7. International Immersive Projection Technologies Workshop
9. Eurographics Workshop on Virtual Environments (2003)
J. Deisinger, A. Kunz (Editors)

Automated Testing of Virtual Reality Application Interfaces

Allen Bierbaum, Patrick Hartling, Carolina Cruz-Neira

allenb, patrick, carolina@vrac.iastate.edu
Virtual Reality Applications Center, Iowa State University

Abstract

We describe a technique for supporting testing of the interaction aspect of virtual reality (VR) applications. Testing is a fundamental development practice that forms the basis of many software engineering methodologies. It is used to ensure the correct behavior of applications. Currently, there is no common pattern for automated testing of VR application interaction. We review current software engineering practices used in testing and explore how they may be applied to the specific realm of VR applications. We then discuss the ways in which current practices are insufficient to test VR application interaction and propose a testing architecture for addressing the problems. We present an implementation of the design written on top of the VR Juggler platform. This system allows VR developers to employ standard software engineering techniques that require automated testing methods.

Categories and Subject Descriptors (according to ACM CCS):
I.3.7 [Computer Graphics]: Virtual reality
D.2.5 [Software Engineering]: Testing tools

Keywords: VR Juggler, Extreme Programming, unit testing

1. Introduction

Virtual Reality (VR) applications are being developed and used in a wide range of domains. These applications provide new insight into many difficult problems and offer advanced interaction techniques that may not otherwise be available. Unfortunately, developing VR applications is still difficult. Development of VR applications requires handling the many low-level details of VR systems while simultaneously managing the variety of human-computer interface requirements for immersive applications.

This complexity makes it infeasible to "get it right" the first time an application is developed. Instead, VR applications are developed using an iterative process of development and evaluation. Such an iterative methodology allows the application developers to refine the application as it evolves and gradually accommodate new requirements. This parallels many widely used software engineering techniques[1, 3], but in the case of VR application development, the evaluation aspect is more involved than in other application domains.

The evaluation process for VR applications usually focuses on four areas: system performance, usability, value for the task, and correctness[5]. System performance can be evaluated by the developers as new features are added. Usability and value for the task are evaluated qualitatively by the developer on a daily basis and more formally using user tests at key points during development. Correctness is normally evaluated by systematically testing the application for the expected responses.

Current correctness testing methods used to evaluate VR applications include manual tests of the application's user interaction and automated tests of the internal application components. During a manual test, the developer runs the application to test the behavior of the user interaction for correctness. Each time a manual test is performed, a basic routine must be followed. First, the developer starts the application with any required configuration options needed for testing. Next, the developer provides some amount of interaction to reach the target state that will be tested. Finally, the developer manually interacts with the application to see if it responds as it should. This process can become very time consuming and error prone if it must be followed every time a feature is added or a system test is required. As applications become more feature rich and complex, the problems with this testing methodology worsen, and systematically testing every feature of the application interface manually becomes infeasible.

Requiring manual testing has not been a serious problem in the past because most VR applications have been either research testbeds or small, highly focused applications. As VR applications have grown to become full production applications, they have increased in size and complexity. As production applications, there exists the development requirement that the application be robust. This means that all features must be guaranteed to function correctly as development continues.

We propose that VR application developers should turn to standard software engineering testing techniques to reduce the burden of testing while also increasing the robustness of the produced code. Many software engineering techniques rely upon automated testing. Automated testing allows developers to test an entire code base in a rapid, reliable, and convenient way.

Automated testing can be applied to VR applications, but there are currently limits to its applicability. Developers can directly apply unit testing to VR applications. Unit tests are small tests that check the behavior of the subsystems and the low-level classes. For example, unit tests could be used to ensure that a matrix math class executes correctly or that an intersection algorithm builds an internal data structure correctly. When testing user interaction, however, it becomes difficult to write automated tests because there is no direct way to test the interaction code in VR applications. Thus, the existing techniques employed for automated testing of desktop interaction cannot be used.

The interaction code in a desktop application is frequently captured by a widget library. A widget library provides a collection of common graphical interface objects that can be used by an application to create a graphical user interface (GUI). Examples of widgets include buttons, sliders, and text entry fields.

Consider a graphical application interface with a button widget. When a user interacts with the application by pressing the button widget, an interface handler within the widget library processes all the device input (i.e., mouse and keyboard). Since the handler knows the location of the button widget, it can recognize the interaction with the widget. The handler converts this interaction into a widget action event that is delivered to an application callback method.

Common techniques for testing this type of application have arisen because the application author is only concerned with implementing widget callback methods. Developers do not have to worry about writing the code to process device input because this is handled by the widget library. Developers write unit tests that simulate interaction by making the same calls into the application back-end that the widgets would trigger at runtime. This is done under the assumption that the developer has configured the user interface callbacks and that the widget library will process the device interaction correctly. As long as the interaction handler behaves, the developers can be assured that, at run time, the application will receive the correct calls.

VR user interfaces are more complex and require a different type of testing. VR applications do not make use of a common interaction handler to process device data but instead process the device input directly. Much of the core application logic revolves around how to process the input to support immersive interaction. Furthermore, it may be possible for the user to interact directly with every object in the virtual environment. To allow for this degree of interactivity, the application code for each object processes the input data and responds accordingly.

To illustrate the complexity further, consider an example of a small application that allows a user to grab and reposition a cube in a virtual environment. The user interface code is responsible for letting the user navigate to the cube and for providing interaction capabilities with the cube. Interactions could include reaching out and touching the cube, grabbing the cube, moving the cube, and finally releasing the cube. The interaction code in this example is not hidden within a widget library. Developers of immersive applications generally do not have common widget libraries available. This is because the open-ended options for immersive interactions preclude the creation of standards which would limit the available interaction methods. Instead, the logic for object grabbing is part of the application code, and it requires processing of the device data and examination of the data model to test the current input state against the current object state.

To automate the testing of VR applications, we need a way to automate the testing of the interaction code of the application. Presently, there is no clear way to automate this type of testing because current methods focus on functional unit tests instead of interaction tests.

1.1. Contributions

This work focuses on how to allow the creation of automated interaction tests for VR applications. These tests can be used as unit tests, acceptance tests, and regression tests throughout the development and evaluation of an application. This paper makes three main contributions:

- We identify the need for automated testing of interaction code in VR applications and describe how it can be used to create more robust applications.
- We describe the unique challenges that must be addressed to automate testing of VR applications and propose methods for overcoming these issues.
- We present the design and implementation of a working system that demonstrates our testing solutions and discuss our experiences with using it.

2. Background

Testing is the process of detecting errors in software[4]. In this section, we provide background information about testing software including the types of tests, the importance of testing, and some testing methodologies. We conclude this section with a review of some existing tools for automating testing.

2.1. Types of Tests

There are two well-known types of tests used in software engineering: *unit tests* and *acceptance tests*. These play different roles depending on the test author and the desired results.

2.1.1. Unit Tests

Unit tests are written by the programmers of a software system to test individual routines, components, or modules (i.e., software units)[1, 4]. Unit tests do not test the system as a whole. For a given class, a collection of unit tests for the member functions ensure that the class code executes properly. This is also described as "confidence testing" because it provides a degree of confidence that the code in question will work. Furthermore, it provides a means to assess the reliability of the software.

Unit tests can be used for a variety of purposes toward ensuring proper functionality of a piece of software. An obvious use would be to ensure that code functions properly under normal use cases. However, unit tests can also be employed to test behavior under abnormal or improper uses. In this way, programmers can test error handling and recovery.

2.1.2. Acceptance Tests

Acceptance tests, on the other hand, are written by customers or clients during the development process[†]. Such tests are written on a "story-by-story" basis[1]. Beck defines a story as "one thing the customer wants the system to do," and each customer-defined story should have a test. The results of these tests, however, are more complex to evaluate than those of unit tests. Instead of a pass/fail evaluation, acceptance tests are usually assessed based on some percentage. As the software is developed, the result of a given test should approach 100%. Ultimately, the purpose of acceptance tests is to demonstrate that the software system as a whole works as desired.

2.2. Importance of Testing

Testing code plays a critical role in ensuring that the software works as expected. By testing each new feature as it is added to an application, the developers can be sure that the new feature works. Within a unit testing framework, all the unit tests are run every time testing is performed. In so doing, testing a new feature also ensures that existing functionality does not degrade as a result of the addition. Furthermore, a programmer can run the tests at any time and know whether each element of the system is working.

2.3. Testing Methodologies

There are many ways to apply the types of testing described above. We focus on the testing processes defined by two programming methodologies that are currently in wide use: Extreme Programming and the Unified Process model.

2.3.1. Extreme Programming

In the Extreme Programming (XP) testing methodology[1], programmers follow a test-driven approach to development. They work in a short cycle where tests are added and then made to work. There are two key points regarding how tests should work:

1. No two tests should interact with each other
2. Testing should be automatic

By keeping the tests distinct, we can avoid problems where the failure of one test causes other tests to fail, thereby resulting in false negatives. By automating the tests, we can be assured that the tests will give the accurate results without being affected by outside factors such as stress levels or time restrictions.

2.3.2. Unified Process Model

The Unified Process (UP) model indicates that testing should begin when the software architecture is defined and continue throughout the complete software life cycle[3]. Testing in the UP model is based on regression tests that are used to ensure that previously tested code still works when new code is written. During the life cycle, the number of regression tests will increase steadily.

In the UP model, a test model describes how system components are tested. The test model is made up of test cases, test procedures, and test components. Test cases are designed to test specific use cases in the project design. As compared to the XP methodology, testing is not part of the core development process and rather is separate from the design implementation.

2.4. Previous Work

Most previous work in the area of testing has focused on how to guarantee requirements of applications such as performance, response, accuracy, etc. The literature discusses methods for writing good tests and outlines development cycles for ensuring good incorporation of testing.

[†] Acceptance tests were formerly called functional tests in the Extreme Programming literature. "Acceptance test" is now the standard term.

2.4.1. XUnit

In the XP testing methodology, automated unit tests are at the center of test-driven development. XUnit is a testing framework based on the principles of XP, and implementations of it have been written for several different programming languages. These include JUnit[‡] for Java, CppUnit[§] for C++, NUnit[¶] for the Microsoft .NET Framework, and PyUnit[‖] for Python. Programmers using these tools write suites of unit tests, and a test runner manages the execution of the tests.

3. Issues of VR Interface Testing

In this section, we describe the primary issues involved in testing VR applications. First, we describe the differences between VR interface testing and traditional unit testing. This serves to identify the shortcomings of unit testing with respect to VR interface testing. We then relate this to how VR applications are currently tested and begin building up a set of requirements for an automated testing system that can effectively test VR interfaces. We finish by giving a high-level overview of our proposed solution to automated testing.

3.1. VR Interface Testing Compared to Unit Testing

As we described above, traditional unit tests systematically test the modules and the components of a software system. For object-oriented designs, this commonly means that each method of a class is tested individually. The test author devises inputs for each method and then verifies that the method behaves correctly when called. This may involve checking the return value of the method or verifying a computation triggered by the method. The author systematically varies the input in an attempt to test all the use cases and error states that the object might encounter in a real application. In this way, the unit test provides all the input states for testing an object. These same unit testing methods can be applied to higher level modules and subsystems to test groups of collaborating objects. Instead of testing individual methods, high-level unit tests operate by testing the results of test scenarios composed of multiple calls to related objects.

In the specific area of desktop GUIs, testing methods do not normally test the GUI explicitly. Although it is possible, in many cases it is not necessary. This is because traditional GUIs are built using interaction toolkits that provide the GUI widgets for the interface. These toolkits manage the processing of the device input and trigger event handling callbacks when a GUI interaction occurs. As a result, unit tests can be designed that manually create the directly invoke the event callbacks and test the results.

Testing VR applications differs from this because much of the VR application code handles user interaction. Most VR applications do not make use of an interaction toolkit, instead they typically process device input directly. Hence, it may not be possible for the test author to deliver input to the application programmatically. In other words, if a GUI application were written to process input directly from the mouse rather than through GUI events, the testing methods described above for graphical applications would be insufficient. In order to test the interaction in a VR application, there must be some way to make the application respond as though it is receiving actual device input normally.

Test cases for VR must be able to test the application's interaction code. When testing interaction code, a unit test should not directly set the application state since this bypasses the code it is trying to test. Instead, the unit test needs to trigger the interaction code by providing input to the application. The application's interaction code then changes the state of the application in response to the input. Once all of the input has been processed by the interaction code, the application's state can be tested for correctness. For example, a navigation test may require the application to respond to a series of user inputs that navigate through the virtual world. Once the input has been processed, the test can verify that the application is at the expected destination coordinates in the virtual world.

To support testing in this way, tests need a way to monitor the state of input processing. The tests need to know when input processing has proceeded up to a predefined point. We call these input processing points *checkpoints*, of which there may be many in an application. A VR interaction test asynchronously monitors the state of the input processing waiting for a checkpoint to be reached. When the checkpoint is reached, the test proceeds to verify that the application has reached the correct state.

3.2. Design of a VR Interface Testing Framework

A potential design for a testing framework for VR interfaces is shown in Figure 1. The design uses a test runner that maintains a list of tests that it manages. Each of these tests externally monitors the state of the VR application, while the application is being controlled by pre-recorded input data. When the application reaches a state that requires testing, the test becomes active and checks the application for the correct state. The next sections go on to describe this high-level design in more detail and discuss the reason for this design structure.

3.2.1. Recorded Input

To test a VR interface, there must be some input for the application to process. In order to automate the testing procedure, we must be able to provide input to the application

‡ http://www.junit.org/

§ http://cppunit.sourceforge.net/

¶ http://nunit.sourceforge.net/

‖ http://pyunit.sourceforge.net/

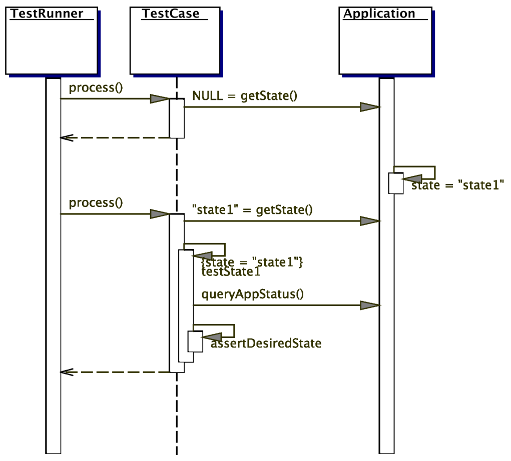

Figure 1: *Test monitoring an application*

in such a way that it responds exactly as it would if the user were interacting with it directly. We can achieve this by recording all device input during an application usage scenario. At key stages during the usage (checkpoints), the input is "stamped," and user-defined aspects of the application state are stored. This log of the input data and application state can then be used later as the basis of a test case.

3.2.2. Test Case Execution

In order to test the VR application, a test case must be created to monitor the application. When the test case begins its execution, it loads the recorded input into the system. This device input is played back for the duration of the test. Periodically, the test case queries the input playback system and the application to get the current state of both. When the application reaches some checkpoint, the test case verifies that the application state matches the previously stored state. If the application state is incorrect, then the test case signals the test runner, and the test runner alerts the user of the failure.

Checkpoints during the application execution can be identified by monitoring the state of the input, by testing the state of the application, or by using a flag in the input playback stream. In all three cases, the test case determines if it has reached a checkpoint by querying the application. The `getState()` application method in Figure 1 is a placeholder for this operation. In this case, a checkpoint is reached when the query operation identifies "`state1`". At such time, the procedure described above is performed to verify that the application state matches the expected result.

The checkpoints within a single test case are arrived at sequentially. As such, the second checkpoint builds off the first state and so on. This leads to the issue that test cases should consist only of test states that can build off each other in sequence.

3.2.3. Test Case Grouping and Management

The test runner is used to manage a group of test cases. It allows groups of tests to be run in sequence with each of the individual tests executed in sequence. The test runner also tracks the results of the tests and is responsible for synchronizing the tests and the application.

When a test starts, it should be guaranteed that the application is in its initial state. The test runner ensures that this is true by calling an application reset method between tests. When the new test starts, it could modify the application state to prepare for the test. For example, the test case may ask the application to load specific data or to start in a specific mode.

When all tests have completed execution, the application can query the test runner for the test results. At this point, the user reviews the results and makes changes to fix test failures.

4. Implementation

We have implemented a VR interface testing module based upon the ideas put forth in the previous section. This design should be applicable to any VR software system. Our implementation is an add-on module for VR Juggler[**2]. VR Juggler is a suite of reusable C++ libraries that form a virtual platform for VR application development. VR Juggler includes libraries for device management, operating system abstractions, and configuration. For this implementation, we extended the Gadgeteer module which is responsible for all input management within VR Juggler. We also added a new subsystem for test management.

Our design uses names for the interfaces and the classes that are similar to those used in CppUnit, the tool we use for testing C++ code not related to the interaction. This should allow developers who are already familiar with CppUnit to understand the functionality of our test system more easily.

The core of the design consists of a test runner that manages a group of tests, as shown in Figure 2. The test runner is responsible for keeping the entire testing module synchronized with the user application and for allocating processing time to the tests. Tests consist a sequence of state checks for a given scenario and any initialization and clean-up code required for that scenario. Test failures are triggered by throwing C++ exceptions that describe the failure. The test runner tracks the test results by storing these failures for later processing.

** http://www.vrjuggler.org/

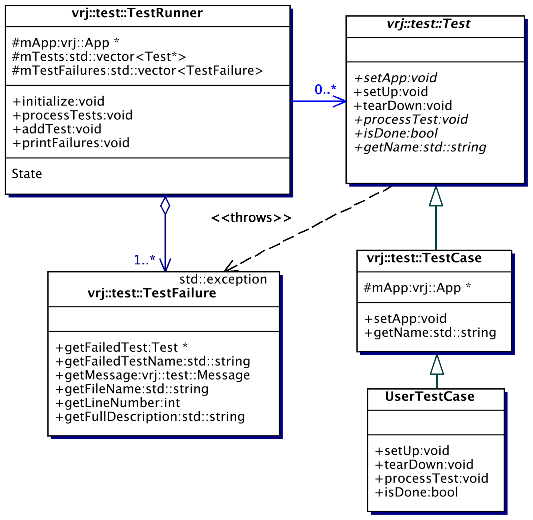

Figure 2: *Overview of test system classes*

```
class UserTestCase : public TestCase
{
virtual void setUp()
{
   inputLogger->play(“input.xml”);
   application->reset();
}
virtual void processTest()
{
   std::string chkpt =
     mInputLogger->getCheckpoint();
   if("selected" == checkpoint)
   {
      bool is_selected =
            mApp->mIsSelected;
      TEST_ASSERT(is_selected);
   }
   else if("moved" == checkpoint)
   {
      Pos cur_pos = mApp->obj.getPos();
      TEST_ASSERT(cur_pos == good_pos);
   }
}
};
```

Figure 3: *Test::processTest example code*

We will now describe each part of the system in more detail, explaining the responsibilities of each class in the testing module. This description will start at the low-level input logging and proceed up to the high-level test runner.

4.1. Input Logger

The implementation uses a module that we created for the VR Juggler Gadgeteer input management subsystem to provide input logging. This module extends Gadgeteer to allow for the recording, playback, and storage of entire sessions of user input. The input is recorded by collecting the current state of each device into a single image of the overall input. This image is created by serializing each active input device into an XML-based form. The serialized input data are then collected together to present the complete input image.

The input logger also provides the ability to store checkpoint identifiers in the logging data. As we described above, these can be queried during playback to detect that the input has reached some predefined state. The insertion of user-defined checkpoints offers a convenient mechanism for developers to create tests of key pieces of interaction functionality.

4.2. Tests

All tests within the system are encapsulated within objects. The `vrj::test::Test` class is the base class for all the test objects within the testing system. It is an abstract base class that defines the interface that all tests must implement to be managed by a test runner. The interface provides a `setUp()` method that is invoked to perform any one-time setup and initialization at the beginning of the test execution. There is a corresponding `tearDown()` method that is called by the test runner when the test has completed execution. The `processTest()` method is called by the test runner once per frame. This is where the code for performing the tests should be located. The code in this method checks the current state of the application and when applicable, it performs tests to verify that the application is in the expected state. See Figure 3 for an example implementation of this method.

The `vrj::test::TestCase` class is designed to be the basis for end-user test cases. This class derives from the `vrj::test::Test` class and extends it to add tracking of the user application being tested. In the current implementation, both test classes could be combined into one, but we plan to add several other customized test classes that will derive from `vrj::test::Test`. For example, we plan to add a test suite class that will allow grouping of multiple tests into a single parent test.

4.3. Failures

The testing system processes test failures using C++ exception handling. When a test failure is detected, the test code throws an exception. The `vrj::test::TestFailure` class is the base class for the test case failure exceptions. This class holds all the relevant information about the test failure including the failed test name, the file name where the failure occurred, the line number in the file, and an ex-

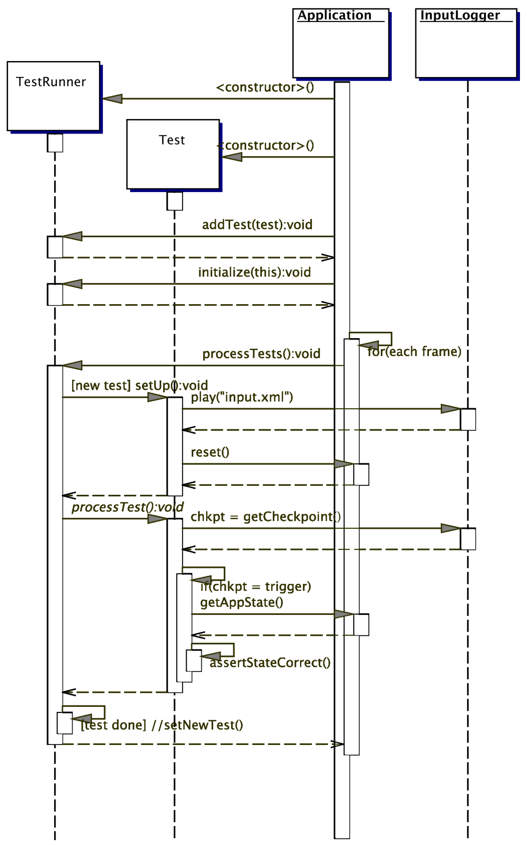

Figure 4: *Test execution*

tended description of the failure. The interface requires that each of these parameters is set at construction time and allows the parameters to be queried by code that catches the exceptions.

To simplify the creation of the failure exceptions, we provide helper macros that take a condition to be tested. If the condition fails, the helper macros throw an exception containing information about where the failure occurred and what caused it. An example macro is `TEST_ASSERT` which is used in Figure 3.

4.4. Runner

The `vrj::test::TestRunner` class is responsible for executing the tests and collecting the failure information. It synchronizes the tests with the application. It is the responsibility of the application to create, initialize, and execute the runner (when testing is enabled).

The interaction and behavior of the test runner and the application can be seen in Figure 4. If an application wants to run tests, it must first create a test runner. Once the application creates a test runner, it add tests to it with the `addTest()` method. When the application has added all the desired tests, it must call the test runner `initialize()` method, passing in a reference to itself. This completes the initialization of the individual test cases and prepares the test runner for execution. The application is then responsible for calling the `processTests()` method once at the beginning of each frame. By doing this, the application gives the test runner processing time to run the current tests. The test runner will manage running the tests and collecting the status information of any failures. When all the tests are completed, the runner will notify the application, and the application can invoke the `printFailure()` method to print the results of the test runs.

5. Discussion

We are currently using this testing system to support application development at the Virtual Reality Applications Center. The testing system described in this paper has allowed us to more fully apply standard software engineering methodologies such as those described in the background material. This has led to an increase in application developer productivity and to more reliable applications. We can make use of iterative development techniques to add new features quickly while remaining confident that existing functionality is maintained.

5.1. Test System in Use

We have used this system to add tests to several VR applications. For application testing, we test the interaction code for correct behavior while performing interactions such as navigation and object manipulation. In each of these applications, we extended the VR Juggler application object to add support for testing. First, we added a new data member to hold a reference to the test runner and a new method to initialize the tests when requested (see `initTesting()` in Figure 5). Then, we added the code to `preFrame()` to call the test runner's `processTests()` method. Once test processing has been added to the application, we must create the tests and record the input needed for playback during testing.

As can be seen in the example shown in Figure 5, existing applications can be extended to support testing with very little new code. Because the application is monitored "as is," there is no need to modify the application in any other way to allow for monitoring.

5.2. Designing for Testing

While we have found that it is possible to add interaction testing to existing application, it is most effective if the application is designed from inception to support testing. If an

```
void UserApp::initTesting
{
   testrunner = new TestRunner;
   testrunner->addTest(new Test1);
   testrunner->addTest(new Test2);
   testrunner->initialize(this);
}
void UserApp::preFrame()
{
   ...
   if(NULL != testrunner)
      testrunner->processTests();
}
```

Figure 5: *Extending an application to support testing.*

application has not been designed to support testing, it can be difficult to check its state externally. Applications that are designed for testing will provide hooks for external querying the application state. They may also provide methods to store the state for later comparison during testing.

We have found this last method to be especially helpful when creating input logs for debugging. Consider what the user must do when the application state cannot be captured and stored. While logging the input, the current state of the application must be recorded manually. This may mean that when the user stamps the input log with a checkpoint, the user must also call a routine that prints out the current state of all the objects in the system. The user then has to manually look up the state information for the object to be tested, write down this state information, and then add code for a test that asserts this state when the given checkpoint is reached.

In the case where the state can be queried and saved, the application can instead just save its state to a file. Then, the test case can load this file and compare the stored state against the active state at the time of the checkpoint.

We are currently experimenting with ways to simplify test creation even further by extending the tests to include a *collection mode*. While running tests in collection mode, the tests store the current status of the application to be used for a later check. This can greatly simplify the maintenance of test cases by allowing test data to be reconstructed rapidly when application design changes break tests.

We are also experimenting with standard ways to store the application data in an XML format so that it can be incorporated into the input log with the testing checkpoints. This builds upon the concept of saving the application state to a persistent form. In this implementation, though, the persistent form is an XML tree that is saved as an extra property of the checkpoint entry. This alleviates the need for the developer to maintain an input log file and several separate files with views of the system state. Instead, all this information is combined into a single unified file.

5.3. Use with Higher Level Tools

There are many higher level VR authoring tools that could benefit from testing. High-level tools can offer users built-in testing features by building upon a low-level system that provides testing capabilities. The high-level programming interface simply needs to provide a way for application authors to access the testing system. This could be provided using a pass-through interface or by creating a customized interface that wraps the low-level testing facilities. The details of how these interfaces would work are specific to the actual high-level tool being used.

6. Conclusions and Future Work

We have described why automated testing of VR interaction code is needed and how it differs from current unit testing methods. We have also described the unique challenges that interaction testing presents and have described the design and implementation of a system that addresses these issues.

This testing system allows developers to test their applications more completely than was previously possible. It provides for a solution that allows testing of VR application interfaces to be automated.

We have several areas of future work that we would like to pursue. We are working on methods to simplify the creation and maintenance of the tests. We are also working on methods to help automate the process of creating the tests. The discussion of storing the application status in XML form in the input log is an example of this research direction.

We are also experimenting with ways to test a wider variety of VR applications. One area that we are particularly interested in is testing of collaborative VR applications. This presents many interesting new challenges for testing, but we are confident that solutions exist.

In conclusion, we have found this system to be a very powerful tool for creating automated testing systems for VR applications. This system allows developers to employ standard software engineering techniques that require unit testing methods.

References

1. K. Beck. *Extreme Programming Explained.* The XP Series. Addison-Wesley, 2001.
2. A. Bierbaum. Vr juggler: A virtual platform for virtual reality appliation development. Master's thesis, Iowa State University, Ames, IA, 2000.
3. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1998.
4. S. McConnell. *Code Complete.* Microsoft Press, 1993.
5. R. Stuart. *The Design of Virtual Environments.* McGraw-Hill, 1996.