CryTest is a feature-testing framework aiming for large and complex test cases in CRYENGINE.

Feature tests can be written in-place along with code being tested, or in a separate source file. The basic interfaces are defined in the **CryTest.h** header. Under the hood, test instances are global variables that auto-register themselves, meaning they should only be in **.cpp** files and **not headers**.

Creating a Simple Test

Simple tests are those that complete the test within a function call. Defining a simple test is as easy as putting all the routines inside a CRY_TEST function scope. When the test framework executes the test, it blocks the main thread until the test is finished; calling a time-consuming function from the test is still possible, but the engine - the game or the editor - would be blocked during that time.

```
// This is a test case
CRY_TEST(YourTestNameHere)
{
        // you can use test asserts here, e.g. CRY_TEST_ASSERT
}

// Optionally, it's possible to group similar tests together
CRY_TEST_SUITE(YourTestSuiteNameHere)
{
        CRY_TEST(YourTestNameHere1)
        {
        }
        CRY_TEST(YourTestNameHere2)
        {
        }
        //and more...
}
```

Creating a feature test

Commands

The key concept of feature testing is first waiting for other parts of the program to update their states and afterwards, examining these states. To do this without any blockage, test commands for a feature test can be queued instead of executing these commands right away.

A command represents a piece of work that may take time to update itself or listen to other event changes.

A minimalistic example is as follows:

```
void ShowConsole
{
	gEnv->pConsole->ShowConsole(true);
}
void HideConsole()
{
	gEnv->pConsole->ShowConsole(false);
}
CRY_TEST(ConsoleOpenCloseTest)
{
	commands =
	{
		ShowConsole,
		CryTest::CCommandWait(1.f),
		[] {
			CRY_TEST_ASSERT(gEnv->pConsole->IsOpened());
		},
		CryTest::CCommandWait(1.f),
		HideConsole,
		CryTest::CCommandWait(1.f),
		[] {
			CRY_TEST_ASSERT(!gEnv->pConsole->IsOpened());
		}
	};
}
```

Commands are a collection of **CCommand**, i.e. the base type of any command. **CCommand** is not a base class of those commands, but instead it is the type-erased container of any class implementing these methods:

```
class CMyCommand
{
public:
	//! The main loop to trigger the command's work. Must not block in
the function. Returns true when the command is done.
	//! The function is required for converting the object to CCommand.
	bool Update();
	//! Returns additional commands to be run after this one is done.
The new commands are inserted to the front of queue.
	//! The function is optional for converting the object to CCommand.
	std::vector<CCommand> GetSubCommands() const;
};
```

Note that only the function **Update()** is required, whereas **GetSubCommands()** can be omitted if unnecessary.

- **CCommand** accepts free functions and lambdas.
- Member functions can be converted to **CCommand** using **CCommandFunction**, or a lambda with capture.
- The test framework comes with several predefined command classes in **CryTestCommands.h**, containing one of the most important command - **CCommandWait**.

Note that the waiting time here refers to the engine main loop time, which occasionally can differ from the real time due to blocking and spikes.

Attributes

The same test in the aforementioned example can be extended with some important attributes that denote when and where the test should be executed.

Attributes are actually the members of the class **CTestFactory**; jump to the symbol from the attribute name to learn more about it.

```
CRY_TEST(ConsoleOpenCloseTest, timeout = 60.f, game = true, editor = false)
{ /* ... */ }
```

- **game (bool) -** Defines whether the test should be executed for the game project. For more information, please refer to the Test Runner reference page.
- **editor (bool) -** Defines whether the test should be executed for the editor.
- **timeout (float) -** Indicates the time limit in seconds that the test should not exceed. Timed out tests will fail and display **time out** as the reason.

Running tests

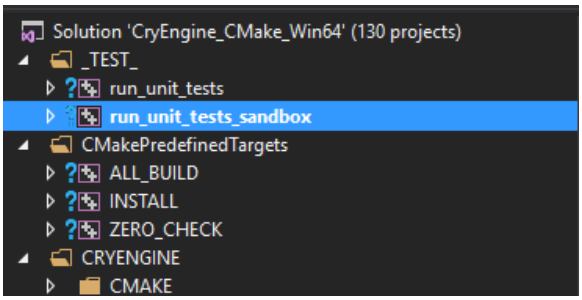
Currently, there are 4 ways to execute tests for a local engine build.

1. Running Test Target from Visual Studio (CRYENGINE solution)

The first option is to run the targets directly on a CryEngine solution that is either of the game project or the Sandbox itself.

Start a new debugging instance or set it as startup project to run. It internally starts the game launcher with the command line parameter, **run_unit_tests**.

Likewise, there is a test batch file for the game and the Sandbox for each platform, which can be found in *Tools\Tests\Test_RunUnitTests_*.cmd*.



Running tests from the solution explorer

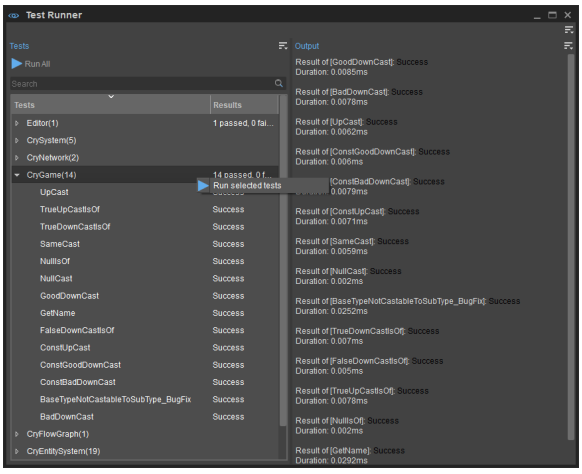
2. Using Test Runner on CRYENGINE Sandbox

The preferred way to run tests for the game developers should be the **Test Runner** tool which can be found in the Sandbox menu, on **Tools Advanced Test Runner**.

With the Test Runner you can:

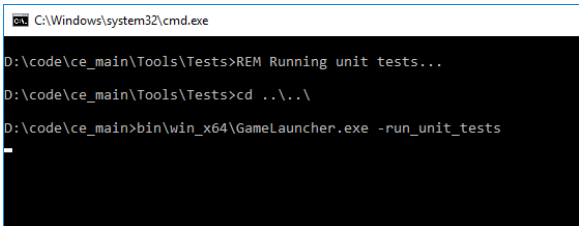
- Run all the tests that have been introduced to the tool;
- Run a specific test or a category of tests.

Note that when a test crashes or hangs, it inevitably crashes or hangs the Sandbox as well. It is recommended to write the test code "defensively", i.e. check for nulls.



Test Runner

3. Run Test Batch File (Tools\Tests\Test_RunUnitTests*.cmd)



4. Console command "CryTest"

The last option is to use the console command **CryTest** on the in-game console or the editor console.

The command supports auto complete, meaning if you press **Tab** after **CryTest**, it's going to display a list of valid test cases.