
{
    DoSome ();
}
```

Package *Precision* provides standard values for SI units and widely adopted by existing modeling algorithms:

- *Precision::Confusion()* for lengths in meters;
- *Precision::Angular()* for angles in radians.

as well as definition of infinite values within normal range of double precision:

- *Precision::Infinite()*
- *Precision::IsInfinite()*
- *Precision::IsPositiveInfinite()*
- *Precision::IsNegativeInfinite()*

Non-indexed iteration

Avoid usage of iteration over non-indexed collections of objects. If such iteration is used, make sure that the result of the algorithm does not depend on the order of iterated items.

Since the order of iteration is unpredictable in case of a non-indexed collection of objects, it frequently leads to different behavior of the application from one run to another, thus embarrassing the debugging process.

It mostly concerns mapped objects for which pointers are involved in calculating the hash function. For example, the hash function of *TopoDS_Shape* involves the address of *TopoDS_TShape* object. Thus the order of the same shape in the *TopTools_MapOfShape* will vary in different sessions of the application.

Do not throw in destructors

Do not throw from within a destructor.

Assigning to reference [MANDATORY]

Avoid the assignment of a temporary object to a reference. This results in a different behavior for different compilers on different platforms.

9 Performance issues

These rules define the ways of avoiding possible loss of performance caused by ineffective programming.

Class fields alignment

Declare fields of a class in the decreasing order of their size for better alignment. Generally, try to reduce misaligned accesses since they impact the performance (for example, on Intel machines).

Fields initialization order [MANDATORY]

List class data members in the constructor's initialization list in the order they are declared.

```
class MyPackage_MyClass
{
public:
    MyPackage_MyClass()
    : myPropertyA (1),
      myPropertyB (2) {}

    // NOT
    // : myPropertyB (2),
    //   myPropertyA (1) {}

private:
    Standard_Integer myPropertyA;
    Standard_Integer myPropertyB;
};
```

Initialization over assignment

Prefer initialization over assignment in class constructors.

```
MyPackage_MyClass()
: myPropertyA (1) // preferred
{
    myPropertyB = 2; // not recommended
}
```

Optimize caching

When programming procedures with extensive memory access, try to optimize them in terms of cache behavior. Here is an example of how the cache behavior can be impacted:

On x86 this code

```
Standard_Real anArray[4096][2];
for (Standard_Integer anIter = 0; anIter < 4096; ++anIter)
{
    anArray[anIter][0] = anArray[anIter][1];
}
```

is more efficient than

```
Standard_Real anArray[2][4096];
for (Standard_Integer anIter = 0; anIter < 4096; ++anIter)
{
    anArray[0][anIter] = anArray[1][anIter];
}
```

since linear access does not invalidate cache too often.

10 Draw Harness command

Draw Harness provides TCL interface for OCCT algorithms.

There is no TCL wrapper over OCCT C++ classes, instead interface is provided through the set of TCL commands implemented in C++.

There is a list of common rules which should be followed to implement well-formed Draw Harness command.

Return value

Command should return 0 in most cases even if the executed algorithm has failed. Returning 1 would lead to a TCL exception, thus should be used in case of a command line syntax error and similar issues.

Validate input parameters

Command arguments should be validated before usage. The user should see a human-readable error description instead of a runtime exception from the executed algorithm.

Validate the number of input parameters

Command should warn the user about unknown arguments, including cases when extra parameters have been pushed for the command with a fixed number of arguments.

```
if (theArgsNb != 3)
{
    std::cout << "Syntax error - wrong number of arguments!\n";
    return 1;
}

Standard_Integer anArgIter = 1;
Standard_CString aResName = theArgVec[anArgIter++];
Standard_CString aFaceName = theArgVec[anArgIter++];
TopoDS_Shape aFaceShape = DBRep::Get (aFaceName);
if (aFaceShape.IsNull()
    || aFaceShape.ShapeType() != TopAbs_FACE)
{
    std::cout << "Shape " << aFaceName << " is empty or not a Face!\n";
    return 1;
}
DBRep::Set (aResName, aFaceShape);
return 0;
```

Message printing

Informative messages should be printed into standard output `std::cout`, whilst command results (if any) – into Draw Interpreter.

Information printed into Draw Interpreter should be well-structured to allow usage in TCL script.

Long list of arguments

Any command with a long list of obligatory parameters should be considered as ill-formed by design. Optional parameters should start with flag name (with '-' prefix) and followed by its values:

```
1 myCommand -flag1 value1 value2 -flag2 value3
```

Arguments parser

- Integer values should be read using `Draw::Atoi()` function.

- Real values should be read using *Draw::Atof()* function.
- Flags names should be checked in case insensitive manner.

Functions *Draw::Atof()* and *Draw::Atoi()* support expressions and read values in C-locale.

```
Standard_Real aPosition[3] = {0.0, 0.0, 0.0};
for (Standard_Integer anArgIter = 1; anArgIter < theArgsNb; ++anArgIter)
{
    Standard_CString anArg = theArgVec[anArgIter];
    TCollection_AsciiString aFlag (anArg);
    aFlag.LowerCase(); //!  
    if (aFlag == "position")
    {
        if ((anArgIt + 3) >= theArgsNb)
        {
            std::cerr << "Wrong syntax at argument '" << anArg << "'\n";
            return 1;
        }
        aPosition[0] = Draw::Atof (theArgVec[anArgIter]);
        aPosition[1] = Draw::Atof (theArgVec[anArgIter]);
        aPosition[2] = Draw::Atof (theArgVec[anArgIter]);
    }
    else
    {
        std::cout << "Syntax error! Unknown flag '" << anArg << "'\n";
        return 1;
    }
}
```

11 Examples

Sample documented class

```

class Package_Class
{
public: ///@name public methods

    ////// Method computes the square value.
    ////// @param theValue the input value
    ////// @return squared value
    Standard_Export Standard_Real Square (const Standard_Real theValue);

private: ////// \@name private methods

    ////// Auxiliary method
    void increment();

private: ////// \@name private fields

    Standard_Integer myCounter; //////< usage counter

};

#include <Package_Class.hxx>
// =====
// function : Square
// purpose  : Method computes the square value
// =====
Standard_Real Package_Class::Square (const Standard_Real theValue)
{
    increment();
    return theValue * theValue;
}

// =====
// function : increment
// purpose  :
// =====
void Package_Class::increment ()
{
    ++myCounter;
}

```

TCL script for Draw Harness

```

1 # show fragments (solids) in shading with different colors
2 proc DisplayColored {theShape} {
3     set aSolids [uplevel #0 explode $theShape so]
4     set aColorIter 0
5     set THE_COLORS {red green blue1 magenta1 yellow cyan1 brown}
6     foreach aSolIter $aSolids {
7         uplevel #0 vdisplay          $aSolIter
8         uplevel #0 vsetcolor         $aSolIter [lindex $THE_COLORS [expr [incr aColorIter] % [llength
9             $THE_COLORS]]]
10        uplevel #0 vsetdispmode      $aSolIter 1
11        uplevel #0 vsetmaterial      $aSolIter plastic
12        uplevel #0 vsettransparency  $aSolIter 0.5
13    }
14 }
15 # load modules
16 pload MODELING VISUALIZATION
17
18 # create boxes
19 box bc 0 0 0 1 1 1
20 box br 1 0 0 1 1 2
21 compound bc br c
22
23 # show fragments (solids) in shading with different colors
24 vinit View1
25 vclear
26 vaxo
27 vzbufftrihedron
28 DisplayColored c
29 vfit
30 vdump $imagedir/${casename}.png 512 512

```

GLSL program:

```
vec3 Ambient; //!< Ambient contribution of light sources
vec3 Diffuse; //!< Diffuse contribution of light sources
vec3 Specular; //!< Specular contribution of light sources

//! Computes illumination from light sources
vec4 ComputeLighting (in vec3 theNormal,
                    in vec3 theView,
                    in vec4 thePoint)
{
    // clear the light intensity accumulators
    Ambient = occLightAmbient.rgb;
    Diffuse = vec3 (0.0);
    Specular = vec3 (0.0);
    vec3 aPoint = thePoint.xyz / thePoint.w;
    for (int anIndex = 0; anIndex < occLightSourcesCount; ++anIndex)
    {
        int aType = occLight_Type (anIndex);
        if (aType == OccLightType_Direct)
        {
            directionalLight (anIndex, theNormal, theView);
        }
        else if (aType == OccLightType_Point)
        {
            pointLight (anIndex, theNormal, theView, aPoint);
        }
    }

    return vec4 (Ambient, 1.0) * occFrontMaterial_Ambient()
        + vec4 (Diffuse, 1.0) * occFrontMaterial_Diffuse()
        + vec4 (Specular, 1.0) * occFrontMaterial_Specular();
}

//! Entry point to the Fragment Shader
void main()
{
    gl_FragColor = computeLighting (normalize (Normal),
                                    normalize (View),
                                    Position);
}
```