Deqp User Guide

Contents

Introduction

The purpose of this document is to give an overview of the GPU testing suite called deqp (drawElements Quality Program) as well as instructions for porting and building the test modules.

You can access the code for deqp in AOSP, in the following location: https://android.googlesource.com/platform/external/deqp

To use the latest submitted code, use the `deqp-dev` branch. If you want the code that matches the Android 5.0 CTS release, use the `lollipop-release` branch.

Deploying deqp

Deploying the deqp test suite to a new environment involves four steps. The steps listed below are described in more detail later in this document.

1. Setting up build system
2. Implementing platform port
3. Deploying to tarrget device(s)
4. Automated test system integration

Source layout

The source code for the deqp test modules and supporting libraries are laid out as shown in the table below. The listing is not complete, but highlights the most important directories.

`android`	Android tester sources and build scripts
`data`	Test data files
`modules`	Test module sources
`egl`	EGL module
`gles2`	GLES2 module
`gles3`	GLES3 module
`gles31`	GLES3.1 module
`targets`	Target-specific build configuration files
`framework`	deqp test module framework and utilities
`delibs`	Base portability and build libraries
`platform`	Platform ports
`qphelper`	Test program integration library (C)
`common`	Deqp framework (C++)
`opengl, egl`	API specific utilities
`execserver`	Device-side ExecServer source
`executor`	Host-side test executor shell tool and utilities
`external`	Build stub dir for external libs libpng and zlib

Open-Source components

The deqp uses libpng and zlib. They can be fetched from the web with the script `external/fetch_sources.py` or with git pulls from git repositories `platform/external/[libpng,zlib]`.

Building test programs

The test framework has been designed with portability in mind. The only mandatory requirements are full C++ support and standard system libraries for IO, threads and sockets.

CMake build system

Deqp sources have build scripts for CMake, which is the preferred tool for compiling the test programs.

CMake is an open-source build system that supports multiple platforms and toolchains. CMake generates native makefiles or IDE project files from target-independent configuration files. For more information on CMake, please see documentation at http://www.cmake.org/cmake/help/documentation.html.

CMake supports and recommends out-of-source-tree builds, i.e., you should always create makefiles or project files into a separate build directory outside the source tree. CMake does not have any kind of "distclean" target, so removing any files generated by CMake must be done manually.

Configuration options are given to CMake using `-D<OPTION_NAME>=<VALUE>` syntax. Some commonly used options for deqp are listed below.

`DEQP_TARGET`	Target name, for example "android". Deqp CMake scripts will include the file `targets/<DEQP_TARGET>/<DEQP_TARGET>.cmake` and expects to find target-specific build options from there.
`CMAKE_TOOLCHAIN_FILE`	Path to toolchain file for CMake. Used for cross compilation.
`CMAKE_BUILD_TYPE`	Build type for makefile targets. Valid values are “Debug” and “Release”. Note that the interpretation and default type depend on the targeted build system. See CMake documentation for details.

Creating target build file

The deqp build system is configured for new targets using target build files. The target file defines which features platform supports and what libraries or additional include paths are required. Target file names follow `targets/<name>/<name>.cmake` format and the target is selected using `DEQP_TARGET` build parameter.

File paths in target files are relative to the base `deqp` directory, not the `targets/<name>` directory. The following standard variables can be set by target build file:

`DEQP_TARGET_NAME`	Target name (will be included into test logs).
`DEQP_SUPPORT_GLES2`	Is GLES2 supported (default: OFF)
`DEQP_GLES2_LIBRARIES`	GLES2 libraries (leave empty if not supported or dynamic loading is used)
`DEQP_SUPPORT_GLES3`	Is GLES3.x supported (default: OFF)
`DEQP_GLES3_LIBRARIES`	GLES3.x libraries (leave empty if not supported or dynamic loading is used)
`DEQP_SUPPORT_VG`	Is OpenVG supported (default: OFF)
`DEQP_OPENVG_LIBRARIES`	OpenVG libraries (leave empty if not supported or dynamic loading is used)
`DEQP_SUPPORT_EGL`	Is EGL supported (default: OFF)
`DEQP_EGL_LIBRARIES`	EGL libraries (leave empty if not supported or dynamic loading is used)
`DEQP_PLATFORM_LIBRARIES`	Additional platform-specific libraries required for linking
`DEQP_PLATFORM_COPY_LIBRARIES`	List of libraries that are copied to each test binary build directory. Can be used to copy libraries that are needed for running tests but are not in default search path.
`TCUTIL_PLATFORM_SRCS`	Platform port source list. Default sources are determined based on the capabilities and OS. **Note:** Paths are relative to the following: `framework/platform`

The target build file can add additional include or link paths using `include_directories()` and `link_directories()` CMake functions.

Win32 build

The easiest way to build deqp modules for Windows is to use the CMake build system. You will need CMake 2.6.12 or newer and the Microsoft Visual C/C++ compiler. The deqp has been tested with Visual Studio 2013.

Visual Studio project files can be generated with a following command:

```
cmake path\to\src\deqp -G"Visual Studio 12”
```

A 64-bit build can be made by selecting “Visual Studio <ver> Win64” as build generator:

```
cmake path\to\src\deqp -G"Visual Studio 12 Win64”
```

You can also generate NMake makefiles with the `-G"NMake Makefiles"` option as well as the build type (`-DCMAKE_BUILD_TYPE="Debug"` or `"Release"`).

Rendering context creation

Rendering context can be created either with WGL or with EGL on Windows.

WGL support

All Win32 binaries support GL context creation with WGL as it requires only standard libraries. WGL context can be selected using the `--deqp-gl-context-type=wgl` command line argument. In the WGL mode the deqp uses the `WGL_EXT_create_context_es_profile` extension to create OpenGL ES contexts. This has been tested to work with latest drivers from NVIDIA and Intel. AMD drivers do not support the required extension.

EGL support

The deqp is built with dynamic loading for EGL on Windows if DEQP_SUPPORT_EGL is ON. This is the default in most targets. Then, if the host has EGL libraries available, it is possible to run tests with that with the command line parameter `--deqp-gl-context-type=egl`.

Android build

The Android build uses CMake build scripts for building the native test code. Java parts, i.e., the Test Execution Server and the Test Application Stub, are compiled using the standard Android build tools.

In order to compile deqp test programs for Android with the provided build scripts you will need:

- Android NDK, the latest available version (`android/scripts/common.py` lists the required version)
- Android stand-alone SDK with API 13, SDK Tools, SDK Platform-tools, and SDK Build-tools packages installed
- Apache Ant 1.9.4 (required by the Java code build)
- CMake 2.8.12 or newer
- Python 2.6 or newer in 2.x series. Python 3.x is not supported.
- For Windows: Either NMake or JOM in `PATH`
    - JOM enables faster builds; available at http://qt-project.org/wiki/jom
- Optional: Ninja make also supported on Linux

Ant and SDK binaries are located based on the PATH environment variable with certain overriding defaults. The logic is controlled by `android/scripts/common.py`.

The NDK directory must be either `~/android-ndk-<version>` or `C:/android/android-ndk-<version>` or defined via the `ANDROID_NDK_PATH` environment variable.

Deqp on-device components, the Test Execution Service and test programs, are built by executing the `android/scripts/build.py` script. The final .apk is created into `android/package/bin` and it can be installed `install.py` script. If the Command Line Executor is used, the ExecService is launched with `launch.py` script on the device via ADB. The scripts can be executed from any directory.

Linux build

Test binaries and command line utilities can be built for Linux by generating Makefiles using CMake. There are multiple, predefined build targets that are useful when building for Linux:

`default`	Default target that uses CMake platform introspection to determine support for various APIs.

`x11_glx`	Uses GLX to create OpenGL (ES) contexts.
`x11_egl`	Uses EGL to create OpenGL (ES) contexts.
`x11_egl_glx`	Supports both GLX and EGL with X11.

Always use `-DCMAKE_BUILD_TYPE=<Debug|Release>` to defined the build type. Release is a good default. Without it, a default, unoptimized release build is made.

The `-DCMAKE_C_FLAGS` and `-DCMAKE_CXX_FLAGS` command-line arguments can be used to pass extra arguments to the compiler. For example the 32-bit or 64-bit build can be done by setting `-DCMAKE_C(XX)_FLAGS="-m32"` or `"-m64"` respectively. If not specified, the toolchain native architecture, typically 64-bit on the 64-bit toolchain, is used.

The `-DCMAKE_LIBRARY_PATH` and `-DCMAKE_INCLUDE_PATH` arguments can be used for CMake, to give CMake additional library or include search paths.

An example of a full command line used to do a 32-bit debug build against driver headers and libraries in a custom location is the following:

```
$ cmake <path to src>/deqp -DDEQP_TARGET=x11_egl -DCMAKE_C_FLAGS="-m32"
-DCMAKE_CXX_FLAGS="-m32" -DCMAKE_BUILD_TYPE=Debug
-DCMAKE_LIBRARY_PATH="<path to driver>/lib"
-DCMAKE_INCLUDE_PATH="<path to driver>/inc"
$ make -j4
```

Cross-compiling

Cross-compiling can be achieved by using a CMake toolchain file. The toolchain file specifies the compiler to use, along with custom search paths for libraries and headers. Several toolchain files for common scenarios are included in the release package in the `framework/delibs/cmake` directory.

In addition to standard CMake variables, the following deqp-specific variables can be set by the toolchain file. CMake can usually detect `DE_OS`, `DE_COMPILER` and `DE_PTR_SIZE` correctly but `DE_CPU` must be set by the toolchain file.

`DE_OS`	Operating system, supported values are: `DE_OS_WIN32, DE_OS_UNIX, DE_OS_WINCE, DE_OS_OSX, DE_OS_ANDROID, DE_OS_SYMBIAN, DE_OS_IOS`.
`DE_COMPILER`	Compiler type, supported values are: `DE_COMPILER_GCC, DE_COMPILER_MSC, DE_COMPILER_CLANG`.
`DE_CPU`	CPU type, values are: `DE_CPU_ARM, DE_CPU_X86`.

`DE_PTR_SIZE`	sizeof(void*) on the platform. Supported values are 4 and 8.

The toolchain file can be selected using the `CMAKE_TOOLCHAIN_FILE` build parameter. For example, the following would create makefiles for a build using the CodeSourcery cross-compiler for ARM/Linux:

```
cmake <path to src>/deqp -DDEQP_BUILD_TYPE=”Release”
-DCMAKE_TOOLCHAIN_FILE=<path to
src>/delibs/cmake/toolchain-arm-cs.cmake -DARM_CC_BASE=<path to cc
directory>
```

Run-time linking of GLES and EGL libraries

The deqp does not need entry points of the API under test during linking. The test code always accesses the APIs through function pointers. Entry points can then be loaded dynamically at run time or the platform port can provide them at link time.

If support for an API is turned on in the build settings and link libraries are not provided, the deqp will load the needed entry points at run time. If the static linking is desired, provide the needed link libraries in the `DEQP_<API>_LIBRARIES` build configuration variable.

Porting test framework

Porting the deqp involves three steps: Adapting base portability libraries, implementing test framework platform integration interfaces, and porting the execution service.

The table below lists locations for likely porting changes. Anything beyond them is likely to be exotic.

`framework/delibs/debase`
`framework/delibs/dethread`
`framework/delibs/deutil`
	Any necessary implementations of OS-specific code.

`framework/qphelper/qpCrashHandler.c`	Optional: Implementation for your OS.
`framework/qphelper/qpWatchDog.c`	Implementation for your OS. Current one is based on `dethread` and standard C library.
`framework/platform`	New platform port and application stub can be implemented as described in the Porting Guide.

Base portability libraries

The base portability libraries already support Windows, most Linux variants, OS X, iOS, and Android. If the test target runs on one of those operating systems, most likely there is no need to touch the base portability libraries at all.

Test framework platform port

The deqp test framework platform port requires two components: An application entry point and a platform interface implementation.

The application entry point is responsible for creating the platform object, creating a command line (`tcu::CommandLine`) object, opening a test log (`tcu::TestLog`) and iterating the test application (`tcu::App`). If the target OS supports a standard `main()` entry point, `tcuMain.cpp` can be used as the entry point implementation.

The deqp platform API is described in detail in the following files:

`framework/common/tcuPlatform.hpp`	Base class for all platform ports
`framework/opengl/gluPlatform.hpp`	OpenGL platform interface
`framework/egl/egluPlatform.hpp`	EGL platform interface
`framework/platform/tcuMain.cpp`	Standard application entry point

The base class for all platform ports is the `tcu::Platform`. The platform port can optionally support GL- and EGL-specific interfaces. See the following table for an overview of what needs to be implemented in order to run the tests.

OpenGL (ES) test modules	GL platform interface
EGL test module	EGL platform interface

Detailed instructions for implementing platform ports can be found in the porting layer headers.

Test Execution Service

To use the deqp test execution infrastructure or command line executor, the test execution service must be available on the target. A portable C++ implementation of the service is provided in the `execserver` directory. The stand-alone binary is built as a part of the deqp test module build for PC targets. You can modify `execserver/CMakeLists.txt` to enable a build on other targets.

The C++ version of the Test Execution Service accepts two command line parameters:

- `--port=<port>` will set the TCP port that the server listens on. The default is 50016.
- `--single` will terminate the server process when the client disconnects. By default, the server process will stay up to serve further test execution requests.

Deploying Test Programs

Linux and Windows environments

The following files and directories must be copied to the target:

Execution Server	`build/execserver/execserver`	`<dst>/execserver`
EGL Module	`build/modules/egl/deqp-egl`	`<dst>/deqp-egl`

GLES2 Module	`build/modules/gles2/deqp-gles2`
	`data/gles2`
		`<dst>/deqp-gles2`
		`<dst>/gles2`

GLES3 Module	`build/modules/gles3/deqp-gles3`
	`data/gles3`
		`<dst>/deqp-gles3`
		`<dst>/gles3`

GLES3.1 Module	`build/modules/gles31/deqp-gles31`
	`data/gles31`
		`<dst>/deqp-gles31`
		`<dst>/gles31`

Execution service and test binaries can be deployed anywhere in the target file system. Test binaries expect to find data directories in the current working directory.

Start the Test Execution Service on the target device. For more details on starting the service, see Test Execution Service.

Command line arguments

The following table lists command line arguments that affect execution of all test programs.

`--deqp-case=<casename>`	Run cases that match a given pattern. Wildcard (*) is supported.
`--deqp-log-filename=<filename>`	Write test results to the file whose name you provide. The Test Execution Service will set the filename when starting a test.

`--deqp-stdin-caselist`
`--deqp-caselist=<caselist>`
`--deqp-caselist-file=<filename>`
	Read case list from stdin or from a given argument. The Test Execution Service will set the argument according to the execution request received. See the next section for a description of the case list format.

`--deqp-test-iteration-count=<count>`	Override iteration count for tests that support a variable number of iterations.
`--deqp-base-seed=<seed>`	Base seed for the test cases that use randomization.

GLES2 and GLES3-specific arguments

`--deqp-gl-context-type=<type>`	OpenGL context type. Available context types depend on the platform. On platforms supporting EGL, the value `egl` can be used to select the EGL context.
`--deqp-gl-config-id=<id>`	Run tests for the provided GL configuration ID. Interpretation is platform-dependent. On the EGL platform this is the EGL configuration ID.
`--deqp-gl-config-name=<name>`	Run tests for a named GL configuration. Interpretation is platform-dependent. For EGL the format is `rgb(a)<bits>d<bits>s<bits>`. For example, a value of `rgb888s8` will select the first configuration where the color buffer is RGB888 and the stencil buffer has 8 bits.
`--deqp-gl-context-flags=<flags>`	Creates a context. Specify `robust` or `debug`.

`--deqp-surface-width=<width>`
`--deqp-surface-height=<height>`
	Try to create a surface with a given size. Support for this is optional.

`--deqp-surface-type=<type>`	Use a given surface type as the main test rendering target. Possible types are `window`, `pixmap`, `pbuffer`, and `fbo`.
`--deqp-screen-rotation=<rotation>`	Screen orientation in increments of 90 degrees for platforms that support it.

Test case list format

The test case list can be given in two formats. The first option is to list the full name of each test on a separate line in a standard ASCII file. As the test sets grow, the repetitive prefixes can be cumbersome. To avoid repeating the prefixes, use a trie (also known as a prefix tree) syntax shown below.

```
{nodeName{firstChild{…},…lastChild{…}}}
```

For example, please review the following:

```
{dEQP-EGL{config-list,create_context{rgb565_depth_stencil}}}
```

That list would translate into two test cases:

```
dEQP-EGL.config_list
dEQP-EGL.create_context.rgb565_depth_stencil
```

Android

The Android application package contains everything required, including the Test Execution Service, test binaries, and data files. The test activity is a `NativeActivity` and it uses EGL, which requires Android 3.2 or later.

The application package can be installed with the following command:

```
adb –d install –r com.drawelements.deqp.apk
```
[1]

To launch the test execution service and to setup port forwarding, use the following:

```
adb –d forward tcp:50016 tcp:50016

adb –d shell am start –n
com.drawelements.deqp/.execserver.ServiceStarter
```

Debug prints can be enabled by executing the following before starting the tests:

```
adb –d shell setprop log.tag.dEQP DEBUG
```

Executing tests on Android without Android CTS

If you want to manually start the test execution activity, construct an Android intent that targets `android.app.NativeActivity`. The activities can be found in the `com.drawelements.deqp` package. The command line must be supplied as an extra string with key "`cmdLine`" in the Intent.

A test log will be written to `/sdcard/dEQP-log.qpa`. If the test run does not start normally, additional debug information is available in the device log.

The activity can be launched from the command line using the "`am`" utility. For example, to run `dEQP-GLES2.info` tests on a platform supporting `NativeActivity`, the following command can be used:

```
adb -d shell am start -n
com.drawelements.deqp/android.app.NativeActivity -e cmdLine "deqp
--deqp-case=dEQP-GLES2.info.* --deqp-log-filename=/sdcard/dEQP-Log.qpa
```

[1] The name depends on the build. The name shown is the name of the APK in the Android CTS package.

Debugging on Android

To run the tests under the GDB debugger on Android, first compile and install the debug build by running the following two scripts:

```
python android/scripts/build.py --native-build-type=Debug
python android/scripts/install.py
```

After the debug build is installed on the device, to launch the tests under GDB running on the host, run the following command:

```
pythonandroid/scripts/debug.py
--deqp-commandline=”--deqp-log-filename=/sdcard/TestLog.qpa
--deqp-case=dEQP-GLES2.functional.*”
```

The deqp command line will depend on test cases to be executed and other required parameters. The script will add a default breakpoint into the beginning of the deqp execution (`tcu::App::App`).

The `debug.py` script accepts multiple command line arguments, e.g. for the following:

- Setting the breakpoints for debugging
- gdbserver connection parameters
- Paths to additional binaries to debug

Running `debug.py --help` will list all command line parameters, with explanations.

The script copies some default libraries from the target device to get symbol listings.

More libraries to be included can be added via `debug.py` command line parameters. If there is a need to step through driver code, for example if the GDB needs to know the locations of the binaries with full debug information. The `debug.py` script writes out a configuration file for the GDB starting from line 132 of the script file. Additional paths to binaries, etc., can be added there, but supplying correct command line parameters should be enough.

Notes:

- On Windows, the gdb binary requires `libpython2.7.dll`. Add `<path to ndk>/prebuilt/windows/bin` to the PATH variable before launching `debug.py`.

- Native code debugging does not work on stock Android 4.3. See the Android bug report below for suggested workarounds. The bug has been fixed in Android 4.4; see the following: https://code.google.com/p/android/issues/detail?id=58373

Special test groups

Some test groups may need or support special command line options, or require special care when used on certain systems.

Memory allocation stress tests

Memory allocation stress tests exercise out-of-memory conditions by repeatedly allocating certain resources until the driver reports an out-of-memory error.

On certain platforms, such as Android and most Linux variants, the following can occur: The operating system may kill the test process instead of allowing a driver to handle or otherwise provide an out-of-memory error. On such platforms, tests that are designed to cause out-of-memory errors are disabled by default, and must be enabled using the `--deqp-test-oom=enable` command line argument. It is recommended that you run such tests manually to check if the system behaves correctly under resource pressure. However, in such a situation, a test process crash should be interpreted as a pass.

Test groups

```
dEQP-GLES2.stress.memory.*
dEQP-GLES3.stress.memory.*
```

Long-running rendering stress tests

Rendering stress tests are designed to reveal robustness issues under sustained rendering load. By default the tests will execute only a few iterations, but they can be configured to run indefinitely by supplying the `--deqp-test-iteration-count=-1` command line argument. The test watchdog should be disabled (`--deqp-watchdog=disable`) when running these tests for a long period of time.

Test groups

```
dEQP-GLES2.stress.long.*

dEQP-GLES3.stress.long.*
```

Integration to automated test systems

Deqp test modules can be integrated to automated test systems in multiple ways. The best approach depends on the existing test infrastructure and target environment.

The primary output from a test run is always the test log file, i.e. the file with an `.qpa` postfix. Full test results can be parsed from the test log. Console output is debug information only and may not be available on all platforms.

Test binaries can be invoked directly from a test automation system. The test binary can be launched for a specific case, for a test set, or for all available tests. If a fatal error occurs during execution (such as certain API errors or a crash), the test execution will abort. For regression testing, the best approach is to invoke the test binaries for individual cases or small test sets separately, in order to have partial results available even in the event of hard failure.

The deqp comes with command line test execution tools that can be used in combination with the execution service to achieve a more robust integration. The executor detects test process termination and will resume test execution on the next available case. A single log file is produced from the full test session. This setup is ideal for lightweight test systems that don't provide crash recovery facilities.

Command line test execution tools

The current command line tool set includes a remote test execution tool, a test log comparison generator for regression analysis, a test-log-to-CSV converter, a test-log-to-XML converter, and a testlog-to-JUnit converter.

The source code for these tools is in the `executor` directory, and the binaries are built into the `<builddir>/executor` directory.

Command line Test Executor

The command line Test Executor is a portable C++ tool for launching a test run on a device and collecting the resulting logs from it over TCP/IP. The Executor communicates with the Execution Service (execserver) on the target device. Together they provide functionality such as recovery from test process crashes. The following examples demonstrate how to use the command line Test Executor (use `--help` for more details):

Example 1: Run GLES2 functional tests on an Android device:

```
executor --connect=127.0.0.1 --port=50016 --binaryname=
com.drawelements.deqp/android.app.NativeActivity
--caselistdir=caselists --testset=dEQP-GLES2.* --out=BatchResult.qpa
--cmdline="--deqp-crashhandler=enable --deqp-watchdog=enable
--deqp-gl-config-name=rgba8888d24s8"
```

Example 2: Continue a partial OpenGL ES 2 test run locally:

```
executor --start-server=execserver/execserver --port=50016
--binaryname=deqp-gles2 --workdir=modules/opengl
--caselistdir=caselists --testset=dEQP-GLES2.*
--exclude=dEQP-GLES2.performance.* --in=BatchResult.qpa
--out=BatchResult.qpa
```

Test log CSV export and compare

The deqp has a tool for converting test logs (`.qpa` files) into CSV files. The CSV output contains a list of test cases and their results. The tool can also compare two or more batch results and list only the test cases that have different status codes in the input batch results. The comparison will also print the number of matching cases.

The output in CSV format is very practical for further processing with standard command line utilities or with Microsoft Excel. An additional, human-readable, plain-text format can be selected using the following command line argument: `--format=text`

Example 1: Export test log in CSV format

```
testlog-to-csv --value=code BatchResult.qpa > Result_statuscodes.csv
```

```
testlog-to-csv --value=details BatchResult.qpa >
Result_statusdetails.csv
```

Example 2: List differences of test results between two test logs

```
testlog-to-csv --mode=diff --format=text Device_v1.qpa Device_v2.qpa
```

Note: The argument `--value=code` outputs the test result code, such as "Pass" or "Fail". The argument `--value=details` selects the further explanation of the result or numerical value produced by a performance, capability, or accuracy test.

Test log XML export

Test log files can be converted to valid XML documents using the `testlog-to-xml` utility. Two output modes are supported:

- Separate documents mode, where each test case and the `caselist.xml` summary document are written to a destination directory

- Single file mode, where all results in the `.qpa` file are written to single XML document.

Exported test log files can be viewed in a browser using an XML style sheet. Sample style sheet documents (`testlog.xsl` and `testlog.css`) are provided in the `doc/testlog-stylesheet` directory. To render the log files in a browser, copy the two style sheet files into the same directory where the exported XML documents are.

If you are using Google Chrome, the files must be accessed over HTTP as Chrome limits local file access for security reasons. The standard Python installation includes a basic HTTP server that can be launched to serve the current directory with the "`python –m SimpleHTTPServer 8000`" command. After launching the server, just point the Chrome browser to `http://localhost:8000` to view the test log.

Conversion to a JUnit test log

Many test automation systems can generate test run result reports from JUnit output. The deqp test log files can be converted to the JUnit output format using the testlog-to-junit tool.

The tool currently supports translating the test case verdict only. As JUnit only supports "pass" and "fail" results, a passing result of the deqp is mapped to "JUnit pass" and other results are considered failures. The original deqp result code is available in the JUnit output. Other data, such as log messages and result images, are not preserved in the conversion.

Android CTS Integration

Duplicating runs without CTS

To replicate the CTS run, install the deqp APK of the CTS package and use the following command:

```
adb -d shell am start -n
com.drawelements.deqp/android.app.NativeActivity -e cmdLine "deqp
--deqp-case=dEQP-GLES3.some_group.* --deqp-gl-config-name=rgba8888d24s8
--deqp-log-filename=/sdcard/dEQP-Log.qpa
```

The important part of that command is the following:

```
--deqp-gl-config-name=rgba8888d24s8
```

This argument requests that the tests be run on an RGBA 8888 on-screen surface with a 24-bit depth buffer and an 8-bit stencil buffer. Also remember to set the desired tests, e.g. using the `--deqp-case` argument.

Mapping of the CTS results

In the Android CTS, a test case can end up in three states: passed, failed, or not executed.

The deqp has more result codes available. A mapping is automatically performed by the CTS. The following deqp result codes are mapped to a CTS pass: `Pass`, `NotSupported`, `QualityWarning`, and `CompatibilityWarning`

The following results are interpreted as a CTS failure: `Fail`, `ResourceError`, `Crash`, `Timeout`, and `InternalError`