White Paper

Testing of Gaming Applications

By Somasekhar CH
Tavant Technologies, Inc.
3101 Jay Street
Santa Clara, CA 95054, USA
T: (866) 9-TAVANT
E: bizdev@tavant.com
W: www.tavant.com
Overview

This document provides an overview of various approaches/techniques that can be adopted for testing gaming applications, especially those developed in C++.

An extremely methodical and planned approach needs to be followed for testing each component of a gaming application individually. This can also help effectively determine cost allocation between the development and testing processes for the gaming application.

Testing of Gaming Applications

A structured approach toward testing of gaming applications ensures:

> Improvement in test coverage: Tests are designed to ensure that defects are detected before the application is launched.
> Adherence to process: Development of a stable, predictable process for game testing.

An effective test strategy to achieve success in testing of gaming applications may include the following techniques:

> White Box Testing
> Integration Testing
> Black Box/System Testing

White Box Testing

Programmed code verification testing activities should be executed at the very outset of gaming application development in C++. This may even include testing during the prototype phase.

The amount of testing at various stages must be planned and specified in terms of coverage types and levels (%), with the objective of delivering code items with near-zero defects. At the same time, costs should be so controlled as to be in congruity with the integrity level and phase of the overall project.

Unit Test Level

The more complicated the structure of a class and interrelationships among its different members, the more complex will it be, as against a class with a single
function, or sub-routine. Achieving full test coverage for some members of a class may be difficult, more so for the private member functions of a class because they are not directly visible or accessible outside of the class.

An effective way to achieve full test coverage for all members of a class is to structurally modify a class for testing, by making all its members externally accessible, without making any temporary modifications to it. This is because any temporary modifications to a class may make problems obscure, which should ideally be detected through unit test, and also lead to new problems.

Temporary (or even permanent) modification of a class to facilitate testing would effectively alter its entire context, defeating the very aim of its initial design -- encapsulation of details.

The test method could be a public member function of the class. However, it is better treated as a friend function of the class (as in example), because it actually instantiates objects of the class rather than acting upon a particular object. A number of classes may have the same test method as their friend function. This allows the private members of any of those classes to be accessed through the test method, which is their friend function. This can prove to be extremely useful during higher levels of testing and integration.

Example - Private members and the test function:

class example_class
{
    // Public member functions public:
    int add(); // sums x and y
    int subtract(); // subtracts y from x
    void PrintLargest(); // Prints greatest of x or y

    // Test Method
    friend void test();

    // Private member functions
    private:
    int largest(); // Returns largest of x or y

    // Private member data
    int x;
    int y;
};
Testing of Gaming Applications

This essentially emphasizes the need to test each unit in the system, possibly building up larger tests by using previously tested units. A unit can be a module/C++ file or the class itself.

Pre-requisites:
The minimum requirement is the existence of a specification, preferably in written form, describing explicitly what each unit should perform.

Class Testing / Hierarchical Class Testing Technique

After testing of the base classes to establish that they are working reasonably well, each successive derived class is tested through an approach generally known as Hierarchical Integration Testing. This technique exploits hierarchical nature of inheritance for the test-related groups of classes by using the test information of a parent class for testing of a sub-class.

Initially, base classes having no parents are tested by designing a test suite that tests each member function individually as well as the interaction among member functions.

To design a test suite for a sub-class, an algorithm may be developed to incrementally update the history of the parent class to reflect the sub-class’s modified, inherited attributes as well as newly-defined attributes. Only the new or affected or inherited attributes of the sub-class are tested, while the parent class's test suites are reused, if possible, for testing. The inherited attributes of a sub-class are retested by analyzing their interaction with the newly-defined attributes.

Heap Leakage

Apart from this, we used to for concepts like checking for heap leakage. This process is followed when memory is allocated for use, but not released even when no longer required. Unit test must ensure that no heap space remains in a class at the end of each test.

Using a Test Script

For testing of a class, a file called test script is created, which contains the code for the test method specified in class declaration. Test cases within the test method should instantiate objects of the class type, provide data in order to achieve a desired path through member functions, and check data against the expected values at the end of each test case execution.
Test cases should be so designed as to achieve full coverage of all functions within a class. Initial test cases must use the public member functions, including constructors. The objective of this approach is to achieve as much test coverage -- both structural and functional -- of all members of a class as possible, without recourse to direct access to the private member functions. If necessary, further test cases may then directly access the private member functions to ensure full test coverage.

Each test case should be so designed as to work in isolation from the other test cases. A test case normally starts by instantiating an object of the class type and finishes by deleting it. The method to be tested will be called after the class member data is set up. This may be done directly through the test script or indirectly by calling the previously tested methods. All class data should also be checked before deleting the object.

Integration Test Level

In a haste to produce a working system, there may not be sufficient time to test it at the unit level. In such a case, a reasonable compromise is to defer formal testing to the cluster level, where a cluster is a group of classes. In a multi-threaded application, testing at the thread level -- usually a stage higher than clusters -- might be more beneficial. However, irrespective of the level of testing chosen, there must be specifications for whatever is to be tested.

In a large object-oriented system, software is typically divided into sets of functionally related classes, commonly termed as class categories. Class categories may be functionally tested in isolation from one another, using class stubs created for unit level testing when simulation is required.

For a system consisting of only a few class categories, system integration can be performed in a single 'big-bang' stage -- integrating all class categories together, once they are fully tested.

An initial integration build of the system might consist of a few fully tested class categories, with the rest of the system being simulated. Functional tests on the simulated part of the system can then be performed to eliminate problems. Integration progresses with subsequent builds, replacing the simulated classes with their real implementations.

This approach to integration is very flexible with regard to timescales; it allows integration even before all class categories (or even classes) are fully tested.
Software development professionals usually procrastinate formal testing at the application level. An obvious advantage of this is that users get an early indication of what they would get and how robust it would seem. However, it also has a drawback; due to non-execution of individual testing of its underlying components, robustness of the system will be obscure. Apart from normal testing of the application, the following processes may be used for checking robustness of the system:

**Compatibility Testing**

Compatibility testing is conducted to evaluate an application's compatibility with the computing environment. Usually, it is required only for PC titles, and testers assigned to this task are expected to have, at least, the basic knowledge of PC hardware. Testers are provided with multiple PCs and/or extra parts, and expected to cross-test different hardware configurations with different operating systems.

**Configuration Testing**

Configuration testing is the process of testing a system with each software / hardware configuration that is supported.

**Game Compliance Testing**

Game (engine/rule) needs to be tested in accordance with the International Game Rules and Regulations (if applicable).

**Play/Logic Testing**

A “playtest” is the process through which a game designer tests a new game for bugs before it is launched into the market. Playtests are very common for computer games, board games and role-playing games, for which they have become an established part of the quality control process. Playtests can be run "open," "closed," "beta," or otherwise.

- **Open**: A playtest is considered as open if it allows anyone to join in the process or if external testers are recruited by a company for the testing process.
- **Closed**: A closed playtest consists of an internal testing process, which does not allow external people to join in.

- **Beta**: Beta testing usually refers to the final stages of testing a product, just before it goes to the market. Beta testing is usually run semi-open, with a limited form of the game, in order to detect any last-minute issues.

Apart from the above processes, extensive usability testing is required to check art and animation effects, different character attributes, user-friendliness, resolution, sound, synchronization and performance of gaming applications.

## Conclusion

Very often, people consider "game testing" and "game development tools and software testing" to be similar. However, this document refutes this notion and describes the overall test strategy that can be adopted for testing game tools and software. Testing of game development tools and software requires testing professionals with strong programming skills.

This document has been prepared on the basis of our experience across various projects, which can be referenced right from the initial stages of a product lifecycle.
About the Author

Somasekhar CH is Manager - QA at Tavant Technologies. He has over 10 years of industry experience in testing a wide range of game development tools and software.

About Tavant Technologies

Tavant Technologies is a specialized IT solutions & services provider that leverages its expertise to provide impactful results to its customers. We have leveraged our unrivaled capabilities and domain insights to create game changing results for leading businesses across chosen industry micro-verticals. We are known for our long-lasting customer relationships, engineering excellence and passionate employees. Founded in 2000, we are headquartered in Santa Clara, California and service customers across North America, Europe, and Asia-Pacific.