Open CASCADE Technology
7.7.0

Automated Testing System

November 2, 2022

Contents

1 Introduction

This document provides OCCT developers and contributors with an overview and practical guidelines for work with OCCT automatic testing system.

Reading the Introduction should be sufficient for developers to use the test system to control non-regression of the modifications they implement in OCCT. Other sections provide a more in-depth description of the test system, required for modifying the tests and adding new test cases.

1.1 Basic Information

OCCT automatic testing system is organized around DRAW Test Harness, a console application based on Tcl (a scripting language) interpreter extended by OCCT-related commands.

Standard OCCT tests are included with OCCT sources and are located in subdirectory *tests* of the OCCT root folder. Other test folders can be included in the test system, e.g. for testing applications based on OCCT.

The tests are organized in three levels:

- Group: a group of related test grids, usually testing a particular OCCT functionality (e.g. blend);
- Grid: a set of test cases within a group, usually aimed at testing some particular aspect or mode of execution of the relevant functionality (e.g. buildevol);
- Test case: a script implementing an individual test (e.g. K4).

See Test Groups chapter for the current list of available test groups and grids.

Note

Many tests involve data files (typically CAD models) which are located separately and (except a few) are not included with OCCT code. These tests will be skipped if data files are not available.

1.2 Intended Use of Automatic Tests

Each modification made in OCCT code must be checked for non-regression by running the whole set of tests. The developer who makes the modification is responsible for running and ensuring non-regression for the tests available to him.

Note that many tests are based on data files that are confidential and thus available only at OPEN CASCADE. The official certification testing of each change before its integration to master branch of official OCCT Git repository (and finally to the official release) is performed by OPEN CASCADE to ensure non-regression on all existing test cases and supported platforms.

Each new non-trivial modification (improvement, bug fix, new feature) in OCCT should be accompanied by a relevant test case suitable for verifying that modification. This test case is to be added by the developer who provides the modification.

If a modification affects the result of an existing test case, either the modification should be corrected (if it causes regression) or the affected test cases should be updated to account for the modification.

The modifications made in the OCCT code and related test scripts should be included in the same integration to the master branch.

1.3 Quick Start

1.3.1 Setup

Before running tests, make sure to define environment variable *CSF_TestDataPath* pointing to the directory containing test data files.

For this it is recommended to add a file *DrawAppliInit* in the directory which is current at the moment of starting DRAWEXE (normally it is OCCT root directory, *$CASROOT*). This file is evaluated automatically at the DRAW start.

Example (Windows)

```
set env(CSF_TestDataPath) $env(CSF_TestDataPath)\;d:/occt/test-data
```

Note that variable *CSF_TestDataPath* is set to default value at DRAW start, pointing at the folder *$CASROOT/data*. In this example, subdirectory *d:/occt/test-data* is added to this path. Similar code could be used on Linux and Mac OS X except that on non-Windows platforms colon ":" should be used as path separator instead of semicolon ";".

All tests are run from DRAW command prompt (run *draw.bat* or *draw.sh* to start it).

1.3.2 Running Tests

To run all tests, type command *testgrid*

Example:

```
Draw[]> testgrid
```

To run only a subset of test cases, give masks for group, grid, and test case names to be executed. Each argument is a list of file masks separated with commas or spaces; by default "*" is assumed.

Example:

```
Draw[]> testgrid bugs caf,moddata*,xde
```

As the tests progress, the result of each test case is reported. At the end of the log a summary of test cases is output, including the list of detected regressions and improvements, if any.

Example:

```
Tests summary

CASE 3rdparty export A1: OK
...
CASE pipe standard B1: BAD (known problem)
CASE pipe standard C1: OK
No regressions
Total cases: 208 BAD, 31 SKIPPED, 3 IMPROVEMENT, 1791 OK
Elapsed time: 1 Hours 14 Minutes 33.7384512019 Seconds
Detailed logs are saved in D:/occt/results_2012-06-04T0919
```

The tests are considered as non-regressive if only OK, BAD (i.e. known problem), and SKIPPED (i.e. not executed, typically because of lack of a data file) statuses are reported. See Interpretation of test results for details.

The results and detailed logs of the tests are saved by default to a new subdirectory of the subdirectory *results* in the current folder, whose name is generated automatically using the current date and time, prefixed by Git branch name (if Git is available and current sources are managed by Git). If necessary, a non-default output directory can be specified using option *-outdir* followed by a path to the directory. This directory should be new or empty; use option *-overwrite* to allow writing results in the existing non-empty directory.

Example:

```
Draw[]> testgrid -outdir d:/occt/last_results -overwrite
```

In the output directory, a cumulative HTML report *summary.html* provides links to reports on each test case. An additional report in JUnit-style XML format can be output for use in Jenkins or other continuous integration system.

To re-run the test cases, which were detected as regressions on the previous run, option *-regress dirname* should be used. *dirname* is a path to the directory containing the results of the previous run. Only the test cases with *FAILED* and *IMPROVEMENT* statuses will be tested.

Example:

```
Draw[]> testgrid -regress d:/occt/last_results
```

Type *help testgrid* in DRAW prompt to get help on options supported by *testgrid* command:

```
Draw[3]> help testgrid
testgrid: Run all tests, or specified group, or one grid
    Use: testgrid [groupmask [gridmask [casemask]]] [options...]
    Allowed options are:
    -parallel N: run N parallel processes (default is number of CPUs, 0 to disable)
    -refresh N: save summary logs every N seconds (default 60, minimal 1, 0 to disable)
    -outdir dirname: set log directory (should be empty or non-existing)
    -overwrite: force writing logs in existing non-empty directory
    -xml filename: write XML report for Jenkins (in JUnit-like format)
    -beep: play sound signal at the end of the tests
    -regress dirname: re-run only a set of tests that have been detected as regressions on the previous run
      .
                      Here "dirname" is a path to the directory containing the results of the previous run.
    Groups, grids, and test cases to be executed can be specified by the list of file
    masks separated by spaces or commas; default is all (*).
```

1.3.3 Running a Single Test

To run a single test, type command *test* followed by names of group, grid, and test case.

Example:

```
Draw[1]> test blend simple A1
CASE blend simple A1: OK
Draw[2]>
```

Note that normally an intermediate output of the script is not shown. The detailed log of the test can be obtained after the test execution by running command *"dlog get"*.

To see intermediate commands and their output during the test execution, add one more argument *"echo"* at the end of the command line. Note that with this option the log is not collected and summary is not produced.

Type *help test* in DRAW prompt to get help on options supported by *test* command:

```
Draw[3]> help test
test: Run specified test case
 Use: test group grid casename [options...]
 Allowed options are:
 -echo: all commands and results are echoed immediately,
        but log is not saved and summary is not produced
        It is also possible to use "1" instead of "-echo"
        If echo is OFF, log is stored in memory and only summary
        is output (the log can be obtained with command "dlog get")
 -outfile filename: set log file (should be non-existing),
        it is possible to save log file in text file or
        in html file(with snapshot), for that "filename"
        should have ".html" extension
 -overwrite: force writing log in existing file
 -beep: play sound signal at the end of the test
 -errors: show all lines from the log report that are recognized as errors
        This key will be ignored if the "-echo" key is already set.
```

1.3.4 Creating a New Test

The detailed rules of creation of new tests are given in Creation and modification of tests chapter. The following short description covers the most typical situations:

Use prefix *bug* followed by Mantis issue ID and, if necessary, additional suffixes, for naming the test script, data files, and DRAW commands specific for this test case.

1. If the test requires C++ code, add it as new DRAW command(s) in one of files in *QABugs* package.
2. Add script(s) for the test case in the subfolder corresponding to the relevant OCCT module of the group *bugs (*$CASROOT/tests/bugs*)*. See the correspondence map.
3. In the test script:

- Load all necessary DRAW modules by command *pload*.
- Use command *locate_data_file* to get a path to data files used by test script. (Make sure to have this command not inside catch statement if it is used.)
- Use DRAW commands to reproduce the tested situation.
- Make sure that in case of failure the test produces a message containing word "Error" or other recognized by the test system as error (add new error patterns in file parse.rules if necessary).
- If the test case reports error due to an existing problem and the fix is not available, add TODO statement for each error to mark it as a known problem. The TODO statements must be specific so as to match the actually generated messages but not all similar errors.
- To check expected output which should be obtained as the test result, add REQUIRED statement for each line of output to mark it as required.
- If the test case produces error messages (contained in parse.rules), which are expected in that test and should not be considered as its failure (e.g. test for *checkshape* command), add REQUIRED statement for each error to mark it as required output.

4. To check whether the data files needed for the test are already present in the database, use DRAW command *testfile* (see below). If the data file is already present, use it for a new test instead of adding a duplicate. If the data file(s) are not yet present in the test database, put them to a folder and add it to the environment variable *CSF_TestDataPath* to be found by the test system. The location of the data files, which need to be accessed by OCC team and put to the official database, should be provided in the comment to Mantis issue, clearly indicating how the names of the files used by the test script match the actual names of the files. The simplest way is to attach the data files to the Mantis issue, with the same names as used by the test script.
5. Check that the test case runs as expected (test for fix: OK with the fix, FAILED without the fix; test for existing problem: BAD), and integrate it to the Git branch created for the issue.

Example:

- Added files:

```
git status -short
A tests/bugs/heal/data/bug210_a.brep
A tests/bugs/heal/data/bug210_b.brep
A tests/bugs/heal/bug210_1
A tests/bugs/heal/bug210_2
```

- Test script

```
puts "OCC210 (case 1): Improve FixShape for touching wires"

restore [locate_data_file bug210_a.brep] a

fixshape result a 0.01 0.01
checkshape result
```

DRAW command *testfile* should be used to check the data files used by the test for possible duplication of content or names. The command accepts the list of paths to files to be checked (as a single argument) and gives a conclusion on each of the files, for instance:

```
Draw[1]> testfile [glob /my/data/path/bug12345*]
Collecting info on test data files repository...
Checking new file(s)...

* /my/data/path/bug12345.brep: duplicate
  already present under name bug28773_1.brep
  --> //server/occt_tests_data/public/brep/bug28773_1.brep

* /my/data/path/cadso.brep: new file
  Warning: DOS encoding detected, consider converting to
           UNIX unless DOS line ends are needed for the test
  Warning: shape contains triangulation (946 triangles),
           consider removing them unless they are needed for the test!
  BREP size=201 KiB, nbfaces=33, nbedges=94 -> private
```

```
* /my/data/path/case_8_wire3.brep: already present
  --> //server/occt_tests_data/public/brep/case_8_wire3.brep

* /my/data/path/case_8_wire4.brep: error
  name is already used by existing file
  --> //server/occt_tests_data/public/brep/case_8_wire4.brep
```

2 Organization of Test Scripts

2.1 General Layout

Standard OCCT tests are located in subdirectory tests of the OCCT root folder ($CASROOT).

Additional test folders can be added to the test system by defining environment variable *CSF_TestScriptsPath*. This should be list of paths separated by semicolons (∗;∗) on Windows or colons (∗:∗) on Linux or Mac. Upon DRAW launch, path to *tests* subfolder of OCCT is added at the end of this variable automatically.

Each test folder is expected to contain:

- Optional file *parse.rules* defining patterns for interpretation of test results, common for all groups in this folder
- One or several test group directories.

Each group directory contains:

- File *grids.list* that identifies this test group and defines list of test grids in it.
- Test grids (sub-directories), each containing set of scripts for test cases, and optional files *cases.list*, *parse.rules*, *begin* and *end*.
- Optional sub-directory data

By convention, names of test groups, grids, and cases should contain no spaces and be lower-case. The names *begin, end, data, parse.rules, grids.list* and *cases.list* are reserved.

General layout of test scripts is shown in Figure 1.

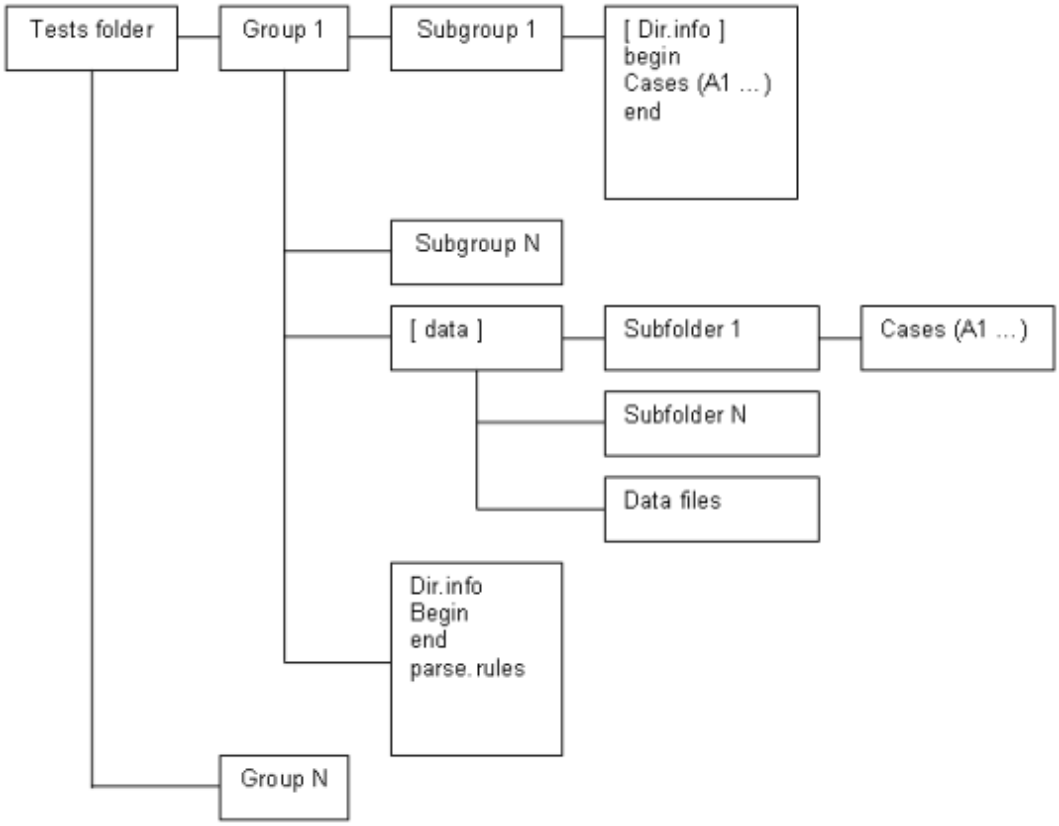

Figure 1: Layout of tests folder

2.2 Test Groups

2.2.1 Group Names

The names of directories of test groups containing systematic test grids correspond to the functionality tested by each group.

Example:

```
caf
mesh
offset
```

Test group *bugs* is used to collect the tests coming from bug reports. Group *demo* collects tests of the test system, DRAW, samples, etc.

2.2.2 File "grids.list"

This test group contains file *grids.list*, which defines an ordered list of grids in this group in the following format:

```
001 gridname1
002 gridname2
...
NNN gridnameN
```

Example:

```
001 basic
002 advanced
```

2.2.3 File "begin"

This file is a Tcl script. It is executed before every test in the current group. Usually it loads necessary Draw commands, sets common parameters and defines additional Tcl functions used in test scripts.

Example:

```
pload TOPTEST ;# load topological command
set cpulimit 300 ;# set maximum time allowed for script execution
```

2.2.4 File "end"

This file is a TCL script. It is executed after every test in the current group. Usually it checks the results of script work, makes a snap-shot of the viewer and writes *TEST COMPLETED* to the output.

Note: *TEST COMPLETED* string should be present in the output to indicate that the test is finished without crash.

See Creation and modification of tests chapter for more information.

Example:

```
if { [isdraw result] } {
    checkshape result
} else {
    puts "Error: The result shape can not be built"
}
puts "TEST COMPLETED"
```

2.2.5 File "parse.rules"

The test group may contain *parse.rules* file. This file defines patterns used for analysis of the test execution log and deciding the status of the test run.

Each line in the file should specify a status (single word), followed by a regular expression delimited by slashes (∗/∗) that will be matched against lines in the test output log to check if it corresponds to this status.

The regular expressions should follow Tcl syntax, with a special exception that "\b" is considered as word limit (Perl-style), in addition to "\y" used in Tcl.

The rest of the line can contain a comment message, which will be added to the test report when this status is detected.

Example:

```
FAILED /\b[Ee]xception\b/ exception
FAILED /\bError\b/ error
SKIPPED /Cannot open file for reading/ data file is missing
SKIPPED /Could not read file .*, abandon/ data file is missing
```

Lines starting with a ∗#∗ character and blank lines are ignored to allow comments and spacing.

See Interpretation of test results chapter for details.

If a line matches several rules, the first one applies. Rules defined in the grid are checked first, then rules in the group, then rules in the test root directory. This allows defining some rules on the grid level with status *IGNORE* to ignore messages that would otherwise be treated as errors due to the group level rules.

Example:

```
FAILED /\\bFaulty\\b/ bad shape
IGNORE /^Error [23]d = [\d.-]+/ debug output of blend command
IGNORE /^Tcl Exception: tolerance ang : [\d.-]+/ blend failure
```

2.2.6 Directory "data"

The test group may contain subdirectory *data*, where test scripts shared by different test grids can be put. See also Directory data.

2.3 Test Grids

2.3.1 Grid Names

The folder of a test group can have several sub-directories (Grid 1... Grid N) defining test grids. Each directory contains a set of related test cases. The name of a directory should correspond to its contents.

Example:

```
caf
   basic
   bugs
   presentation
```

Here *caf* is the name of the test group and *basic*, *bugs*, *presentation*, etc. are the names of grids.

2.3.2 File "begin"

This file is a TCL script executed before every test in the current grid.

Usually it sets variables specific for the current grid.

Example:

```
set command bopfuse ;# command tested in this grid
```

2.3.3 File "end"

This file is a TCL script executed after every test in current grid.

Usually it executes a specific sequence of commands common for all tests in the grid.

Example:

```
vdump $imagedir/${casename}.png ;# makes a snap-shot of AIS viewer
```

2.3.4 File "cases.list"

The grid directory can contain an optional file cases.list defining an alternative location of the test cases. This file should contain a single line defining the relative path to the collection of test cases.

Example:

```
../data/simple
```

This option is used for creation of several grids of tests with the same data files and operations but performed with differing parameters. The common scripts are usually located place in the common subdirectory of the test group, *data/simple* for example.

If file *cases.list* exists, the grid directory should not contain any test cases. The specific parameters and pre- and post-processing commands for test execution in this grid should be defined in the files *begin* and *end*.

2.3.5 Directory "data"

The test grid may contain subdirectory *data*, containing data files used in tests (BREP, IGES, STEP, etc.) of this grid.

2.4 Test Cases

The test case is a TCL script, which performs some operations using DRAW commands and produces meaningful messages that can be used to check the validity of the result.

Example:

```
pcylinder c1 10 20 ;# create first cylinder
pcylinder c2 5 20 ;# create second cylinder
ttranslate c2 5 0 10 ;# translate second cylinder to x,y,z
bsection result c1 c2 ;# create a section of two cylinders
checksection result ;# will output error message if result is bad
```

The test case can have any name (except for the reserved names *begin, end, data, cases.list* and *parse.rules*). For systematic grids it is usually a capital English letter followed by a number.

Example:

```
A1
A2
B1
B2
```

Such naming facilitates compact representation of tests execution results in tabular format within HTML reports.

3 Creation And Modification Of Tests

This section describes how to add new tests and update existing ones.

3.1 Choosing Group, Grid, and Test Case Name

The new tests are usually added in the frame of processing issues in OCCT Mantis tracker. Such tests in general should be added to group bugs, in the grid corresponding to the affected OCCT functionality. See Mapping of OCCT functionality to grid names in group bugs.

New grids can be added as necessary to contain tests for the functionality not yet covered by existing test grids. The test case name in the bugs group should be prefixed by the ID of the corresponding issue in Mantis (without leading zeroes) with prefix *bug*. It is recommended to add a suffix providing a hint on the tested situation. If more than one test is added for a bug, they should be distinguished by suffixes; either meaningful or just ordinal numbers.

Example:

```
bug12345_coaxial
bug12345_orthogonal_1
bug12345_orthogonal_2
```

If the new test corresponds to a functionality already covered by the existing systematic test grid (e.g. group *mesh* for *BRepMesh* issues), this test can be added (or moved later by OCC team) to that grid.

3.2 Adding Data Files Required for a Test

It is advisable to make self-contained test scripts whenever possible, so as they could be used in the environments where data files are not available. For that simple geometric objects and shapes can be created using DRAW commands in the test script itself.

If the test requires a data file, it should be put to the directory listed in environment variable *CSF_TestDataPath*. Alternatively, it can be put to subdirectory *data* of the test grid. It is recommended to prefix the data file with the corresponding issue id prefixed by *bug*, e.g. *bug12345_face1.brep*, to avoid possible conflicts with names of existing data files.

Note that when the test is integrated to the master branch, OCC team will move the data file to the data files repository, to keep OCCT sources repository clean from data files.

When you prepare a test script, try to minimize the size of involved data model. For instance, if the problem detected on a big shape can be reproduced on a single face extracted from that shape, use only that face in the test.

3.3 Adding new DRAW commands

If the test cannot be implemented using available DRAW commands, consider the following possibilities:

- If the existing DRAW command can be extended to enable possibility required for a test in a natural way (e.g. by adding an option to activate a specific mode of the algorithm), this way is recommended. This change should be appropriately documented in a relevant Mantis issue.
- If the new command is needed to access OCCT functionality not exposed to DRAW previously, and this command can be potentially reused (for other tests), it should be added to the package where similar commands are implemented (use *getsource* DRAW command to get the package name). The name and arguments of the new command should be chosen to keep similarity with the existing commands. This change should be documented in a relevant Mantis issue.
- Otherwise the new command implementing the actions needed for this particular test should be added in *QABugs* package. The command name should be formed by the Mantis issue ID prefixed by *bug*, e.g. *bug12345*.

Note that a DRAW command is expected to return 0 in case of a normal completion, and 1 (Tcl exception) if it is incorrectly used (e.g. a wrong number of input arguments). Thus if the new command needs to report a test error, this should be done by outputting an appropriate error message rather than by returning a non-zero value. File names must be encoded in the script rather than in the DRAW command and passed to the DRAW command as an argument.

3.4 Script Implementation

The test should run commands necessary to perform the tested operations, in general assuming a clean DRAW session. The required DRAW modules should be loaded by *pload* command, if it is not done by *begin* script. The messages produced by commands in a standard output should include identifiable messages on the discovered problems if any.

Usually the script represents a set of commands that a person would run interactively to perform the operation and see its results, with additional comments to explain what happens.

Example:

```
# Simple test of fusing box and sphere
box b 10 10 10
sphere s 5
bfuse result b s
checkshape result
```

Make sure that file *parse.rules* in the grid or group directory contains a regular expression to catch possible messages indicating the failure of the test.

For instance, for catching errors reported by *checkshape* command relevant grids define a rule to recognize its report by the word *Faulty*:

```
FAILED /\bFaulty\b/ bad shape
```

For the messages generated in the script it is recommended to use the word 'Error' in the error message.

Example:

```
set expected_length 11
if { [expr $actual_length - $expected_length] > 0.001 } {
    puts "Error: The length of the edge should be $expected_length"
}
```

At the end, the test script should output *TEST COMPLETED* string to mark a successful completion of the script. This is often done by the *end* script in the grid.

During execution of a test, the following Tcl variables are defined on global level:

Variable	Value
dirname	Path to the root directory of the current set of test scripts
groupname	Name of the test group (subfolder of $dirname)
gridname	Name of the test grid (subfolder of $dirname/$gridname)
casename	Name of the test
imagedir	Path to folder where test log and other artifacts are saved

The test script can use some data stored in a separate file (e.g. reference results of the test execution). Such file can be put in subfolder *data* of the test grid directory. During execution of the test, location of such data file can be constructed using the variables listed above.

Example:

```
checkresult $result $::dirname/$::groupname/$::gridname/data/${::casename}.txt
```

CAD models and other data files which are not going to change over time should be stored separately from the source repository. Use Tcl procedure *locate_data_file* to get a path to such data files, instead of coding the path explicitly. For the file to be found by that procedure, add directory that contains it into the environment variable *CSF_TestDataPath* (list of paths separated by semicolons on Windows or colons on other platforms). The search is recursive, thus adding only root folder of a directory containing data files is sufficient. If the file is not found, *locate_data_file* will raise exception, and the test will be reported as SKIPPED.

Example:

```
stepread [locate_data_file CAROSKI_COUPELLE.step] a *
```

When the test needs to produce some snapshots or other artefacts, use Tcl variable *imagedir* as the location where such files should be put.

- Command *testgrid* sets this variable to the subdirectory of the results folder corresponding to the grid.
- Command *test* by default creates a dedicated temporary directory in the system temporary folder (normally the one specified by environment variable *TempDir*, *TEMP*, or *TMP*) for each execution, and sets *imagedir* to that location.

However if variable *imagedir* is defined on the top level of Tcl interpreter, command *test* will use it instead of creating a new directory.

Use Tcl variable *casename* to prefix all files produced by the test. This variable is set to the name of the test case.

The test system can recognize an image file (snapshot) and include it in HTML log and differences if its name starts with the name of the test case (use variable *casename*), optionally followed by underscore or dash and arbitrary suffix.

The image format (defined by extension) should be *png*.

Example:

```
xwd $::imagedir/${::casename}.png
vdisplay result; vfit
vdump $::imagedir/${::casename}-axo.png
vfront; vfit
vdump $::imagedir/${::casename}-front.png
```

would produce:

```
A1.png
A1-axo.png
A1-front.png
```

Note that OCCT must be built with FreeImage support to be able to produce usable images.

In order to ensure that the test works as expected in different environments, observe the following additional rules:

- Avoid using external commands such as *grep, rm,* etc., as these commands can be absent on another system (e.g. on Windows); use facilities provided by Tcl instead.
- Do not put call to *locate_data_file* in catch statement – this can prevent correct interpretation of the missing data file by the test system.
- Do not use commands *decho* and *dlog* in the test script, to avoid interference with use of these commands by the test system.

3.5 Interpretation of test results

The result of the test is evaluated by checking its output against patterns defined in the files *parse.rules* of the grid and group.

The OCCT test system recognizes five statuses of the test execution:

- SKIPPED: reported if a line matching SKIPPED pattern is found (prior to any FAILED pattern). This indicates that the test cannot be run in the current environment; the most typical case is the absence of the required data file.
- FAILED: reported if a line matching pattern with status FAILED is found (unless it is masked by the preceding IGNORE pattern or a TODO or REQUIRED statement), or if message TEST COMPLETED or at least one of REQUIRED patterns is not found. This indicates that the test has produced a bad or unexpected result, and usually means a regression.
- BAD: reported if the test script output contains one or several TODO statements and the corresponding number of matching lines in the log. This indicates a known problem. The lines matching TODO statements are not checked against other patterns and thus will not cause a FAILED status.
- IMPROVEMENT: reported if the test script output contains a TODO statement for which no corresponding line is found. This is a possible indication of improvement (a known problem has disappeared).
- OK: reported if none of the above statuses have been assigned. This means that the test has passed without problems.

Other statuses can be specified in *parse.rules* files, these will be classified as FAILED.

For integration of the change to OCCT repository, all tests should return either OK or BAD status. The new test created for an unsolved problem should return BAD. The new test created for a fixed problem should return FAILED without the fix, and OK with the fix.

3.6 Marking BAD cases

If the test produces an invalid result at a certain moment then the corresponding bug should be created in the OCCT issue tracker located at https://tracker.dev.opencascade.org, and the problem should be marked as TODO in the test script.

The following statement should be added to such a test script:

```
puts "TODO BugNumber ListOfPlatforms: RegularExpression"
```

Here:

- *BugNumber* is the bug ID in the tracker. For example: #12345.
- *ListOfPlatforms* is a list of platforms, at which the bug is reproduced (Linux, Windows, MacOS, or All). Note that the platform name is custom for the OCCT test system; Use procedure *checkplatform* to get the platform name.

Example:

```
Draw[2]> checkplatform
Windows
```

- RegularExpression is a regular expression, which should be matched against the line indicating the problem in the script output.

Example:

```
puts "TODO #22622 Mandriva2008: Abort .* an exception was raised"
```

The parser checks the test output and if an output line matches the *RegularExpression* then it will be assigned a BAD status instead of FAILED.

A separate TODO line must be added for each output line matching an error expression to mark the test as BAD. If not all TODO messages are found in the test log, the test will be considered as possible improvement.

To mark the test as BAD for an incomplete case (when the final *TEST COMPLETE* message is missing) the expression *TEST INCOMPLETE* should be used instead of the regular expression.

Example:

```
puts "TODO OCC22817 All: exception.+There are no suitable edges"
puts "TODO OCC22817 All: \\*\\* Exception \\*\\*"
puts "TODO OCC22817 All: TEST INCOMPLETE"
```

3.7 Marking required output

To check the obtained test output matches the expected results considered correct, add REQUIRED statement for each specific message. For that, the following statement should be added to the corresponding test script:

```
puts "REQUIRED ListOfPlatforms: RegularExpression"
```

Here *ListOfPlatforms* and *RegularExpression* have the same meaning as in TODO statements described above.

The REQUIRED statement can also be used to mask the message that would normally be interpreted as error (according to the rules defined in *parse.rules*) but should not be considered as such within the current test.

Example:

```
puts "REQUIRED Linux: Faulty shapes in variables faulty_1 to faulty_5"
```

This statement notifies test system that errors reported by *checkshape* command are expected in that test case, and test should be considered as OK if this message appears, despite of presence of general rule stating that 'Faulty' signals failure.

If output does not contain required statement, test case will be marked as FAILED.

4 Advanced Use

4.1 Running Tests on Older Versions of OCCT

Sometimes it might be necessary to run tests on the previous versions of OCCT (<= 6.5.4) that do not include this test system. This can be done by adding DRAW configuration file *DrawAppliInit* in the directory, which is current by the moment of DRAW start-up, to load test commands and to define the necessary environment.

Note: in OCCT 6.5.3, file *DrawAppliInit* already exists in *$CASROOT/src/DrawResources*, new commands should be added to this file instead of a new one in the current directory.

For example, let us assume that *d:/occt* contains an up-to-date version of OCCT sources with tests, and the test data archive is unpacked to *d:/test-data*):

```
set env(CASROOT) d:/occt
set env(CSF_TestScriptsPath) $env(CASROOT)/tests
source $env(CASROOT)/src/DrawResources/TestCommands.tcl
set env(CSF_TestDataPath) $env(CASROOT)/data;d:/test-data
return
```

Note that on older versions of OCCT the tests are run in compatibility mode and thus not all output of the test command can be captured; this can lead to absence of some error messages (can be reported as either a failure or an improvement).

4.2 Adding custom tests

You can extend the test system by adding your own tests. For that it is necessary to add paths to the directory where these tests are located, and one or more additional data directories, to the environment variables *CSF_TestScriptsPath* and *CSF_TestDataPath*. The recommended way for doing this is using DRAW configuration file *DrawAppliInit* located in the directory which is current by the moment of DRAW start-up.

Use Tcl command *_path_separator* to insert a platform-dependent separator to the path list.

For example:

```
set env(CSF_TestScriptsPath) \
  $env(TestScriptsPath)[_path_separator]d:/MyOCCTProject/tests
set env(CSF_TestDataPath) \
  d:/occt/test-data[_path_separator]d:/MyOCCTProject/data
return ;# this is to avoid an echo of the last command above in cout
```

4.3 Parallel execution of tests

For better efficiency, on computers with multiple CPUs the tests can be run in parallel mode. This is default behavior for command *testgrid* : the tests are executed in parallel processes (their number is equal to the number of CPUs available on the system). In order to change this behavior, use option parallel followed by the number of processes to be used (1 or 0 to run sequentially).

Note that the parallel execution is only possible if Tcl extension package *Thread* is installed. If this package is not available, *testgrid* command will output a warning message.

4.4 Checking non-regression of performance, memory, and visualization

Some test results are very dependent on the characteristics of the workstation, where they are performed, and thus cannot be checked by comparison with some predefined values. These results can be checked for non-regression (after a change in OCCT code) by comparing them with the results produced by the version without this change. The most typical case is comparing the result obtained in a branch created for integration of a fix (CR***) with the results obtained on the master branch before that change is made.

OCCT test system provides a dedicated command *testdiff* for comparing CPU time of execution, memory usage, and images produced by the tests.

```
testdiff dir1 dir2 [groupname [gridname]] [options...]
```

Here *dir1* and *dir2* are directories containing logs of two test runs.

Possible options are:

- *-save <filename>* – saves the resulting log in a specified file (*$dir1/diff-$dir2.log* by default). HTML log is saved with the same name and extension .html;
- *-status {same|ok|all}* – allows filtering compared cases by their status:
  - *same* – only cases with same status are compared (default);
  - *ok* – only cases with OK status in both logs are compared;
  - *all* – results are compared regardless of status;
- *-verbose <level>* – defines the scope of output data:
  - 1 – outputs only differences;
  - 2 – additionally outputs the list of logs and directories present in one of directories only;
  - 3 – (by default) additionally outputs progress messages;
- *-image [filename]* - compare images and save the resulting log in specified file (*$dir1/diffimage-$dir2.log* by default)
- *-cpu [filename]* - compare overall CPU and save the resulting log in specified file (*$dir1/diffcpu-$dir2.log* by default)
- *-memory [filename]* - compare memory delta and save the resulting log in specified file (*$dir1/diffmemory-$dir2.log* by default)
- *-highlight_percent <value>* - highlight considerable (>value in %) deviations of CPU and memory (default value is 5%)

Example:

```
Draw[]> testdiff results/CR12345-2012-10-10T08:00 results/master-2012-10-09T21:20
```

Particular tests can generate additional data that need to be compared by *testdiff* command. For that, for each parameter to be controlled, the test should produce the line containing keyword "COUNTER∗ followed by arbitrary name of the parameter, then colon and numeric value of the parameter.

Example of test code:

```
puts "COUNTER Memory heap usage at step 5: [meminfo h]"
```

5 APPENDIX

5.1 Test groups

5.1.1 3rdparty

This group allows testing the interaction of OCCT and 3rdparty products.

DRAW module: VISUALIZATION.

Grid	Commands	Functionality
export	vexport	export of images to different formats
fonts	vtrihedron, vcolorscale, vdrawtext	display of fonts

5.1.2 blend

This group allows testing blends (fillets) and related operations.

DRAW module: MODELING.

Grid	Commands	Functionality
simple	blend	fillets on simple shapes
complex	blend	fillets on complex shapes, non-trivial geometry
tolblend_simple	tolblend, blend	
buildevol	buildevol	
tolblend_buildvol	tolblend, buildevol	use of additional command tolblend
bfuseblend	bfuseblend	
encoderegularity	encoderegularity	

5.1.3 boolean

This group allows testing Boolean operations.

DRAW module: MODELING (packages *BOPTest* and *BRepTest*).

Grids names are based on name of the command used, with suffixes:

- *_2d* – for tests operating with 2d objects (wires, wires, 3d objects, etc.);
- *_simple* – for tests operating on simple shapes (boxes, cylinders, toruses, etc.);
- *_complex* – for tests dealing with complex shapes.

Grid	Commands	Functionality
bcommon_2d	bcommon	Common operation (old algorithm), 2d
bcommon_complex	bcommon	Common operation (old algorithm), complex shapes
bcommon_simple	bcommon	Common operation (old algorithm), simple shapes
bcut_2d	bcut	Cut operation (old algorithm), 2d
bcut_complex	bcut	Cut operation (old algorithm), complex shapes
bcut_simple	bcut	Cut operation (old algorithm), simple shapes
bcutblend	bcutblend	
bfuse_2d	bfuse	Fuse operation (old algorithm), 2d
bfuse_complex	bfuse	Fuse operation (old algorithm), complex shapes
bfuse_simple	bfuse	Fuse operation (old algorithm), simple shapes

Grid	Commands	Functionality
bopcommon_2d	bopcommon	Common operation, 2d
bopcommon_complex	bopcommon	Common operation, complex shapes
bopcommon_simple	bopcommon	Common operation, simple shapes
bopcut_2d	bopcut	Cut operation, 2d
bopcut_complex	bopcut	Cut operation, complex shapes
bopcut_simple	bopcut	Cut operation, simple shapes
bopfuse_2d	bopfuse	Fuse operation, 2d
bopfuse_complex	bopfuse	Fuse operation, complex shapes
bopfuse_simple	bopfuse	Fuse operation, simple shapes
bopsection	bopsection	Section
boptuc_2d	boptuc	
boptuc_complex	boptuc	
boptuc_simple	boptuc	
bsection	bsection	Section (old algorithm)

5.1.4 bugs

This group allows testing cases coming from Mantis issues.

The grids are organized following OCCT module and category set for the issue in the Mantis tracker. See Mapping of OCCT functionality to grid names in group bugs chapter for details.

5.1.5 caf

This group allows testing OCAF functionality.

DRAW module: OCAFKERNEL.

Grid	Commands	Functionality
basic		Basic attributes
bugs		Saving and restoring of document
driver		OCAF drivers
named_shape		*TNaming_NamedShape* attribute
presentation		*AISPresentation* attributes
tree		Tree construction attributes
xlink		XLink attributes

5.1.6 chamfer

This group allows testing chamfer operations.

DRAW module: MODELING.

The test grid name is constructed depending on the type of the tested chamfers. Additional suffix *_complex* is used for test cases involving complex geometry (e.g. intersections of edges forming a chamfer); suffix *_sequence* is used for grids where chamfers are computed sequentially.

Grid	Commands	Functionality
equal_dist		Equal distances from edge
equal_dist_complex		Equal distances from edge, complex shapes
equal_dist_sequence		Equal distances from edge, sequential operations
dist_dist		Two distances from edge

Grid	Commands	Functionality
dist_dist_complex		Two distances from edge, complex shapes
dist_dist_sequence		Two distances from edge, sequential operations
dist_angle		Distance from edge and given angle
dist_angle_complex		Distance from edge and given angle
dist_angle_sequence		Distance from edge and given angle

5.1.7 de

This group tests reading and writing of CAD data files (iges, step) to and from OCCT.

Test cases check transfer status, shape and attributes against expected reference values.

Grid	Commands	Functionality
iges_1, iges_2, iges_3	igesbrep, brepiges, ReadIges, WriteIges	IGES tests
step_1, step_2, step_3, step_4, step_5	stepread, stepwrite, ReadStep, WriteStep	STEP tests

5.1.8 demo

This group allows demonstrating how testing cases are created, and testing DRAW commands and the test system as a whole.

Grid	Commands	Functionality
draw	getsource, restore	Basic DRAW commands
testsystem		Testing system
samples		OCCT samples

5.1.9 draft

This group allows testing draft operations.

DRAW module: MODELING.

Grid	Commands	Functionality
Angle	depouille	Drafts with angle (inclined walls)

5.1.10 feat

This group allows testing creation of features on a shape.

DRAW module: MODELING (package *BRepTest*).

Grid	Commands	Functionality
featdprism		
featlf		
featprism		
featrevol		
featrf		

5.1.11 heal

This group allows testing the functionality provided by *ShapeHealing* toolkit.

DRAW module: XSDRAW

Grid	Commands	Functionality
fix_shape	fixshape	Shape healing
fix_gaps	fixwgaps	Fixing gaps between edges on a wire
same_parameter	sameparameter	Fixing non-sameparameter edges
same_parameter_locked	sameparameter	Fixing non-sameparameter edges
fix_face_size	DT_ApplySeq	Removal of small faces
elementary_to_revolution	DT_ApplySeq	Conversion of elementary surfaces to revolution
direct_faces	directfaces	Correction of axis of elementary surfaces
drop_small_edges	fixsmall	Removal of small edges
split_angle	DT_SplitAngle	Splitting periodic surfaces by angle
split_angle_advanced	DT_SplitAngle	Splitting periodic surfaces by angle
split_angle_standard	DT_SplitAngle	Splitting periodic surfaces by angle
split_closed_faces	DT_ClosedSplit	Splitting of closed faces
surface_to_bspline	DT_ToBspl	Conversion of surfaces to b-splines
surface_to_bezier	DT_ShapeConvert	Conversion of surfaces to bezier
split_continuity	DT_ShapeDivide	Split surfaces by continuity criterion
split_continuity_advanced	DT_ShapeDivide	Split surfaces by continuity criterion
split_continuity_standard	DT_ShapeDivide	Split surfaces by continuity criterion
surface_to_revolution_advanced	DT_ShapeConvertRev	Convert elementary surfaces to revolutions, complex cases
surface_to_revolution_standard	DT_ShapeConvertRev	Convert elementary surfaces to revolutions, simple cases
update_tolerance_locked	updatetolerance	Update the tolerance of shape so that it satisfy the rule: toler(face)<=toler(edge)<=toler(vertex)

5.1.12 mesh

This group allows testing shape tessellation (*BRepMesh*) and shading.

DRAW modules: MODELING (package *MeshTest*), VISUALIZATION (package *ViewerTest*)

Grid	Commands	Functionality
advanced_shading	vdisplay	Shading, complex shapes
standard_shading	vdisplay	Shading, simple shapes
advanced_mesh	mesh	Meshing of complex shapes
standard_mesh	mesh	Meshing of simple shapes
advanced_incmesh	incmesh	Meshing of complex shapes
standard_incmesh	incmesh	Meshing of simple shapes
advanced_incmesh_parallel	incmesh	Meshing of complex shapes, parallel mode
standard_incmesh_parallel	incmesh	Meshing of simple shapes, parallel mode

5.1.13 mkface

This group allows testing creation of simple surfaces.

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
after_trim	mkface	
after_offset	mkface	
after_extsurf_and_offset	mkface	
after_extsurf_and_trim	mkface	
after_revsurf_and_offset	mkface	
mkplane	mkplane	

5.1.14 nproject

This group allows testing normal projection of edges and wires onto a face.

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
Base	nproject	

5.1.15 offset

This group allows testing offset functionality for curves and surfaces.

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
compshape	offsetcompshape	Offset of shapes with removal of some faces
faces_type_a	offsetparameter, offsetload, offsetperform	Offset on a subset of faces with a fillet
faces_type_i	offsetparameter, offsetload, offsetperform	Offset on a subset of faces with a sharp edge
shape_type_a	offsetparameter, offsetload, offsetperform	Offset on a whole shape with a fillet
shape_type_i	offsetparameter, offsetload, offsetperform	Offset on a whole shape with a fillet
shape	offsetshape	
wire_closed_outside_0_005, wire_closed_outside_0_025, wire_closed_outside_0_075, wire_closed_inside_0_005, wire_closed_inside_0_025, wire_closed_inside_0_075, wire_unclosed_outside_0_005, wire_unclosed_outside_0_025, wire_unclosed_outside_0_075	mkoffset	2d offset of closed and unclosed planar wires with different offset step and directions of offset (inside / outside)

5.1.16 pipe

This group allows testing construction of pipes (sweeping of a contour along profile).

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
Standard	pipe	

5.1.17 prism

This group allows testing construction of prisms.

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
seminf	prism	

5.1.18 sewing

This group allows testing sewing of faces by connecting edges.

DRAW module: MODELING (package *BRepTest*)

Grid	Commands	Functionality
tol_0_01	sewing	Sewing faces with tolerance 0.01
tol_1	sewing	Sewing faces with tolerance 1
tol_100	sewing	Sewing faces with tolerance 100

5.1.19 thrusection

This group allows testing construction of shell or a solid passing through a set of sections in a given sequence (loft).

Grid	Commands	Functionality
solids	thrusection	Lofting with resulting solid
not_solids	thrusection	Lofting with resulting shell or face

5.1.20 xcaf

This group allows testing extended data exchange packages.

Grid	Commands	Functionality
dxc, dxc_add_ACL, dxc_add_CL, igs_to_dxc, igs_add_ACL, brep_to_igs_add_CL, stp_↩ to_dxc, stp_add_ACL, brep_to_stp_add_CL, brep_to_dxc, add_ACL_brep, brep_add_CL		Subgroups are divided by format of source file, by format of result file and by type of document modification. For example, *brep_to_igs* means that the source shape in brep format was added to the document, which was saved into igs format after that. The postfix *add_CL* means that colors and layers were initialized in the document before saving and the postfix *add_ACL* corresponds to the creation of assembly and initialization of colors and layers in a document before saving.

5.2 Mapping of OCCT functionality to grid names in group ∗bugs∗

OCCT Module / Mantis category	Toolkits	Test grid in group bugs

OCCT Module / Mantis category	Toolkits	Test grid in group bugs
Application Framework	PTKernel, TKPShape, TKCDF, TKLCAF, TKCAF, TKBinL, TKXmlL, TKShapeSchema, TKPLCAF, TKBin, TKXml, TKPCAF, FWOSPlugin, TKStdLSchema, TKStdSchema, TKTObj, TKBinTObj, TKXmlTObj	caf
Draw	TKDraw, TKTopTest, TKViewerTest, TKXSDRAW, TKDCAF, TKXDEDRAW, TKTObjDRAW, TKQADraw, DRAWEXE, Problems of testing system	draw
Shape Healing	TKShHealing	heal
Mesh	TKMesh, TKXMesh	mesh
Data Exchange	TKIGES	iges
Data Exchange	TKSTEPBase, TKSTEPAttr, TKSTEP209, TKSTEP	step
Data Exchange	TKSTL, TKVRML	stlvrml
Data Exchange	TKXSBase, TKXCAF, TKXCAFSchema, TKXDEIGES, TKXDESTEP, TKXmlXCAF, TKBinXCAF	xde
Foundation Classes	TKernel, TKMath	fclasses
Modeling_algorithms	TKGeomAlgo, TKTopAlgo, TKPrim, TKBO, TKBool, TKHLR, TKFillet, TKOffset, TKFeat, TKXMesh	modalg
Modeling Data	TKG2d, TKG3d, TKGeomBase, TKBRep	moddata
Visualization	TKService, TKV2d, TKV3d, TKOpenGl, TKMeshVS, TKNIS	vis

5.3 Recommended approaches to checking test results

5.3.1 Shape validity

Run command *checkshape* on the result (or intermediate) shape and make sure that *parse.rules* of the test grid or group reports bad shapes (usually recognized by word "Faulty") as error.

Example

```
checkshape result
```

To check the number of faults in the shape command *checkfaults* can be used.

Use: checkfaults shape source_shape [ref_value=0]

The default syntax of *checkfaults* command:

```
checkfaults results a_1
```

The command will check the number of faults in the source shape (*a_1*) and compare it with number of faults in the resulting shape (*result*). If shape *result* contains more faults, you will get an error:

```
checkfaults results a_1
Error : Number of faults is 5
```

It is possible to set the reference value for comparison (reference value is 4):

```
checkfaults results a_1 4
```

If number of faults in the resulting shape is unstable, reference value should be set to "-1". As a result command *checkfaults* will return the following error:

```
checkfaults results a_1 -1
Error : Number of faults is UNSTABLE
```

5.3.2 Shape tolerance

The maximal tolerance of sub-shapes of each kind of the resulting shape can be extracted from output of tolerance command as follows:

```
set tolerance [tolerance result]
regexp { *FACE +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_face
regexp { *EDGE +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_edgee
regexp { *VERTEX +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_vertex
```

It is possible to use command *checkmaxtol* to check maximal tolerance of shape and compare it with reference value.

Use: checkmaxtol shape [options...]

Allowed options are:

- *-ref* – reference value of maximum tolerance;
- *-source* – list of shapes to compare with;
- *-min_tol* – minimum tolerance for comparison;
- *-multi_tol* – tolerance multiplier.

The default syntax of *checkmaxtol* command for comparison with the reference value:

```
checkmaxtol result -ref 0.00001
```

There is an opportunity to compare max tolerance of resulting shape with max tolerance of source shape. In the following example command *checkmaxtol* gets max tolerance among objects *a_1* and *a_2*. Then it chooses the maximum value between founded tolerance and value -min_tol (0.000001) and multiply it on the coefficient -multi↩ _tol (i.e. 2):

```
checkmaxtol result -source {a_1 a_2} -min_tol 0.000001 -multi_tol 2
```

If the value of maximum tolerance more than founded tolerance for comparison, the command will return an error.

Also, command *checkmaxtol* can be used to get max tolerance of the shape:

```
set maxtol [checkmaxtol result]
```

5.3.3 Shape volume, area, or length

Use command *vprops, sprops,* or *lprops* to correspondingly measure volume, area, or length of the shape produced by the test. The value can be extracted from the result of the command by *regexp*.

Example:

```
# check area of shape result with 1% tolerance
regexp {Mass +: +([-0-9.+eE]+)} [sprops result] dummy area
if { abs($area - $expected) > 0.1 + 0.01 * abs ($area) } {
    puts "Error: The area of result shape is $area, while expected $expected"
}
```

5.3.4 Memory leaks

The test system measures the amount of memory used by each test case. Considerable deviations (as well as the overall difference) in comparison with reference results can be reported by command *testdiff* (see Checking non-regression of performance, memory, and visualization).

To check memory leak on a particular operation, run it in a cycle, measure the memory consumption at each step and compare it with a threshold value. The command *checktrend* (defined in *tests/bugs/begin*) can be used to analyze a sequence of memory measurements and to get a statistically based evaluation of the leak presence.

Example:

```
set listmem {}
for {set i 1} {$i < 100} {incr i} {
    # run suspect operation
    ...
    # check memory usage (with tolerance equal to half page size)
    lappend listmem [expr [meminfo w] / 1024]
    if { [checktrend $listmem 0 256 "Memory leak detected"] } {
        puts "No memory leak, $i iterations"
        break
    }
}
```

5.3.5 Visualization

The following command sequence allows you to take a snapshot of the viewer, give it the name of the test case, and save in the directory indicated by Tcl variable *imagedir*.

```
vinit
vclear
vdisplay result
vsetdispmode 1
vfit
vzfit
vdump $imagedir/${casename}_shading.png
```

This image will be included in the HTML log produced by *testgrid* command and will be checked for non-regression through comparison of images by command *testdiff*.

Also it is possible to use command *checkview* to make a snapshot of the viewer.

Use: checkview [options...] Allowed options are:

- *-display shapename* – displays shape with name *shapename*;
- *-3d* – displays shape in 3d viewer;
- *-2d [v2d / smallview]* - displays shape in 2d viewer (the default viewer is *smallview*);
- *-path PATH* – sets the location of the saved viewer screenshot;
- *-vdispmode N* – sets *vdispmode* for 3d viewer (default value is 1)
- *-screenshot* – makes a screenshot of already created viewer
- The procedure can check a property of shape (length, area or volume) and compare it with value *N*:
  - *-l [N]*
  - *-s [N]*
  - *-v [N]*
  - If the current property is equal to value *N*, the shape is marked as valid in the procedure.
  - If value *N* is not given, the procedure will mark the shape as valid if the current property is non-zero.
- *-with {a b c}* – displays shapes *a, b* and *c* together with the shape (if the shape is valid)
- *-otherwise {d e f}* – displays shapes *d, e* and *f* instead of the shape (if the shape is NOT valid)

Note that is required to use either option *-2d* or option *-3d*.

Examples:

```
checkview -display result -2d -path ${imagedir}/${test_image}.png
checkview -display result -3d -path ${imagedir}/${test_image}.png
checkview -display result_2d -2d v2d -path ${imagedir}/${test_image}.png
```

```
box a 10 10 10
box b 5 5 5 10 10 10
bcut result b a
set result_vertices [explode result v]
checkview -display result -2d -with ${result_vertices} -otherwise { a b } -l -path ${imagedir}/${test_image
      }.png
```

```
box a 10 10 10
box b 5 5 5 10 10 10
bcut result b a
vinit
vdisplay a b
vfit
checkview -screenshot -3d -path ${imagedir}/${test_image}.png
```

5.3.6 Number of free edges

Procedure *checkfreebounds* compares the number of free edges with a reference value.

Use: checkfreebounds shape ref_value [options...]

Allowed options are:

- *-tol N* – used tolerance (default -0.01);
- *-type N* – used type, possible values are "closed" and "opened" (default "closed").

```
checkfreebounds result 13
```

Option *-tol N* defines tolerance for command *freebounds*, which is used within command *checkfreebounds*.

Option *-type N* is used to select the type of counted free edges: closed or open.

If the number of free edges in the resulting shape is unstable, the reference value should be set to "-1". As a result command *checkfreebounds* will return the following error:

```
checkfreebounds result -1
Error : Number of free edges is UNSTABLE
```

5.3.7 Compare numbers

Procedure *checkreal* checks the equality of two reals with a tolerance (relative and absolute).

Use: checkreal name value expected tol_abs tol_rel

```
checkreal "Some important value" $value 5 0.0001 0.01
```

5.3.8 Check number of sub-shapes

Procedure *checknbshapes* compares the number of sub-shapes in "shape" with the given reference data.

Use: checknbshapes shape [options...]

Allowed options are:

- *-vertex N*
- *-edge N*

- *-wire N*
- *-face N*
- *-shell N*
- *-solid N*
- *-compsolid N*
- *-compound N*
- *-shape N*
- *-t* – compares the number of sub-shapes in "shape" counting the same sub-shapes with different location as different sub-shapes.
- *-m msg* – prints "msg" in case of error

```
checknbshapes result -vertex 8 -edge 4
```

5.3.9 Check pixel color

Command *checkcolor* can be used to check pixel color.

Use: checkcolor x y red green blue

where:

- *x, y* – pixel coordinates;
- *red green blue* – expected pixel color (values from 0 to 1).

This procedure checks color with tolerance (5x5 area).

Next example will compare color of point with coordinates x=100 y=100 with RGB color R=1 G=0 B=0. If colors are not equal, procedure will check the nearest ones points (5x5 area)

```
checkcolor 100 100 1 0 0
```

5.3.10 Compute length, area and volume of input shape

Procedure *checkprops* computes length, area and volume of the input shape.

Use: checkprops shapename [options...]

Allowed options are:

- *-l LENGTH* – command *lprops*, computes the mass properties of all edges in the shape with a linear density of 1;
- *-s AREA* – command *sprops*, computes the mass properties of all faces with a surface density of 1;
- *-v VOLUME* – command *vprops*, computes the mass properties of all solids with a density of 1;
- *-eps EPSILON* – the epsilon defines relative precision of computation;
- *-deps DEPSILON* – the epsilon defines relative precision to compare corresponding values;
- *-equal SHAPE* – compares area, volume and length of input shapes. Puts error if they are not equal;
- *-notequal SHAPE* – compares area, volume and length of input shapes. Puts error if they are equal.

Options *-l, -s* and *-v* are independent and can be used in any order. Tolerance *epsilon* is the same for all options.

```
checkprops result -s 6265.68
checkprops result -s -equal FaceBrep
```

5.3.11 Parse output dump and compare it with reference values

Procedure *checkdump* is used to parse output dump and compare it with reference values.

Use: checkdump shapename [options...]

Allowed options are:

- *-name NAME* – list of parsing parameters (e.g. Center, Axis, etc.);
- *-ref VALUE* – list of reference values for each parameter in *NAME*;
- *-eps EPSILON* – the epsilon defines relative precision of computation.

```
checkdump result -name {Center Axis XAxis YAxis Radii} -ref {{-70 0} {-1 -0} {-1 -0} {0 -1} {20 10}} -eps 0
      .01
```

5.3.12 Compute length of input curve

Procedure *checklength* computes length of the input curve.

Use: checklength curvename [options...]

Allowed options are:

- *-l LENGTH* – command *length*, computes the length of the input curve with precision of computation;
- *-eps EPSILON* – the epsilon defines a relative precision of computation;
- *-equal CURVE* – compares the length of input curves. Puts error if they are not equal;
- *-notequal CURVE* – compares the length of input curves. Puts error if they are equal.

```
checklength cp1 -l 7.278
checklength res -l -equal ext_1
```

5.3.13 Check maximum deflection, number of triangles and nodes in mesh

Command *checktrinfo* can be used to to check the maximum deflection, as well as the number of nodes and triangles in mesh.

Use: checktrinfo shapename [options...]

Allowed options are:

- *-tri [N]* – compares the current number of triangles in *shapename* mesh with the given reference data. If reference value N is not given and the current number of triangles is equal to 0, procedure *checktrinfo* will print an error.
- *-nod [N]* – compares the current number of nodes in *shapename* mesh with the given reference data. If reference value N is not given and the current number of nodes is equal to 0, procedure *checktrinfo* will print an error.
- *-defl [N]* – compares the current value of maximum deflection in *shapename* mesh with the given reference data. If reference value N is not given and current maximum deflection is equal to 0, procedure *checktrinfo* will print an error.
- *-max_defl N* – compares the current value of maximum deflection in *shapename* mesh with the max possible value.
- *-tol_abs_tri N* – absolute tolerance for comparison of number of triangles (default value 0).
- *-tol_rel_tri N* – relative tolerance for comparison of number of triangles (default value 0).

- *-tol_abs_nod N* – absolute tolerance for comparison of number of nodes (default value 0).
- *-tol_rel_nod N* – relative tolerance for comparison of number of nodes (default value 0).
- *-tol_abs_defl N* – absolute tolerance for deflection comparison (default value 0).
- *-tol_rel_defl N* – relative tolerance for deflection comparison (default value 0).
- *-ref [trinfo a]* – compares deflection, number of triangles and nodes in *shapename* and *a*.

Note that options *-tri, -nod* and *-defl* do not work together with option *-ref*.

Examples:

Comparison with some reference values:

```
checktrinfo result -tri 129 -nod 131 -defl 0.01
```

Comparison with another mesh:

```
checktrinfo result -ref [tringo a]
```

Comparison of deflection with the max possible value:

```
checktrinfo result -max_defl 1
```

Check that the current values are not equal to zero:

```
checktrinfo result -tri -nod -defl
```

Check that the number of triangles and the number of nodes are not equal to some specific values:

```
checktrinfo result -tri !10 -nod !8
```

It is possible to compare current values with reference values with some tolerances. Use options *-tol_∗* for that.

```
checktrinfo result -defl 1 -tol_abs_defl 0.001
```