# Clementine Integration Test Document

## Project Clementine Group

### Table of Contents

1. Introduction ................................................................. 4
2. White Box Testing .............................................................. 5
   2.1. Core ........................................................................ 5
       2.1.1. Input Devices ....................................................... 5
           2.1.1.1. InputDeviceFactory ........................................ 5
           2.1.1.2. InputDevice ................................................. 5
       2.1.2. Data Aggregation .................................................. 7
           2.1.2.1. Input Point Data Table .................................... 7
           2.1.2.2. LinearRegressor inherits from Regressor .............. 8
               2.1.2.2.1. LinearRegressor sets Regressor::_RegressorType ........................................ 8
               2.1.2.2.2. LinearRegressor overrides unimplemented Regressor::UpdateModel .................. 8
               2.1.2.2.3. LinearRegressor uses new data when Regressor::UpdateModel is called .................. 8
           2.1.2.3. DataPoint with LinearRegressor and Regressor .......................................................... 9
               2.1.2.3.1. SmoothedValue .......................................... 9
               2.1.2.3.2. GetPredictedPoint ....................................... 9
           2.1.2.4. DataPointMarshal and Scene ................................ 9
               2.1.2.4.1. Scene and InputDevice have 1:1 relationship ......................... 10
               2.1.2.4.2. DataPointMarshal does not demote early ..................... 10
               2.1.2.4.3. DataPointMarshal demotes on time ......................... 11
               2.1.2.4.4. DataPointMarshal properly promotes .......................... 11
       2.1.2.5. Subscriber SubscriberMarshal and InputDeviceFactory ................................................. 12
               2.1.2.5.1. Subscriber inheritance and registration ...................... 12
               2.1.2.5.2. Subscriber request of InputDevice ............................ 12
               2.1.2.5.3. Subscriber change of InputDevice ............................ 13
       2.1.2.6. Subscriber SubscriberMarshal and Timer ................................................................. 13
               2.1.2.6.1. Subscriber can request a call .................................. 13
       2.2. Interfaces .................................................................... 14
       2.2.1. 3DInterface ............................................................. 14
       2.2.1.1. Compute3DPosition calculation testing .................................................. 14
2.2.1.1.1. Procedure ................................................................. 14
2.2.1.1.2. Pass Condition ......................................................... 14
2.2.1.2. InitializeDefaults variable set testing ................................. 14
2.2.1.2.1. Procedure ................................................................. 14
2.2.1.2.2. Pass Condition ......................................................... 15
2.2.2. Gesture Interface ........................................................... 15
2.2.2.1. Gesture and Vector Interaction ........................................... 15
2.2.2.1.1. Vector Concatenation ................................................. 15
2.2.2.1.2. Vector Computation ................................................... 16
2.2.2.2. Gesture and GestureInterface Interaction .............................. 16
2.2.2.2.1. Gesture Importing ....................................................... 16
2.3. Demos ............................................................................. 17
2.3.1. 3D demo ..................................................................... 17
2.3.1.1. ProcessTick thoroughness check ......................................... 17
2.3.1.1.1. Procedure ................................................................. 17
2.3.1.1.2. Pass Condition ......................................................... 17
2.3.1.2. ProcessTick timing accuracy check ..................................... 17
2.3.1.2.1. Procedure ................................................................. 17
2.3.1.2.2. Pass Condition ......................................................... 17
2.3.1.3. RenderableObject ....................................................... 18
2.3.1.3.1. ProcessTick thoroughness check ..................................... 18
2.3.1.4. BoundingBox ............................................................... 18
2.3.1.4.1. isCollidingWith functionality test .................................. 18
2.3.1.5. Player ...................................................................... 18
2.3.1.5.1. Update3DPotition functionality test ................................. 18
2.3.2. Gesture Interface Demo .................................................... 19
2.3.2.1. GestureInterpreter handles gestures properly. ....................... 19
2.3.2.1.1. moveMouse .............................................................. 19
2.3.2.1.2. mouseClick .............................................................. 19
2.3.2.1.3. holdKey ................................................................. 19
2.3.2.1.4. releaseKey .............................................................. 20
2.3.2.1.5. pressKey ............................................................... 20
2.3.2.2. Blender receives ClemX11 events ....................................... 20
2.3.2.2.1. mouse movement ...................................................... 20
2.3.2.2.2. mouse click ............................................................ 21
2.3.2.2.3. key press ............................................................... 21
2.4. Diagnostic Utility .............................................................. 22
2.4.1. Data Point Sets .............................................................. 22
2.4.2. GestureResult Sets .......................................................... 23
2.4.3. InputPoint Sets ............................................................... 23
2.4.4. Chart ........................................................................... 23
2.4.4.1. LocationChart and Chart ............................................... 24
2.4.4.1.1. Procedure ................................................................. 24
2.4.4.1.2. Pass Condition ......................................................... 24
2.4.4.2. ScrollingTimeChart and Chart ......................................... 24
2.4.4.2.1. Point pushing removes x dimension ............................... 24
2.4.4.2. Time advance enforcement ................................................................. 25
2.4.4.2.3. Max time back deletion ................................................................. 25
2.4.4.3. GestureChart and Chart ........................................................................ 25
  2.4.4.3.1. Confidence added to scrolling graph ............................................. 26
2.4.5. Perspective and Input Controllers ............................................................ 26
  2.4.5.1. 3DInterfacePerspective and Perspective with Charts ......................... 26
    2.4.5.1.1. SIGNAL/SLOT hookup and Chart Updates for 3D ....................... 26
  2.4.5.2. SpatialInterfacePerspective and Perspective with Charts .................. 27
    2.4.5.2.1. SIGNAL/SLOT hookup and Chart Updates for spatial .................. 28
3. Black Box Testing .......................................................................................... 29
  3.1. Input To Core ............................................................................................ 29
    3.1.1. Core Timer and InputDevice ............................................................... 29
      3.1.1.1. InputPoint time assigning for a Wii Remote Device ......................... 29
        3.1.1.1.1. Procedure .................................................................................. 29
        3.1.1.1.2. Pass Condition ......................................................................... 29
  3.2. Core To Three Dimensional Interface ......................................................... 29
    3.2.1. Core Subscription ................................................................................. 29
      3.2.1.1. Procedure ...................................................................................... 29
      3.2.1.2. Pass Condition .............................................................................. 30
  3.3. Core To Gesture Interface ........................................................................... 30
    3.3.1. Core Subscription ................................................................................. 30
      3.3.1.1. Procedure ...................................................................................... 30
      3.3.1.2. Pass Condition .............................................................................. 30
    3.3.2. Gesture Observation ............................................................................. 30
      3.3.2.1. Procedure ...................................................................................... 30
      3.3.2.2. Pass Condition .............................................................................. 31
  3.4. 3D Interface to Demo .................................................................................. 31
    3.4.1. Procedure ............................................................................................ 31
    3.4.2. Pass Condition ..................................................................................... 31
  3.5. Gesture Interface to Demo .......................................................................... 31
    3.5.1. Procedure ............................................................................................ 31
    3.5.2. Pass Condition ..................................................................................... 32
  3.6. Interfaces to Diagnostic Utilities ................................................................. 32
    3.6.1. SubscriberMarshall to 3DInterfaceInputController ............................. 32
      3.6.1.1. Procedure ...................................................................................... 32
      3.6.1.2. Pass Condition .............................................................................. 32
    3.6.2. SubscriberMarshall to SpatialInterfaceInputController ...................... 32
      3.6.2.1. Procedure ...................................................................................... 33
      3.6.2.2. Pass Condition .............................................................................. 33
Glossary ......................................................................................................... 34
1. Introduction

This document describes and tests how information is passed between objects and modules within the Clementine library. Each of the major modules for the Clementine library must be tested, both inside and out.

The first of these major sections is Core. Core is divided up into two sub-modules. One module for gathering input (input) and one for the aggregation and processing of that input (data aggregation).

The second and third major sections involve the interfaces and demos. These sections will ensure that all of the classes and dependencies within these modules work together as planned.

The document is divided up into two sections. The first section, white box testing, contains tests that ensure that the separate classes in each module communicate properly.

The second section, Black Box Testing, ensures that the modules that were individually verified in the white box tests can adequately communicate with each other. These tests will ignore most of the classes in each module favoring a more abstract approach, testing the module interaction rather than class interaction.

When combined, the white box tests and the black box tests will ensure that every entity in our library can communicate and interpret data properly. This will satisfy the end goal of this document.
2. White Box Testing

2.1. Core

2.1.1. Input Devices

2.1.1.1. InputDeviceFactory

2.1.1.1.1. Factory creates WiimoteDevice

This will test that the factory returns a WiimoteDevice when asked to.

2.1.1.1.1.1. Procedure

Create a InputDeviceFactory object. Call its getInputDevice(enum) method with WiimoteDevice as the enum.

2.1.1.1.1.2. Pass Condition

Method call returns an object of type WiimoteDevice

2.1.1.1.2. Factory creates WebcamDevice

This will test that the factory returns a WebcamDevice when asked to.

2.1.1.1.2.1. Procedure

Create a InputDeviceFactory object. Call its getInputDevice(enum) method with WebcamDevice as the enum.

2.1.1.1.2.2. Pass Condition

Method call returns an object of type WebcamDevice

2.1.1.2. InputDevice

2.1.1.2.1. Inheritance of InputDevice

Here we are ensuring that the individual input devices all inherit from InputDevice.

2.1.1.2.1.1. Procedure

An object of type WiimoteDevice is created.

2.1.1.2.1.2. Pass Condition

Casting it to type InputDevice succeeds.
2.1.1.2.1.3. Procedure

An object of type WebcamDevice is created.

2.1.1.2.1.4. Pass Condition

Casting it to type InputDevice succeeds.

2.1.1.2.2. Timer schedules timerTrigger

These tests will ensure that when an InputDevice is created, it will properly schedule a polling method.

2.1.1.2.2.1. Procedure

Create an object of type InputDeviceFactory and generate a device of type WiimoteDevice.

2.1.1.2.2.2. Pass Condition

The timerTrigger method is scheduled at a regular interval.

2.1.1.2.2.3. Procedure

Create an object of type InputDeviceFactory and generate a device of type WebcamDevice.

2.1.1.2.2.4. Pass Condition

The timerTrigger method is scheduled at a regular interval.

2.1.1.3. WebcamDevice

2.1.1.3.1. getImage

This will ensure that the getImage method will always return an image.

2.1.1.3.1.1. Procedure

A an object w of type WebcamDevice is created. w.getImage(); is called.

2.1.1.3.1.2. Pass Condition

It returns an object of type IplImage.

2.1.1.3.2. getBlobs

This will ensure that the getBlobs method will always return blobs.
2.1.1.3.2.1. Procedure

A an object \( w \) of type `WebcamDevice` is created. \( w \).getBlobs(\( w \).getImage()); is called.

2.1.1.3.2.2. Pass Condition

It returns an object of type `CBlobResult`.

2.1.1.3.3. `blobsToInputPoints`

This will ensure that blobs can be converted into `InputPoint`'s.

2.1.1.3.3.1. Procedure

A an object \( w \) of type `WebcamDevice` is created.
\( w \).blobsToInputPoints(\( w \).getBlobs(\( w \).getImage())); is called.

2.1.1.3.3.2. Pass Condition

It returns an array of objects of type `InputPoint`.

2.1.2. Data Aggregation

This portion of Core is responsible for taking the data assembled by the `input devices` and aggregating it in such a form that is easily readable by the rest of the library.

2.1.2.1. Input Point Data Table

This table holds input values that will be referred to during the following tests and used as test data.

The values in parentheses ("()") represent the `Point::X` and `Point::Y` values of the `InputPoint::_location` property. The value following the colon (":") is the timestamp to be given to that via the `InputPoint::_time` property. Separate `InputPoint` instances are then delineated by dashes ("—").

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPoint Set A</td>
<td>(0.2,0.1):100—(0.3,0.3):200—(0.4,0.7):300—(0.5,0.8):400</td>
</tr>
<tr>
<td>InputPoint Set B</td>
<td>(0.2,0.1):100—(0.3,0.3):100—(0.4,0.7):100—(0.5,0.8):100</td>
</tr>
<tr>
<td>InputPoint Set C</td>
<td>(0.2,0.1):200—(0.3,0.3):200—(0.4,0.7):200</td>
</tr>
<tr>
<td>InputPoint Set D</td>
<td>(0.2,0.1):700—(0.3,0.3):700—(0.4,0.7):700—(0.5,0.8):700</td>
</tr>
</tbody>
</table>
### 2.1.2.2. LinearRegressor inherits from Regressor

LinearRegressor implements the Regressor interface. The implementation should be tested to ensure proper design as well as data availability.

#### 2.1.2.2.1. LinearRegressor sets Regressor::_RegressorType

Each type of regressor must set Regressor::_RegressorType (obtained via Regressor::RegressorType) within its constructor.

##### 2.1.2.2.1.1. Procedure

Create a new LinearRegressor instance. Cast this object to a Regressor. Obtain the result from Regressor::RegressorType

##### 2.1.2.2.1.2. Pass Condition

The type returned is equivalent to RegressorType::LINEAR_LEAST_SQUARE.

#### 2.1.2.2.2. LinearRegressor overrides unimplemented Regressor::UpdateModel

Each regressor must implement the Regressor::UpdateModel abstract method.

##### 2.1.2.2.2.1. Procedure

Create a new LinearRegressor instance. Call LinearRegressor::UpdateModel.

##### 2.1.2.2.2.2. Pass Condition

A MethodNotImplemented exception is not thrown.

#### 2.1.2.2.3. LinearRegressor uses new data when Regressor::UpdateModel is called

When LinearRegressor::UpdateModel is called, a change in the underlying InputPoint set that defines a Regressor has occurred. Regressor then calls Regressor::UpdateModel and the call filters down to the implementation.

##### 2.1.2.2.3.1. Procedure

Create a LinearRegressor instance and call LinearRegressor::InputPoints with the set of InputPoints from InputPoint set A. Call LinearRegressor::GetSmoothedValueOfMostRecent with no parameter.

##### 2.1.2.2.3.2. Pass Condition

The call should return a Point equivalent to \{0.5,0.7,1,0\}. 
2.1.2.3. **DataPoint with LinearRegressor and Regressor**

`DataPoint` uses a `Regressor` to discover information about its data set using the history it has. This needs to be tested.

### 2.1.2.3.1. SmoothedValue

This test will ensure that `DataPoint` and `LinearRegressor` are capable of smoothing data. This test will also ensure that `LinearRegressor` properly extends `Regressor`.

#### 2.1.2.3.1.1. Procedure

Create a `DataPoint` object. Continually push `InputPoint set A` onto the `DataPoint`. Once finished, call `DataPoint::ChangeRegressor` with `RegressorType::LINEAR_LEAST_SQUARE`. Call `DataPoint::SmoothedValue`.

#### 2.1.2.3.1.2. Pass Condition

The call to `DataPoint::SmoothedValue` returns a `Point` equivalent to `{0.5, 0.7, 1, 0}`

### 2.1.2.3.2. GetPredictedPoint

This test will ensure that `DataPoint` and `LinearRegressor` are capable of predicting a point properly.

#### 2.1.2.3.2.1. Procedure

Create a `DataPoint` object. Continually push `InputPoint set A` onto the `DataPoint`. Once finished, call `DataPoint::ChangeRegressor` with `RegressorType::LINEAR_LEAST_SQUARE`. Call `DataPoint::PredictPoint` with a parameter of `50`.

#### 2.1.2.3.2.2. Pass Condition

The call to `DataPoint::GetPredictedPoint` returns a point equivalent to `{0.6, 0.75, 1, 0}`

### 2.1.2.4. DataPointMarshal and Scene

`DataPointMarshal` is responsible for keeping track of the pairings between `Scene` instances and `InputDevice` instances. Each `InputDevice` can have a `Scene`. This is enforced by the `DataPointMarshal`. The `DataPointMarshal` is also responsible for taking an array of `InputPoints` from an `InputDevice` and assigning them to available `DataPoints` within the `Scene` that is attributed to that `InputDevice`.

Both of these processes are integral to the operation of Clementine and therefore need to be tested fully.
2.1.2.4.1. Scene and InputDevice have 1:1 relationship

It is important that each InputDevice have a Scene that represents it. It is also important that that Scene be unique to that InputDevice, so that data can be interpreted from it confidently knowing that it all came from the same device.

2.1.2.4.1.1. Procedure

Create an InputDevice mock instance with InputDevice::_id of 1. Have it's InputDevice::onInputEvent method provide the InputPoints from InputPoint set B to the DataPointMarshal as an InputEvent via the DataPointMarshal::onInputEvent method.

Call your mock InputDevice's onInputEvent method. Retrieve the Scene from the DataPointMarshal via the DataPointMarshal::Scene method, passing your mock instance as a parameter.

Obtain the current set of DataPoints from the Scene returned using the Scene::ActiveDataPoints. Then obtain the most recent InputPoint assigned to each DataPoint using the DataPoint::InputPoints method.

2.1.2.4.1.2. Pass Condition

Every InputPoint in InputPoint set B should be represented in the set you retrieve from the DataPoints, with no exceptions and no omissions.

2.1.2.4.2. DataPointMarshal does not demote early

DataPointMarshal is responsible for promoting and demoting DataPoints in a Scene from active to waiting and vice versa.

2.1.2.4.2.1. Procedure

Create an InputDevice mock instance with InputDevice::_id of 1. Have it's InputDevice::onInputEvent method provide the InputPoints passed in as a parameter to the DataPointMarshal as an InputEvent via the DataPointMarshal::onInputEvent method.

Call your mock InputDevice with the InputPoints outlined in InputPoint set B. Then do the same but use InputPoint set C.

Retrieve the Scene from the DataPointMarshal via the DataPointMarshal::Scene method, passing your mock instance as a parameter. Now obtain the current set of DataPoints from the Scene returned using the Scene::ActiveDataPoints. Then obtain the most recent InputPoint assigned to each DataPoint using the DataPoint::InputPoints method.
2.1.2.4.2. Pass Condition

Every InputPoint in InputPoint set B should be represented in the set you retrieve from the DataPoints, with no exceptions and no omissions.

Repeat this test for 2, 3 and 4 iterations of passing InputPoint set C to the InputDevice::onInputEvent method. Each time, increment the time value associated with each InputPoint in InputPoint set C by 100 ms.

2.1.2.4.3. DataPointMarshal demotes on time

2.1.2.4.3.1. Procedure

Create an InputDevice mock instance with InputDevice::_id of 1. Have it's InputDevice::onInputEvent method provide the InputPoints passed in as a parameter to the DataPointMarshal as an InputEvent via the DataPointMarshal::onInputEvent method.

Call your mock InputDevice with the InputPoints outlined in InputPoint set B. Then do the same but use InputPoint set C.

Repeat iterations of passing InputPoint set C to the InputDevice::onInputEvent method 5 times. Each time, increment the time value associated with each InputPoint in InputPoint set C by 100 ms.

Retrieve the Scene from the DataPointMarshal via the DataPointMarshal::Scene method, passing your mock instance as a parameter. Now obtain the current set of DataPoints from the Scene returned using the Scene::ActiveDataPoints. Then obtain the most recent InputPoint assigned to each DataPoint using the DataPoint::InputPoints method.

Do the same for the DataPoints located in the waiting queue, by using Scene::WaitingDataPoints. Obtain its most recent InputPoint assigned using the DataPoint::InputPoints method.

2.1.2.4.3.2. Pass Condition

Every InputPoint in InputPoint set B should be represented in the set you retrieve from the Scene::ActiveDataPoints with the exception of the InputPoint defined by (0.5,0.8):100, which must not appear.

This point however must appear in the set you retrieve from the Scene::WaitingDataPoints method, with no others.

2.1.2.4.4. DataPointMarshal properly promotes

Once a data
2.1.2.4.4.1. Procedure

Repeat the procedure for the preceding test except before you pull information out of the Scene, you pass in InputPoint set D to the InputDevice::onInputEvent method.

Now you obtain all of the DataPoints from the Scene via the Scene::ActiveDataPoints and Scene::WaitingDataPoints methods.

2.1.2.4.4.2. Pass Condition

Every InputPoint in InputPoint set B should be represented in the set you retrieve from the Scene::ActiveDataPoints with no exceptions.

There must be 0 DataPoints retrieved from the Scene::WaitingDataPoints method.

2.1.2.5. Subscriber SubscriberMarshal and InputDeviceFactory

The Subscriber class is the portal through which the Interfaces will talk to the core of Clementine. Subscribers are tracked by the SubscriberMarshal, and have a 1:1 relationship with InputDevices.

2.1.2.5.1. Subscriber inheritance and registration

Each Subscriber gets assigned a unique identifier during its Subscriber::Initialize

2.1.2.5.1.1. Procedure

Create a mock class that publicly inherits from Subscriber. Be sure to make its Initialize method fire a call up the hierarchy.

Call the mocks Initialize method.

2.1.2.5.1.2. Pass Condition

The Subscriber::_id property of the mock object must be greater than or equal to 0;

2.1.2.5.2. Subscriber request of InputDevice

A class that extends Subscriber can specify which InputDeviceType it needs to work. This is done via the protected member Subscriber::_inputDeviceType. When Subscriber::Initialize or Subscriber::RetryInputDeviceRegistration is called, the InputDevice is found and assigned to Subscriber::_inputDevice.
2.1.2.5.2.1. Procedure

Create a mock class that publicly inherits from Subcriber. Be sure to make its Initialize method fire a call up the hierarchy. Set Subscriber::_inputDeviceType to InputDeviceType::WEBCAM.

Call the mocks Initialize method.

2.1.2.5.2.2. Pass Condition

The mock's Subscriber::_inputDevice property should be set to an InputDevice instance whose InputDevice::_deviceType is equivalent to InputDeviceType::WEBCAM.

2.1.2.5.3. Subscriber change of InputDevice

2.1.2.5.3.1. Procedure

Create a mock class that publicly inherits from Subcriber. Be sure to make its Initialize method fire a call up the hierarchy. Set Subscriber::_inputDeviceType to InputDeviceType::WEBCAM.

Call the mocks Initialize method.

Now set the Subscriber::_inputDeviceType to InputDeviceType::WII_REMOTE and call Subscriber::RetryInputDeviceRegistration.

2.1.2.5.3.2. Pass Condition

The mock's Subscriber::_inputDevice property should be set to an InputDevice instance whose InputDevice::_deviceType is equivalent to InputDeviceType::WII_REMOTE.

2.1.2.6. Subscriber SubscriberMarshal and Timer

The Subscriber instances will need to be updated frequently with the most recent Scene available to Clementine representing what the InputDevice has seen. These updates will be handled by the Timer and its Timer::ScheduelCall function.

2.1.2.6.1. Subscriber can request a call

2.1.2.6.1.1. Procedure

Create a mock class that publicly inherits from Subcriber. Be sure to make its Initialize method fire a call up the hierarchy. Set Subscriber::_inputDeviceType to InputDeviceType::WEBCAM. Set Subscriber::_updateFrequency to 100.
Modify your mock's `Subscriber::ProcessUpdate` method to store the system time in an array each time it is called.

Call the mocks `Initialize` method.

Wait one second and then delete the mock instance.

2.1.2.6.1.2. Pass Condition

View the array that was stored in the `Subscriber::ProcessUpdate` function. It should have 9 or 10 entries, each 100 or 200 milliseconds apart. If there are 10 entries, each must be 100ms apart. If there are 9, there may only be one 200ms gap, the rest must be 100ms.

2.2. Interfaces

2.2.1. 3DInterface

This section will deal with tests to ensure that the three dimensional interface is able to take data in, process it properly, and pass it out.

2.2.1.1. Compute3DPosition calculation testing

This test will create a situation where `CLThreeDimensionalInterface::ComputeThreeDimensionalPosition()` will be called with a dummy set of points. The output of the function will than be monitored to ensure accuracy of the calculation.

2.2.1.1.1. Procedure

Create an array of sets of two-dimensional points. These sets should have the 3D position already calculated and checked by hand. Pass the sets of points into the function and store the output of each call in another array.

2.2.1.1.2. Pass Condition

The contents of the output array should contain the same values as the pre-calculated ones.

2.2.1.2. InitializeDefaults variable set testing

This test will initialize a new `ThreeDimensionalInterface` and make sure that it sets itself up properly.

2.2.1.2.1. Procedure

Create a new `CLThreeDimensionalInterface`. Check the values that the `Initialize()` function is supposed to set.
2.2.1.2.2. Pass Condition

If the values are what they should be, than the function is operating properly.

2.2.2. Gesture Interface

The Gesture interface is responsible for retrieving data from Clementine Core and making it available to developers in a way that places it in the context of a series of gestures that a developer's application can process as a series of command invocations.

2.2.2.1. Gesture and Vector Interaction

The main atomic unit of gestures, the Gesture class, is itself made up of instances of the Vector class. To ensure data integrity, the interactions between these two classes must be observed and tested.

2.2.2.1.1. Vector Concatenation

2.2.2.1.1.1. Procedure

1. Instantiate a Gesture object, and a Vector object with its magnitude set to 0.0335, and its direction set to 0.156.

2. Invoke Gesture::addStep() with the Vector object as its parameter.

2.2.2.1.1.2. Pass Condition

The Gesture object's steps array should have a size of 1, and dereferencing the first object should retrieve the Vector object. This may be checked by observing its magnitude and direction are the same as the ones provided in step 1 of the test procedure.

2.2.2.1.1.3. Procedure

1. Instantiate a Gesture object, and two Vector objects with magnitudes and directions set as follows: (0.0335, 0.156), (0.182, 2.504).

2. Invoke Gesture::addStep() on the first Vector object, then on the second Vector object.

2.2.2.1.1.4. Pass Condition

The Gesture object's steps array should have a size of 2, and dereferencing the first object, followed by the second object, in the array, should retrieve the first and second vector objects added to it in chronological order.
### 2.2.2.1.2. Vector Computation

#### 2.2.2.1.2.1. Procedure

1. Instantiate a Gesture object, and two Vector objects with magnitudes and directions set as follows: \((0.0335, 0.156), (0.182, 2.504)\).

2. Invoke Gesture::addStep() on the first Vector object, then on the second Vector object.

3. Invoke Gesture::endLocation() on the object with a DataPoint parameter with X and Y values centered at \((0.5, 0.5)\).

#### 2.2.2.1.2.2. Pass Condition

The return value of the Gesture::endLocation() method should be the final location of the gesture's point in space as defined by the sum of the individual Vector object magnitudes, using the rotations specified by the directions of the Vector objects.

### 2.2.2.2. Gesture and GestureInterface Interaction

Gestures typically interact with an instantiated GestureInterface object to provide information on existing gestures, gestures that should be considered, et cetera.

#### 2.2.2.2.1. Gesture Importing

##### 2.2.2.2.1.1. Procedure

1. Instantiate a GestureInterface object, and a Gesture object much like the one instantiated in the Vector Computation test.

2. Invoke GestureInterface::ImportGesture() with the Gesture object from step one as its parameter.

##### 2.2.2.2.1.2. Pass Condition

The return value of the GestureInterface::ImportGesture() call should indicate that there was no error in importing the Gesture object, and the gestureRepository array in the GestureInterface object should have a size of one.

##### 2.2.2.2.1.3. Procedure

1. Instantiate a GestureInterface object, and two Gesture object, each much like the one instantiated in the Vector Computation test.

2. Invoke GestureInterface::ImportGesture() twice, with one of the Gesture objects from step one as its parameter both times.
2.2.2.2.1.4. Pass Condition

The return value of the GestureInterface::ImportGesture() call should indicate that there was no error in importing the Gesture objects, and the gestureRepository array in the GestureInterface object should have a size of two.

2.3. Demos

2.3.1. 3D demo

This section will deal with a wide variety of tests for the Three Dimensional demonstration application.

2.3.1.1. ProcessTick thoroughness check

This test will create a situation where ProcessTick() will be called with a dummy set of renderableObjects. That set will be loaded into a game's SceneGraph. Once ProcessTick() is called, it should iterate over every updatable object in the sceneGraph and allow them to update their positions.

2.3.1.1.1. Procedure

Add renderableObjects to the Scene Graph. Make sure that they are tickable. Then run the processTick() function. The renderableObjects need to be set up so that they will have values that change once they are ticked.

2.3.1.1.2. Pass Condition

The values of the objects should not be the same as before ProcessTick() was called.

2.3.1.2. ProcessTick timing accuracy check

This test will properly evaluate ProcessTick's ability to update every 32 milliseconds and keep that number reliable.

2.3.1.2.1. Procedure

Add a dummy RenderableObject to the sceneGraph. This object will do nothing other than print out the system time every time it is ticked. Game should be run to start generating output about the timing.

2.3.1.2.2. Pass Condition

The time in between each system timestamp should be exactly 32 milliseconds.
2.3.1.3. RenderableObject

2.3.1.3.1. ProcessTick thoroughness check

This test will create a situation where ProcessTick() will be called. Once ProcessTick() is called, it should update the values in its object accordingly.

2.3.1.3.1.1. Procedure

Run the processTick() function. The renderableObject needs to be set up so that it will have values that change once it is ticked.

2.3.1.3.1.2. Pass Condition

The values of the objects should not be the same as before ProcessTick() was called.

2.3.1.4. BoundingBox

2.3.1.4.1. isCollidingWith functionality test

This test will create a situation where two bounding boxes will intersect and need to be detected. An series of intersecting and non-intersecting BoundingBoxes will be created to properly test this.

2.3.1.4.1.1. Procedure

Generate a list of BoundingBox sets. Some that intersect, and some that do not. Run all of these through ProcessTick and record the results.

2.3.1.4.1.2. Pass Condition

The collision detection algorithm should match the expected outcome.

2.3.1.5. Player

2.3.1.5.1. Update3DPosition functionality test

This will test the ability of the demo to interact with the Clementine Library and receive its smoothed 3D positional data.

2.3.1.5.1.1. Procedure

Make calls to the Update3DPosition() function. Send a pre-determined list of coordinates into this function. Print out the position every time it changes.

2.3.1.5.1.2. Pass Condition

Every time the position changes, compare it to the pre-determined values that were set up earlier.
2.3.2. Gesture Interface Demo

The Gesture Demo section needs to handle messages from the Gesture Interface through the X11 calls into the Blender Environment.

2.3.2.1. GestureInterpreter handles gestures properly.

This section ensures the processGesture function of the GestureInterpreter object calls the correct commands when processing a gesture.

2.3.2.1.1. moveMouse

2.3.2.1.1.1. Procedure

Set up a logging scheme for the GestureInterpreter object whenever a call to the ClemX11 module is made. Create a mock GestureInterface object that will automatically call the processEvent function of a GestureInterpreter object. Set the parameters to (Null, -100, 50).

2.3.2.1.1.2. Pass Condition

The Gesture Interpreter log should show a call to the moveMouse function with parameters (-100, 50, true).

2.3.2.1.2. mouseClick

2.3.2.1.2.1. Procedure

Set up a logging scheme for the GestureInterpreter object whenever a call to the ClemX11 module is made. Create a mock GestureInterface object that will automatically call the processEvent function of a GestureInterpreter object. Create a "left mouse button click" gesture and pass this as the first parameter ("gesture", 20, 20).

2.3.2.1.2.2. Pass Condition

The Gesture Interpreter log should show a call to the moveClick function with parameters (1).

2.3.2.1.3. holdKey

2.3.2.1.3.1. Procedure

Set up a logging scheme for the GestureInterpreter object whenever a call to the ClemX11 module is made. Create a mock GestureInterface object that will automatically call the processEvent function of a GestureInterpreter object. Create a "hold left shift key" gesture and pass this as the first parameter ("gesture", 20, 20).
2.3.2.1.3.2. Pass Condition

The Gesture Interpreter log should show a call to the holdKey function with parameters (0xFFE1).

2.3.2.1.4. releaseKey

2.3.2.1.4.1. Procedure

Set up a logging scheme for the GestureInterpreter object whenever a call to the ClemX11 module is made. Create a mock GestureInterface object that will automatically call the processEvent function of a GestureInterpreter object. Create a "release left shift key" gesture and pass this as the first parameter ("gesture", 20, 20).

2.3.2.1.4.2. Pass Condition

The Gesture Interpreter log should show a call to the releaseKey function with parameters (0xFFE1).

2.3.2.1.5. pressKey

2.3.2.1.5.1. Procedure

Set up a logging scheme for the GestureInterpreter object whenever a call to the ClemX11 module is made. Create a mock GestureInterface object that will automatically call the processEvent function of a GestureInterpreter object. Create a "press r key" gesture and pass this as the first parameter ("gesture", 20, 20).

2.3.2.1.5.2. Pass Condition

The Gesture Interpreter log should show a call to the pressKey function with parameters (0x072).

2.3.2.2. Blender receives ClemX11 events

These tests ensure that the Blender environment receives the correct commands from the ClemX11 module through the X11 server.

2.3.2.2.1. mouse movement

2.3.2.2.1.1. Procedure

Create a simple c++ object that includes the ClemX11 header file. The object will simply execute a command after 5 seconds.

Have the c++ object call the moveMouse function passing the following parameters (-50, 0, true).
Start Blender and run the c++ object then ensure the Blender application is fully displayed and has focus.

2.3.2.2.1.2. Pass Condition

The mouse should move to the left by 50 pixels

2.3.2.2.2. mouse click

2.3.2.2.2.1. Procedure

Create a simple c++ object that includes the ClemX11 header file. The object will simply execute a command after 5 seconds.

Have the c++ object call the moveClick function passing the following parameters (1).

Start Blender and run the c++ object then ensure the Blender application is fully displayed and has focus.

2.3.2.2.2.2. Pass Condition

A left mouse click event should occur. The point of view cursor moves the the current mouse location.

2.3.2.2.3. key press

2.3.2.2.3.1. Procedure

Create a simple c++ object that includes the ClemX11 header file. The object will simply execute a command after 5 seconds.

Have the c++ object call the pressKey function passing the following parameters (0x072). (the "r" key)

Start Blender and run the c++ object then ensure the Blender application is fully displayed and has focus.

2.3.2.2.3.2. Pass Condition

The application should enter rotation mode. Any mouse movement will rotate the blender object.

2.3.2.2.3.3. Procedure

Create a simple c++ object that includes the ClemX11 header file. The object will simply execute a command after 5 seconds.
Have the c++ object call the holdKey function passing the following parameters (0xFFE1). (Left shift key)

Start Blender and run the c++ object then ensure the Blender application is fully displayed and has focus.

After 5 seconds, depress and hold the middle mouse key and move the mouse to the right or left.

2.3.2.2.3.4. Pass Condition

The perspective will move to the right or left while keeping the angle the same.

2.3.2.2.3.5. Procedure

Create a simple c++ object that includes the ClemX11 header file. The object will simply execute a command after 5 seconds.

Have the c++ object call the releaseKey function passing the following parameters (0xFFE1). (Left shift key)

Start Blender and run the c++ object then ensure the Blender application is fully displayed and has focus. Quickly press and hold the left shift key.

After 5 seconds, depress and hold the middle mouse key and move the mouse to the right or left.

2.3.2.2.3.6. Pass Condition

The perspective will rotate around an arbitrary point.

2.4. Diagnostic Utility

The Diagnostic Utility is an application that pulls from both Interfaces and displays the data coming from them in an organized and intuitive format. Its aim is to help programmers visualize the data context they are receiving from Clementine, as well as serve as a valuable debugging tool when they are fine tuning their applications.

The Diagnostic Utility uses the QT Library as its application and windowing framework. This section will be a white box test for the Diagnostic Utility as well as a black box test for the interaction between QT and the Diagnostic Utility.

2.4.1. Data Point Sets

This table holds input values that will be referred to during the following tests and used as test data.
The values in the parenthesis ("()") represent the DataPoint::X and DataPoint::Y values. Separate DataPoint instances are then deliniated by dashes ("—"). (x,y)

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataPoint Set A</td>
<td>(0.2,0.1)—(0.3,0.3)—(0.4,0.7)—(0.5,0.8)</td>
</tr>
<tr>
<td>DataPoint Set B</td>
<td>(0.2,0.1)</td>
</tr>
</tbody>
</table>

### 2.4.2. GestureResult Sets

This table holds input values that will be referred to during the following tests and used as test data.

The below table values coorespond to all of the properties of a GestureResult object, which is used to pass gesture event information around Clementine.

<table>
<thead>
<tr>
<th>Set</th>
<th>gesture</th>
<th>certanly</th>
<th>position</th>
<th>rotation</th>
<th>percentComplete</th>
<th>potentialOtherGestures</th>
</tr>
</thead>
<tbody>
<tr>
<td>GestureResult A</td>
<td>null</td>
<td>0.5</td>
<td>(.5,.5,.6)</td>
<td>45</td>
<td>80</td>
<td>null</td>
</tr>
</tbody>
</table>

### 2.4.3. InputPoint Sets

This table holds input values that will be referred to during the following tests and used as test data.

The values in the parenthesis ("()") represent the Point::X and Point::Y values of the InputPoint::_location property. The value following the colon (":") is the timestamp to be given to that via the InputPoint::_time property. Separate InputPoint instances are then deliniated by dashes ("—"). (x,y):time

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPoint Set A</td>
<td>(0.2,0.1):100—(0.3,0.3):200—(0.4,0.7):300—(0.5,0.8):400</td>
</tr>
<tr>
<td>InputPoint Set B</td>
<td>(0.5,0.5):100—(0.4,0.4):200—(0.3,0.3):300—(0.2,0.2):400</td>
</tr>
</tbody>
</table>

### 2.4.4. Chart

The Chart class is the base for all displays in the Diagnostic Utility. It can show multiple sets of data along a set of axis. Even with these capabilites, it is intended to be expanded upon using inheritance. This expansion is handled by multiple children. The children's expansion and inheritance will be tested in this section, along with interoperability with the Chart's helper classes.
2.4.4.1. LocationChart and Chart

The LocationChart is a Chart with only one DataSet with only one DataPoint. However, this DataPoint moved around the chart and simulates the location of something in 2 dimensions.

This test will ensure that LocationChart keeps its DataSets within the above outlined constraints.

2.4.4.1.1. Procedure

Create a LocationChart instance. Call LocationChart::PushPoint consecutively with the values in DataPoint set A. Obtain the DataSet being tracked by the Chart using the Chart::GetDataSetByName method. Pass in the string obtained from LocationChart::DATA_SET_NAME as the parameter to this function, that is what LocationChart names it’s one DataSet.

2.4.4.1.2. Pass Condition

The DataSet returned by the Chart::GetDataSetByName method should only have one DataPoint in its `DataSet::_points array. That DataPoint should be equivalent to (0.5,0.8).

2.4.4.2. ScrollingTimeChart and Chart

The ScrollingTimeChart class is simply a Chart that scrolls from right to left as time progresses. The Chart::_xAxis is required to be time for this chart. So it is used to plot different real time events vs. time.

2.4.4.2.1. Point pushing removes x dimension

When a ScrollingTimeChart is updated with a new point for a DataSet, it assumes that this is the most recent point available for that DataSet, and therefore belongs at an X value of 0 on that chart (since 0 is 0 seconds back). This test will ensure that ScrollingTimeChart enforces this assumption.

2.4.4.2.1.1. Procedure

Create a ScrollingTimeChart instance. Call ScrollingTimeChart::PushPoint with the point from DataPoint set B and pass the string "test" as the DataSet name parameter.

Obtain the DataSet from the Chart by calling Chart::GetDataSetByName method and passing it the parameter "test".

2.4.4.2.1.2. Pass Condition

The DataSet returned should have only one DataPoint in its DataSet::_points property, and that point should be equivalent to (0,0.1).
2.4.4.2. Time advance enforcement

This test will ensure that ScrollingTimeChart enforces the effect of time progression on each of its DataSets.

2.4.4.2.1. Procedure

Create a ScrollingTimeChart instance. Call ScrollingTimeChart::PushPoint with the point from DataPoint set B and pass the string "test" as the DataSet name parameter.

Call ScrollingTimeChart::UpdateChart three times.

Obtain the DataSet from the Chart by calling Chart::GetDataSetByName method and passing it the parameter "test".

2.4.4.2.2. Pass Condition

The DataSet returned should have only one DataPoint in its DataSet::_points property, and that point should be equivalent to (-0.015,0.1).

2.4.4.2.3. Max time back deletion

DataPoints will only be tracked as far back as the ScrollingTimeChart::_maxTimeBAck member allows. If a DataPoint has been around longer than that, it is to be removed. (also if this is the case the DataPoint is on the left Y axis)

2.4.4.2.3.1. Procedure

Create a ScrollingTimeChart instance. Call ScrollingTimeChart::PushPoint with the point from DataPoint set B and pass the string "test" as the DataSet name parameter.

Call ScrollingTimeChart::UpdateChart 1001 times (use a loop).

Obtain the DataSet from the Chart by calling Chart::GetDataSetByName method and passing it the parameter "test".

2.4.4.2.3.2. Pass Condition

The DataSet returned should have no items in its DataSet::_points property.

2.4.4.3. GestureChart and Chart

The GestureChart class is an expanded ScrollingTimeChart. It has a DataSet that represents confidence in gestures at the time they show at, as well as what the gestures were.
It gets gesture objects pushed to it rather than xy positions. These gestures are stored in the GestureChart::_gestures array and their relative confidence (GestureResult::Confidence) is added to a DataSt.

### 2.4.4.3.1. Confidence added to scrolling graph

The relative confidence (GestureResult::Confidence) is added to a GestureChart's DataSt.

#### 2.4.4.3.1.1. Procedure

Create a GestureChart instance. Pass GestureResult A to GestureChart::PushGestureResult.

Obtain the DataSt from the Chart by calling Chart::GetDataSetByName method and passing it the parameter GestureChart::DATA_SET_NAME.

#### 2.4.4.3.1.2. Pass Condition

The DataSt returned should have only one DataPt in its DataSt::points property, and that point should be equivalent to (0,0.5).

### 2.4.5. Perspective and Input Controllers

The Perspective class is an abstract class meant to be used as a method of viewing data. Perspective inherits from QT::QGridLayout, and therefor has the capability to arrange and display QWidget objects, from which all of the Charts extend.

#### 2.4.5.1. 3DInterfacePerspective and Perspective with Charts

The 3DInterfacePerspective implementer of Perspective contains six chart instances, four ScrollingTimeCharts and two LocationCharts. These Charts are updated via the three SLOT functions 3DInterfacePerspective::UpdateLeftEyeCharts, 3DInterfacePerspective::UpdateRightEyeCharts and 3DInterfacePerspective::UpdatePositionalCharts.

The connections between these SLOTS is created when the 3DInterfacePerspective::Initialize method is called with the proper Subscriber child. In this case, this must be a 3DInterfaceInputController.

#### 2.4.5.1.1. SIGNAL/SLOT hookup and Chart Updates for 3D

Once the SIGNALS and SLOTS are hooked up, a call to 3DInterfaceInputController::Update3DPosition will cause all 3 methods to fire off, which will cause updates to the 3DInterfacePerspective's Charts.
2.4.5.1.1.1. Procedure

Create a 3DInterfacePerspective instance. Create a 3DInterfaceInputController instance. Call 3DInterfacePerspective::Initialize with the 3DInterfaceInputController instance you just created as a parameter.

Create a DataPoint object instance. Call DataPoint::PushInputPoint consecutively with the values in InputPoint Set A. Pass this DataPoint to the 3DInterfaceInputController::UpdateLeftEye method.

Create a DataPoint object instance. Call DataPoint::PushInputPoint consecutively with the values in InputPoint Set B. Pass this DataPoint to the 3DInterfaceInputController::UpdateRightEye method.

Create a Point object with values (0.5,0.6,0.7,1) and pass it to 3DInterfaceInputController::UpdatePosition.

Obtain the DataSet from each of the Chart objects in the 3DInterfacePerspective

2.4.5.1.1.2. Pass Condition

The DataSet from 3DInterfacePerspective::_leftXChart must have 1 DataPoint, (0,0.5)

The DataSet from 3DInterfacePerspective::_leftYChart must have 1 DataPoint, (0,0.8)

The DataSet from 3DInterfacePerspective::_rightXChart must have 1 DataPoint, (0,0.2)

The DataSet from 3DInterfacePerspective::_rightYChart must have 1 DataPoint, (0,0.2)

The DataSet from 3DInterfacePerspective::_topDownChart must have 1 DataPoint, (0.5,0.7)

The DataSet from 3DInterfacePerspective::_backForwardChart must have 1 DataPoint, (0.5,0.6)

2.4.5.2. SpatialInterfacePerspective and Perspective with Charts

The SpatialInterfacePerspective implementer of Perspective contains three chart instances, one GestureChart and two LocationCharts. These Charts are updated via the one SLOT function SpatialInterfacePerspective::UpdateGestureData
The connection between this **SLOT** is created when the `SpatialInterfacePerspective::Initialize` method is called with the proper `Subscriber` child. In this case, this must be a `SpatialInterfaceInputController`.

### 2.4.5.2.1. SIGNAL/SLOT hookup and Chart Updates for spatial

Once the SIGNAL and SLOT are hooked up, a call to `SpatialInterfaceInputController::UpdateGestureResults` will cause the SIGNAL method to fire off, which will cause updates to the `SpatialInterfacePerspective`'s Charts.

#### 2.4.5.2.1.1. Procedure

Create a `SpatialInterfacePerspective` instance. Create a `SpatialInterfaceInputController` instance. Call `SpatialInterfacePErspective::Initialize` with the `SpatialInterfaceInputController` instance you just created as a parameter.

Create a `GestureResult` object instance with the data in `GestureResult A`. Call `SpatialInterfacePerspective::UpdateGestureResults` with this as a parameter.

Obtain the `DataSet` from each of the `Chart` objects in the `3DInterfacePerspective`.

#### 2.4.5.2.1.2. Pass Condition

The `DataSet` from `SpatialInterfacePerspective::_topDownChart` must have 1 DataPoint, $(0.5,0.6)$

The `DataSet` from `SpatialInterfacePerspective::_backForwardChart` must have 1 DataPoint, $(0.5,0.5)$

The `DataSet` from `SpatialInterfacePerspective::_gestureChart` must have 1 DataPoint, $(0,0.5)$
3. Black Box Testing

3.1. Input To Core

This section will deal with tests to ensure that the Input Device module and Data Aggregation module pass data fluently between them.

3.1.1. Core Timer and InputDevice

Tests are needed to ensure that InputDevice instances can read the current application time from Core and assign it properly to an InputPoint.

3.1.1.1. InputPoint time assigning for a Wii Remote Device

This test specifically will create an environment where an InputDevice will call its InputDevice::onInputEvent method in which it assigns times received from core to each InputPoint.

3.1.1.1.1. Procedure

Create a mock DataPointMarshal singleton to intercept the DataPointMarshal::GetInstance() #onInputEvent call coming from the InputDevice. Create a mock version of the libWiiMote WiiRemote class to provide fake input data to the InputDevice. Instantiate the Timer, and then delay the method with a for loop. Take the system time from before the loop and after, and then subtract to get the number of milliseconds between. Be sure the loop only delays for around 10 milliseconds.

Create an instance of a WiiMoteDevice and call its Read method.

3.1.1.2. Pass Condition

The data passed to the mock DataPointMarshal class should contain the current time since the application has started, about 10-12 milliseconds.

3.2. Core To Three Dimensional Interface

The Three Dimensional Interface and the Core of the library are constantly transferring information. These tests will check that information transferrance for accuracy.

3.2.1. Core Subscription

3.2.1.1. Procedure

1. Instantiate a ThreeDimensionalInterface object.
2. Invoke an instance of the Core Clementine library.

3. Register the `ThreeDimensionalInterface` object as a subscriber to the Core Clementine library.

### 3.2.1.2. Pass Condition

The Core Clementine object should report that the `ThreeDimensionalInterface` object is subscribed to its updates.

### 3.3. Core To Gesture Interface

The Core Clementine library must interact on a regular basis with the Gesture interface. These tests ensure that those interactions are consistent and do not cause unexpected errors.

#### 3.3.1. Core Subscription

##### 3.3.1.1. Procedure

1. Instantiate a `GestureInterface` object.

2. Invoke an instance of the Core Clementine library.

3. Register the `GestureInterface` object as a subscriber to the Core Clementine library.

##### 3.3.1.2. Pass Condition

The Core Clementine object should report that the `GestureInterface` object is subscribed to its updates.

#### 3.3.2. Gesture Observation

##### 3.3.2.1. Procedure

1. Instantiate a `GestureInterface` object, and a `Gesture` object much like the one instantiated in the Vector Computation test.

2. Invoke `GestureInterface::ImportGesture()` with the `Gesture` object from step one as its parameter.

3. Implement `GestureInterface::SendGestureResults()` to return the `GestureResult` object that is passed to it as a parameter.

4. Invoke a mock instance of the Core Clementine library whose expectations are set to deliver a series of points to the `GestureInterface` class. These points should be mathematically exact with reference to the `Gesture` object that has been added to the gesture interface, and its internal `steps` array.
5. For a set interval, invoke `GestureInterface::FindGestures()` on the `GestureInterface` object. At the last iteration of this set of invocations, the last set of data should be sent from the Core mock object.

6. Invoke `GestureInterface::SendGestureResults()`.

### 3.3.2.2. Pass Condition

The `GestureResult` object returned by the invocation in step 6 should contain a copy of the `Gesture` object defined in step 1. The certainty should be within a certain range of 100 percent.

### 3.4. 3D Interface to Demo

The demo for the 3D Interface must interact properly with the Core Clementine library as well as the 3D Interface.

#### 3.4.1. Procedure

1. Create a 3D Interpreter class.

2. Instantiate both the Core Clementine object as well as a `3DInterface`.

3. Create a mock `InputDevice` for the `3DInterface`.

#### 3.4.2. Pass Condition

When the device passes its data through the Core object, and it through the `3DInterface`, to the `3D Interpreter` in the Demo, it will have passed.

### 3.5. Gesture Interface to Demo

The demo for the Gesture interface must interact properly with the Core Clementine library and the Gesture interface itself.

#### 3.5.1. Procedure

1. Create an instance of the `GestureInterpreter` class.

2. Instantiate a Core Clementine object, as well as a `GestureInterface` object, and import a single gesture into it, as described in the Gesture Observation test.

3. Provide the Core Clementine object with an `InputDevice` instantiation. This could be a mock device that sends preprogrammed data to Core.
4. Ensure that the mock device sends data that matches the Gesture object in the 
   GestureInterface object's repository to within a certain degree of accuracy.

### 3.5.2. Pass Condition

This test is looking to ensure the signal programmed into the GestureInterpreter class, linked 
   to the gesture performed by the mock data in the InputDevice, is correctly fired. This test passes 
   if this signal is fired when the GestureInterpreter class receives the first GestureResult back 
   from the interface.

### 3.6. Interfaces to Diagnostic Utilities

This section will deal with tests to ensure that the Diagnostic Utility receives data correctly from the 
   interfaces.

#### 3.6.1. SubscriberMarshall to 3DInterfaceInputController

This tests simulates the passing of a scene object to the 3DInterface object within the Diagnostic 
   Utility

##### 3.6.1.1. Procedure

create a mock perspective object (extends QObject) for the purpose of intercepting the data 
   from the three 3DInterfaceInputController object signals.

create a mock SubscriberMarshall object and modify the RegisterinputDevice function to 
   immediately call the registering subscribers processUpdate function with a mock scene. The 
   scene will include all generic information in addition to two data point objects within the 
   activeDataPoints array: DataPoint A will contain one InputPoint (110, 100, 0, 0) DataPoint 
   B will contain one InputPoint (-110, 100, 0, 0)

##### 3.6.1.2. Pass Condition

The 3DInterfaceInputController should report three sets of data to the mock perspective: 
   LeftEye position - A data point object containing one input point (x,x,x,x) RightEye position 
   - A data point object containing one input point (x,x,x,x) Position (depth) - A point object 
   (x,x,x,x)

#### 3.6.2. SubscriberMarshall to 
   SpatialInterfaceInputController

This tests simulates the passing of a scene object to the SpatialInterface object within the 
   Diagnostic Utility
3.6.2.1. Procedure

The SpatialInterfaceInputController should contain a gesture defined as the movement of a single point to the right.

create a mock perspective object (extends QObject) for the purpose of intercepting the data from the SpatialInterfaceInputController object's UpdateGesture signal.

create a mock SubscriberMarshall object and modify the RegisterInputDevice function to immediately call the registering subscribers processUpdate function with a mock scene. The scene will include all generic information in addition to one data point object within the activeDataPoints array: DataPoint will contain multiple InputPoints (110, 100, 0, 0), (120, 100, 0, 0), (130, 100, 0, 0)

3.6.2.2. Pass Condition

The scene should be interpreted by the SpatialInterfaceInputController object to be the previously defined right movement gesture with a certainty rate > .9
Glossary

2D Projection
The representation of a 3D environment projected onto a 2D plane.

Alpha value
A value used to determine the transparency of a color. Typically the 4th value added on to an RGB triplet, denoted by RGBA. For example, 0xFF000088 is a semi transparent red.

Angle Axis
An alternate structure for storing rotational data. It is defined by a vector and a rotation about that vector.

API
Application programming interface, a class library meant to house similar functions and algorithms for use in 3rd party applications.

Avatar
The object in an application created by the developer that the user directly interacts with/controls.

Blender
Blender is an open source 3D modeling environment available under the GNU license.

Blob
A group of similarly colored pixels found in a frame of web camera input, defined by an area in pixels.
Bluetooth
Bluetooth is a short-distance wireless protocol for portable devices. It is used within the scope of our project, as in the Wii Remote, to provide wireless connectivity between devices and a computer.

Circular regression
Finding a circle that most closely represents the trends of a set of independent points of data.

Context-sensitive
An action that is different depending on the current area of interaction. For example: a context menu, when requested on a file, shows "Delete", "Rename", "Edit", et cetera.

Data Point
A data point is a point being tracked by the engine. Its values are what are reported to the subscriber.

Dead Reckoning
A method for guessing the next value of a predictable pattern. Take the pattern or function that represents your data, and plug in the next independent variable value to it.

Derivative
The slope of a line. The rate of change in a function over time.

Developer
The target audience for our library. This is the person that will develop custom applications for users to use.

Error Bounds
The amount of error tolerated for a certain type of input. When the input

Extrapolation
Estimating the next value of an input event based on previous data.

Extrusion
Extrusion is an operation performed on an object in three-dimensional modeling. A plane of an object, when extruded, expands out or in from its original location, while its edges are still attached to the connecting edges of the object.
First-in Last-out Queue
A queue who can only provide the most recently entered piece of data. An example would be a stack of paper. You can only retrieve the piece of paper on top, and you can only place a piece of paper on top.

Gimble lock
An event that occurs when using primitive forms of rotational representation. When rotating 2 of the 3 dimensions at the same time, the 3rd will become locked and any rotation on that axis will be nullified.

Global Positioning System/Service
A service that provides the position on the earth of a receiver. It uses a minimum of 3 satellites to triangulate surface position. The civilian service is accurate to within 3-5 meters.

Graphics Library
A library containing algorithms and classes for displaying 3D environments on a screen.

Homogenization
Homogenization in the context of this project is simply dividing each dimension by the highest available dimension. For example, to homogenize the following point (4,8,2), place 1/2 in the dimension higher, resulting in (4,8,2,0.5), then multiply through by the 4th dimension. Resulting in (2,4,1). This method is used frequently in computer graphics for translating 3D points to 2D points.

Infrared
Infrared is a form of light invisible to human vision. It is commonly used to provide basic connectivity, wirelessly, between devices, such as television remotes.

Input Frame
An input frame is a set of input points that represent all of the read points coming from the input device at that time.

Input Point
An input point is an XY value for a point obtained from an input device.

Interface
An interface is a piece of software that allows a device to interact with the core library through a series of function calls.
Lathing
Lathing is an operation performed on a two-dimensional plane. The final result of lathing is a three-dimensional shape whose edge entirely around is the same shape as the external edge of the two-dimensional plane.

Least Squares
A method of linear regression. The line is defined as follows

Library
A library is a collection of classes that can be used by a programmer when writing an application. A library usually performs a common or complex function that many applications can use.

Line of best fit
The line that best represents a grouping of coorelated or semi-random data. It is computed using linear regression.

Linear regression
Finding a line that most closely represents the trends of a set of independent points of data.

Multipoint object
Multipoint Object - an object that contains multiple data points, all statically positioned.

Parallax Effect
The parallax effect is a phenomenon that occurs when an object is viewed from two viewpoints causing the object to appear at its proper depth. In the scope of our project, when an object moves and rotates appropriately opposite on screen to the viewers position in the room, an illusion of depth is created.

Perspective
Some applications need a different internal windowing/dialog scheme depending on the operation they are trying to perform. These different schemes are called perspectives. An example would be the Eclipse debug and development perspectives. Each has dialogs open that best suit the current mode of operation of the application.

Precompilation Directive
A preprocessor directive is code that is interpreted as a compiler compiles code. These segments tell the compiler what sections of code to compile and how to compile them. Examples are #ifndef … #endif.
Preprocessor Variable
   A variable that can be set in a compilation call, and can be used by percompilation directives.

QT
   QT is a robust cross platform application framework originally written for Linux in C++, developed by Trolltech. http://www.qtsoftware.com/title-en

QT Library
   An application framework for linux containing many useful functions and classes. One such function taken advantage of by this library is the QTimer, which will repeatedly call a function every X seconds.

Quaternions
   Quaternions are the standard structure for storing rotational data. They are immune to gimble lock.

QWidget
   The base class for any object in an application using the QT Libraries that wishes to be rendered to a location in a window.

Random Variable
   A statistics term that denotes an entity that can take on any value within a data set.

Reference frame
   A reference frame is a set of axes within which a position in space can be measured.

Regression
   A statistical method for finding the mean of a data set.

Signal/Slot Macro
   A C++ macro used in QT applications in which you can link functions together without the use of function pointers. A member must be defined as a SIGNAL or a SLOT before it can utilize the macro. You must compile your application using the qmake-qt4 builder utility.

Smoothing
   Smoothing in the scope of this project refers to the removal of outliers from a data set in order to yield a more consistent function, as well as bounding accurate data to within an error range to consider it acceptable data.
STI
The STI, or Spatial Tracking Interface, is a set of functionality that allows programmers to provide manipulation of their application through observable gestures that the user makes. See Spatial Tracking Interface for more information.

Subscriber
A subscriber is an agent that requests data point values from the library.

Thread
A thread is a process in a computer. A single processor can only process one thread at a time, though processed threads are switched rapidly in most modern computers. Having multiple threads allows for a long running process to run independent of a quicker one. Multi-core machines take advantage of threads by executing multiple threads at the same time.

User
The target audience of our developer. This is the person that will use the custom application that the developer creates.

Variance
The variance of a random variable is the average of the squared distance of its values from the statistical mean.

Wii Remote
A Wii Remote is an input device created by Nintendo for their Wii video game console. It has several buttons for digital manipulation, as well as an accelerometer, for measuring relative velocity, Bluetooth for connectivity, and infrared sensors for measuring contextual position.

Yaw-Pitch-Roll
An alternate structure for storing rotational data. It is defined by a rotation amount in each of the 3 axes, X, Y and Z, or Pitch, Yaw, and Roll, respectively. This form of representation is vulnerable to gimble lock.