Android VirtuTrace Remote
VTRemote

Final Report

Group May14-21

Kollin Burns

Tanner Borglum

Sheil Patel

Alexander Maxwell

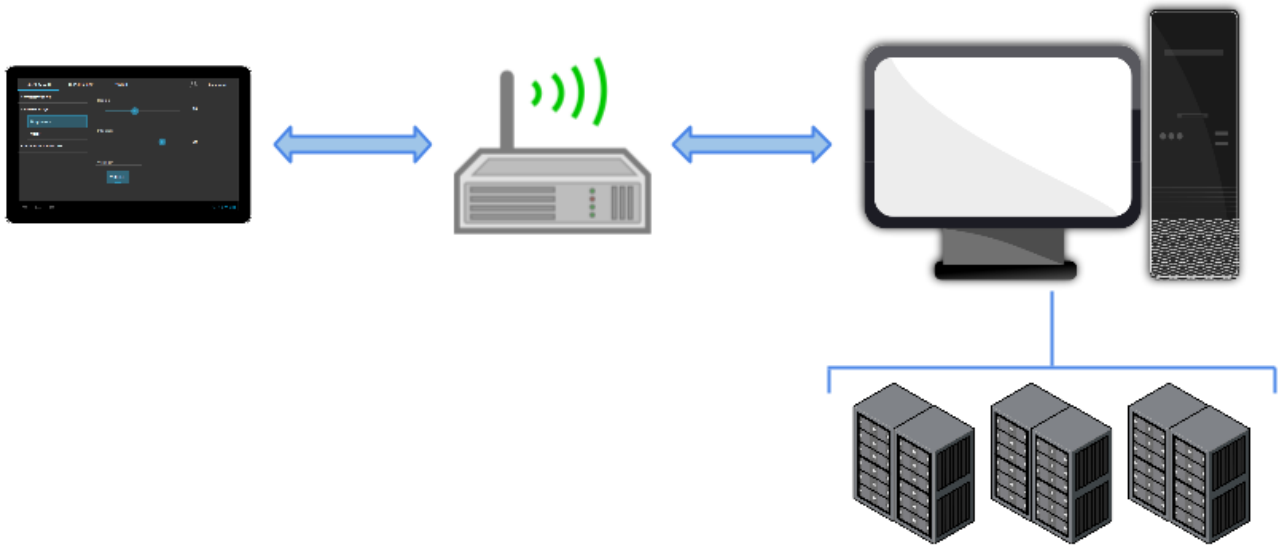
Lukas Herrmann

Table of Contents

Project Overview

The purpose of our project was to create a wireless remote for a virtual reality simulation. The remote application (VTRemote) was to be run on an Android-based mobile device and needed to interact wirelessly with the simulation engine (VirtuTrace) being run inside the C6 Virtual Reality Simulator located in Howe Hall. The main function of VTRemote is to allow the user to monitor and make changes to objects and properties in the simulation and have them reflected in real-time.

For example, the scene might have various objects in the scene, let's say an apple, and our app allows the user to control properties of the apple, such as color, size, and placement in the scene. The app aids the user in debugging by allowing the user to watch objects in the scene to see how they change over time.



Conceptual Model

Project Goals

Prior to the start of this project, the process for making a change to a simulation scene while inside the C6 was lengthy and costly. Since there was no support for real-time updates, the user had to exit the C6, shutdown the simulation, edit a value in a configuration file or even modify source code, possibly rebuild the simulation software, restart the engine, and re-enter the C6. Our main goal for the project was to completely deprecate this process.

The user would need to be able find the object or property that they want to manipulate, have the ability to view its value(s), make a modification, and see the change occur before they even lift their finger off the device. Now, with the help of VTRemote, they can do just that without ever leaving the C6.

From a high-level perspective, these were our major design goals:

- The VTRemote user interface needs to be robust enough to accommodate the various types of objects and properties in the scene but also remain accessible so users would not need an extensive understanding of the implementation of the app in order to effectively utilize it.
- The user needs to be able to easily locate any given property or object.
- All changes to properties and object values need to be reflected synchronously in real-time for both VTRemote and VirtuTrace.
- The implementation design of VTRemote needs to be easily extensible to allow for the additional support of future data types.

Deliverables

Our main deliverable is the Android application that is the core of VTRemote. Secondary deliverables will be various documentation on how to use the app from within the C6 and also information about the implementation details for the app so future developers can easily understand how to maintain and extend it. Additionally, we implemented several key additions to the VirtuTrace code base.

System Requirements

- The system must provide the user with a listview or treeview that allows them to select objects in the scene.
- The system should provide visual feedback when a user has selected an object for editing.
- If the user makes changes to an object in the scene, the system must reflect the changes in real time (or with as little lag time as possible).
- The system should allow the user to save the current configuration if the user has made changes to the environment.

Functional Decomposition

VTRemote and VirtuTrace have several large functional units. VTRemote has a network management thread, a fragment to display the VirtuTrace environment structure, and a fragment to display properties of a single item in VirtuTrace. Our modules in VirtuTrace include a network management thread, an action queue, and an action listener.

The VirtuTrace network manager will look for an open port, create a TCP server socket on it, and wait for commands from the client currently connected to it. After it receives a start message macro, it will add the operation to an action queue after receiving the end message macro. The VirtuTrace action listener will wait until VirtuTrace is not rendering, then check for an action in the queue and lock rendering if there is. Once it has the lock, it will complete the action and update the queue front. If an action requires VirtuTrace to send data back, the network manager will wait for it to return the data.

The main interaction with VirtuTrace requires a network manager for VTRemote. This manager will respond to actions in other views and send messages to VirtuTrace. When an action needs to be performed, the manager will send a start macro, the operation, the data (if any), and an end macro through a TCP connection in an ASCII string or binary format depending on connection speed. Both formats use a native Open Scene Graph notation for data representation.

There are three main UI components to generate actions. The tab selector at the top of the screen switches modes; one for editing the scene graph and another for the property map. Both of these views represent the corresponding part of the VirtuTrace environment in a listview. When a user selects a tab, VTRemote queries VirtuTrace for the latest environment state and updates the view. When a user selects an item in the tree view, a second view on the right will query VirtuTrace for the current item property states and show them in the view. If a user changes one of the properties, VTRemote will send the change to the VirtuTrace which will update the simulation in real-time.

System Analysis

Interdependencies

Functionally, VTRemote will require a connection to an instance of VirtuTrace since it is a tool specifically designed to augment a simulation in progress. However, the app is not dependent on VirtuTrace to run. It will essentially idle until it makes a connection.

For VirtuTrace, there are not any dependencies on the VTRemote. It will function as normal regardless whether it is connected to the Android app.

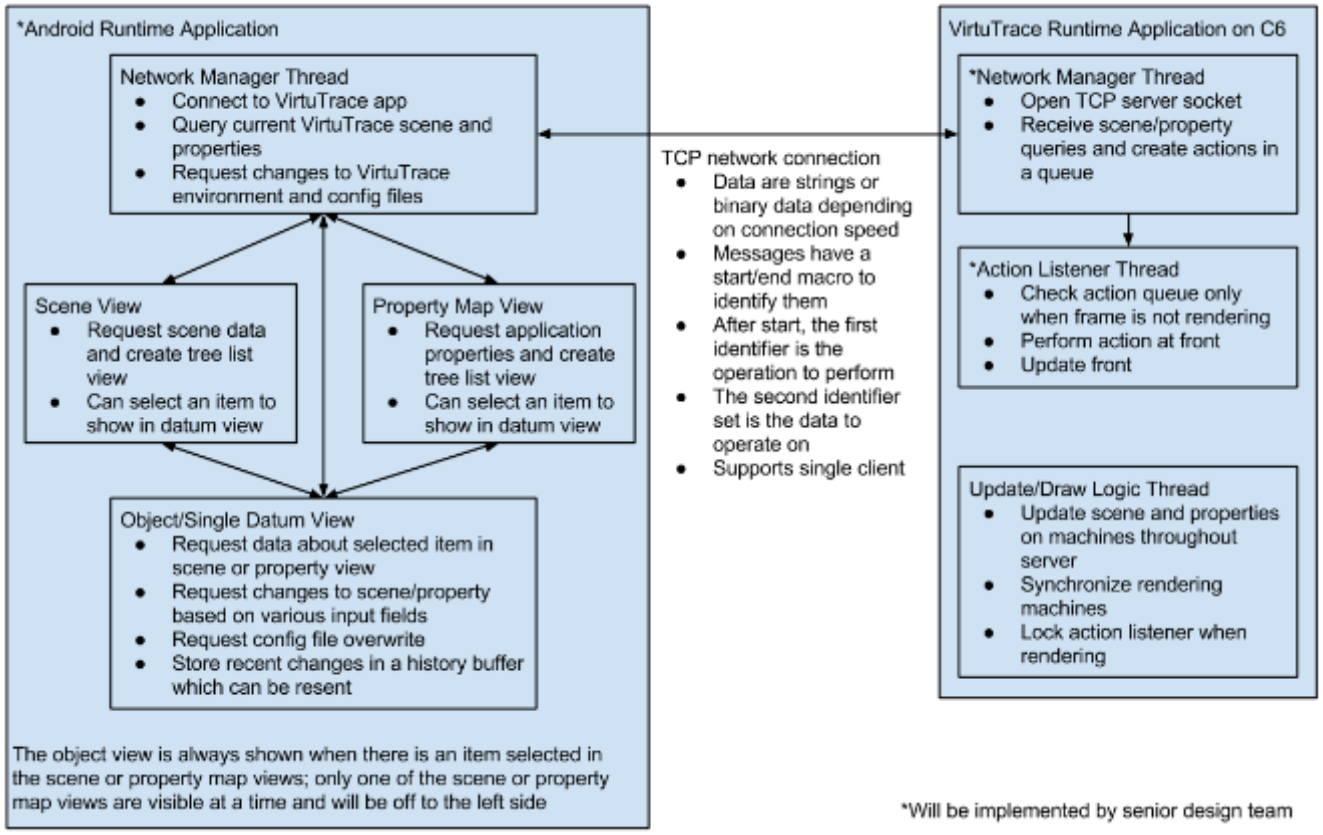
Potential Bottlenecks

The most threatening performance impact will come from the reliance on the wireless network for communication. In an effort to reduce this impact, we will focus on optimizing network packet size and minimizing the frequency of data transmissions.

Security

No sensitive data will be sent between the VTRemote and VirtuTrace. The only types of information that will be transferred are configurable properties of the simulation or objects and the scene objects themselves. Both sides will utilize Open Scene Graph, an open-source library, for serializing and parsing the transmitted data. The library does not encrypt the data as there is no need for it.

System Block Diagram



I/O Specifications

VTRemote

Being an Android app targeting tablets, VTRemote will utilize the device's built-in touch input manager for handling user input. Based on user input, it may then send data over a TCP/IP wireless connection to VirtuTrace. Additionally, it will receive information, through the same connection, from VirtuTrace for populating the list of scene objects, gathering a list of variables available to "watch," etc.

VirtuTrace

VirtuTrace will send and receive wireless communication to and from VTRemote. It may also communicate with a variety of other I/O devices. However, that communication and those devices will not have any impact on VTRemote.

Interface Specifications

VirtuTrace - VTRemote Communication Interface

Scenes within VirtuTrace are built upon an existing open-source library called Open Scene Graph. It provides the implementation and models for each scene object as well as many useful operations on those objects. VirtuTrace will utilize this library for communication with VTRemote since it has functions for serializing and deserializing the scene objects.

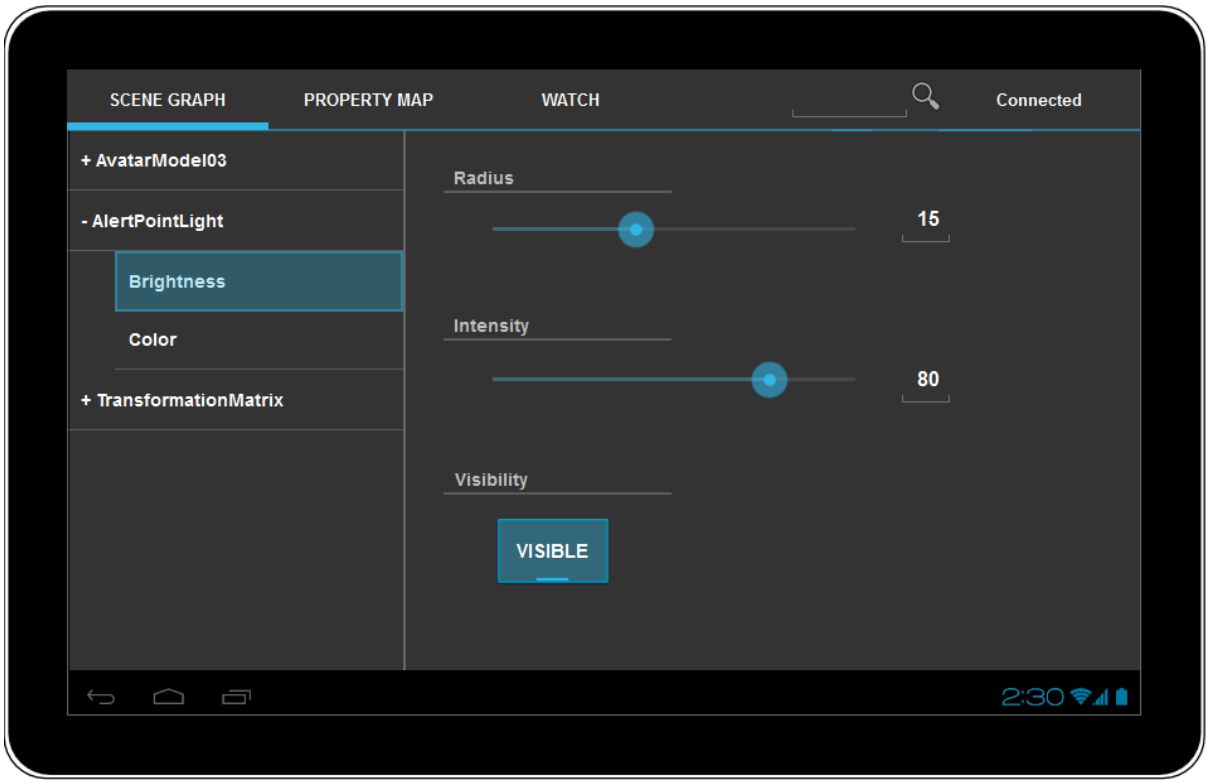
On the other side, VTRemote will use a SWIG (Simplified Wrapper and Interface Generator) Java wrapper library that wraps around Open Scene Graph to provide the same serialization benefits within the Android app.

To keep the interface consistent, we have defined a set of keywords to indicate the beginning and end of a message, as well as a delimiter for easy separation of the elements within the message. Other than the initial handshaking, VTRemote will initiate all communication and VirtuTrace will only respond. The message syntax is shown in the table below.

Start Keyword	A static keyword indicating the start of a message.
Mode	The mode parameter indicating whether the message is related to Scene Graph or Property Map.
Operation	The operation to be performed (e.g. getting or setting a property).
Operands	The operation's parameters.
Stop Keyword	A static keyword indicating the end of a message.

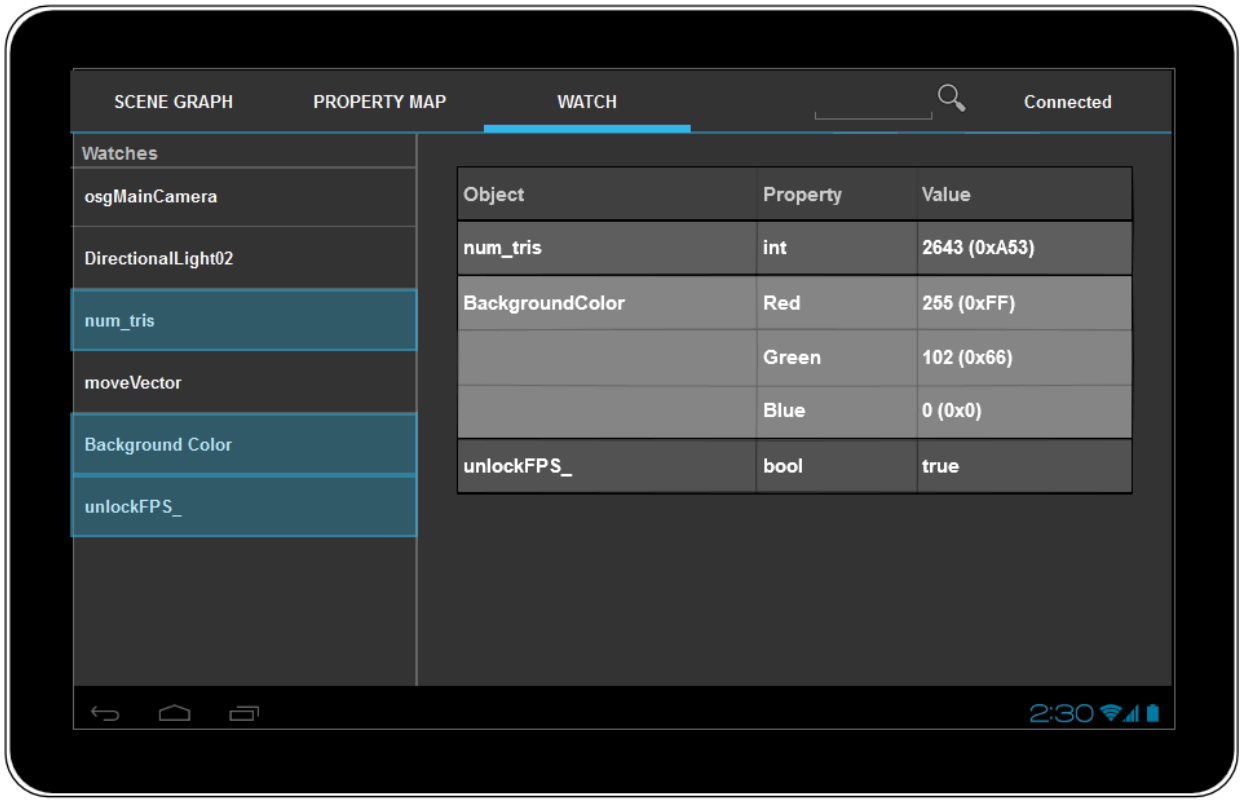
VTRemote - User Interface

The user interface for VTRemote is split into three main tabs: Scene Graph, Property Map, and Watch. The Scene Graph and Property Map tabs are very similar. They both have a listview on the left hand side of the screen that the user is allowed to traverse. Once the user selects an editable object in the list, its modifiable properties are displayed in the screen space to the right.



Scene Graph Tab Concept Rendering

The Watch tab allows the user to select a handful of variables, objects, and properties to observe how they change as the simulation runs.This feature is for mostly for debugging purposes.



Watch Tab Concept Rendering

HW Specifications

- Wireless Networking - 802.11a/b/g/n
- Screen Size/Resolution - at least 7.0” screen measured diagonally with at least 720p (1280x720) resolution

Software Specifications

- Operating System - Android 4.2 or higher

Testing Procedures and Specifications

The testing for our system falls into three main areas: the android application, the VirtuTrace functions, and the testing for the network managed between the two of them. Each area will be tested using unit tests whenever possible to maximize automation as well as manual testing as necessary.

VTRemote

Testing of the Android application happened both under forced conditions and live conditions. The forced conditions guaranteed that the Android application can request and receive data, display data in desired fashion, and then send changes. After testing under forced conditions,

live standalone testing and C6 testing showed that there is all around proper functioning of the application. We also have a tool called UI Automator that allowed us to test the dynamic construction of UI elements automatically. Through this testing process, we were able to design and implement a user interface on the Android side that met our client's specifications.

VirtuTrace

Testing of the VirtuTrace functions was mostly done through unit tests and standalone live testing. The unit tests made sure that the functions perform as needed for the rest of the application. Live standalone testing makes sure that the functions are behaving as expected under working conditions.

Networking

The network tests fall into two areas. The first area is proper serialization, deserialization, and transfer of data. The second area is speed testing which will be both virtual and live to account for shear volume of data and also actual handling of the C6 network constraints. It is important that our application maintain high speeds for all data volumes.

Implementation Details

VirtuTrace

Networking: A separate thread was created for handling the networking within VirtuTrace. This worker thread is responsible for the sending and receiving of messages. When a message is received, it is added to a queue which is processed just before the drawing of each frame.

Making Changes: While processing messages, VirtuTrace determines whether the operation is meant for the Scene Graph or Property Map, parses out the necessary information, and creates a work order item to executed. These work orders are processed in between the processing of messages and drawing of the next frame, ensuring that all objects and properties are updated prior to drawing them in the scene.

Reporting Changes: After attempting to execute a work order, VirtuTrace sends a status report to VTRemote indicating whether the operation was performed succesfully. Additionally, VirtuTrace will also send updates to VTRemote if a value changes while the simulation is running, in case VTRemote was not the cause of the change.

VTRemote

Networking: Similar to VirtuTrace's networking, VTRemote has its own thread for the receiving and sending of messages. However, VTRemote's networking thread also performs any necessary operations associated with the message and then simply tells UI thread to update the

views accordingly. This is done to ensure that the UI is always responsive to user input and not busy processing network messages.

Making Changes: After the user selects an object or property, they are presented with possibly several UI elements that can be used to modify the desired value. There are a variety of widgets at the user's disposal, but only those that are applicable to the selected object are presented to the user. For example, the user may select an object with a integer value. It's value could be edited by selecting the text box it's contained in and using the virtual keyboard to alter it or by using the custom rotary dial for smooth continual modifications.

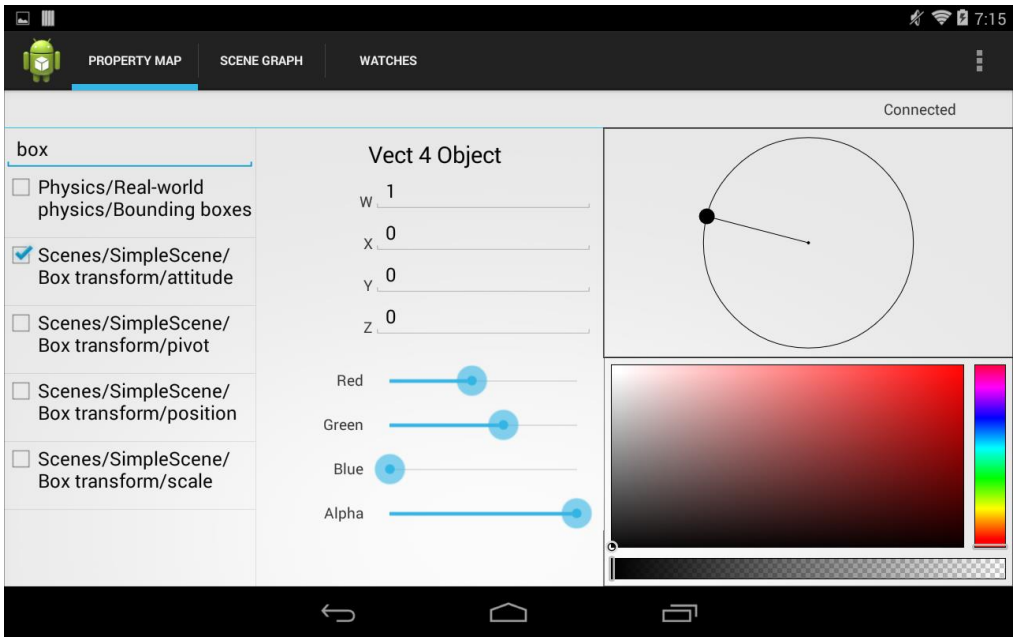
Reporting Changes: Each interactive UI element that corresponds to a property or an object's value, has a listener that performs some function when the UI is interacted with. Usually, these interactions involve the direct modification of values and so each update is sent to VirtuTrace at the time the change occurs.

Design Decisions

The client had explicitly requested an Android application which is the leading reason we chose the platform. Additionally, Android is preferred in this project because it is open source and based in Java, a language that is very well known at Iowa State and by most developers. This improves the likelihood that future developers will be able to more quickly continue the project.

Another important design decision we made was the use of TCP as the protocol for communication between the C++ server and the Java Android app. We chose to do this because TCP has built in error checking and correction, so if there are any issues with sending data back and forth between the server and client, TCP can resolve most of them without the development team having to change the implementation. One downside to using TCP over UDP is speed. The UDP protocol allows data to be send much quicker over a network than TCP does, but because of the nature of the data we are sending, we felt that having the error checking capabilities of TCP was more useful for this particular project than using UDP.

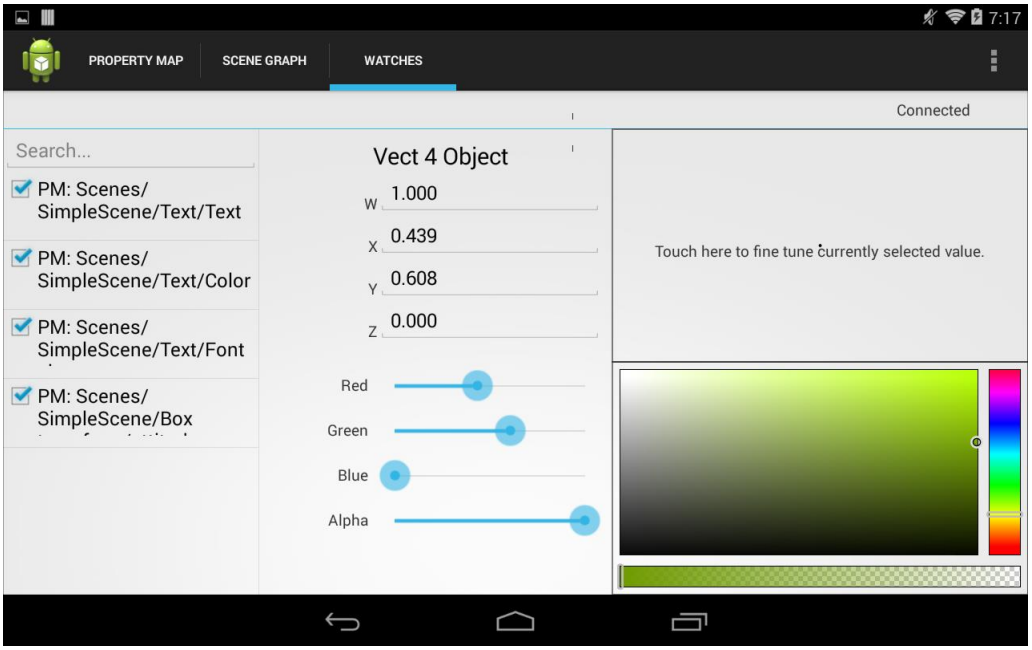
Below are final user interface screenshots to visualize the difference between the concept renderings and the final design:



Property Map Tab:

As mentioned previously, the Scene Graph tab is identical in design to the Property Map tab.

The left list view allows the user to find the object they want to edit in the scene. There is also a search feature as seen at the top of the list view that allows the user to quickly find objects without having to scroll through large lists of objects. Once an object is selected, the right ⅔ of the screen inflate with a view that show the various properties that are editable for the selected object. Due to the variety of objects that are available in a given scene, each object has a specific layout that will inflate when selected. Any changes the user makes appear in the scene in realtime.



Watches Tab:

The user can add watched objects in either the Scene Graph tab or the Property Map tab. Once an object is selected to be watched, the object will appear in the Watches tab. Essentially, the Watches tab appears as a favorites list where the user can add object they want to save for editing at a later time.

Work Breakdown

Implementation of the project was broken up into two main teams, one team that worked on the VirtuTrace side and another team working on the Android side. The breakdown of work was as follows:

Tanner Borglum

- (De)Serialization of VirtuTrace objects for synchronization across the C6 server cluster.
- Scene Graph support on both VirtuTrace and VTRemote.
- Documentation and administrative responsibilities.

Kollin Burns

- Android GUI, backend, and networking implementation and optimization.
- Creation and integration of custom UI elements.
- Networking within VirtuTrace.
- Documentation and administrative responsibilities.

Lukas Herrmann

- Reading and writing to the Property Map within VirtuTrace.
- Documentation and administrative responsibilities.

Alexander Maxwell

- Android UI framework design and implementation.
- Documentation and administrative responsibilities.

Sheil Patel

- Android GUI design and implementation.
- Poster design and creation.
- Networking within VTRemote.
- Documentation and administrative responsibilities.

Appendix I: Operation Manual

To connect to a scene in VirtuTrace:

1. Click the overflow settings button in the top right corner of screen
2. Select “Settings”
3. Select “IP Information”
4. Enter IP address and Port of the VirtuTrace machine
5. Select OK
6. Press back on the tablet to return to the application main page
7. If successfully connected, text in the top right should show “Connected”
8. If not connected, please ensure correct information was entered and ensure VirtuTrace is running on the host machine

To load the property map object list:

NOTE: You must be first connected to VirtuTrace to be able to load the property map object list

1. Ensure you are on the Property Map tab
2. Press button that says “Download Property Map”
3. The list view on the left ⅓ of the screen will populate with property map objects if available

To edit the properties of a property map object in a VirtuTrace scene:

NOTE: You must be first connected to VirtuTrace to be able to load the property map object list

1. Ensure you are on the Property Map tab
2. Ensure that the property map object list has first been loaded
3. Select an object in the list view
4. Once an object has been selected, the right ⅔ of the screen will inflate with a property configuration layout customized for the specific object you selected
    a. NOTE: you may see options available for certain objects that are not available for others
5. Use the configuration layout to change any of the editable properties of the object you’ve selected
6. Changes will be seen in real time to the scene

To load the scene graph object list:

NOTE: You must be first connected to VirtuTrace to be able to load the scene graph object list

1. Ensure you are on the Scene Graph tab
2. Press button that says “Download Scene Graph”
3. The list view on the left ⅓ of the screen will populate with scene graph objects if available

To edit the properties of a scene graph object in a VirtuTrace scene:

NOTE: You must be first connected to VirtuTrace to be able to load the scene graph object list

1. Ensure you are on the Scene Graph tab

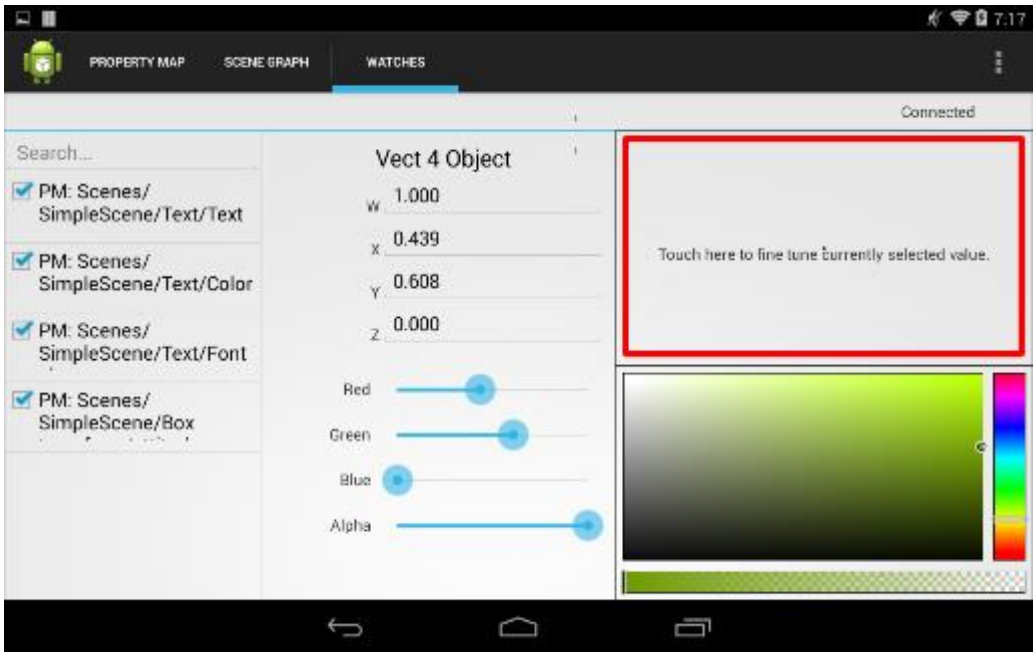
2. Ensure that the scene graph object list has first been loaded
3. Select an object in the list view
4. Once an object has been selected, the right ⅔ of the screen will inflate with a property configuration layout customized for the specific object you selected
    a. NOTE: you may see options available for certain objects that are not available for others
5. Use the configuration layout to change any of the editable properties of the object you've selected
6. Changes will be seen in real time to the scene

To add watched objects to the Watches list (in the Watches tab)

NOTE: You must first be connected to ViruTrace to be able to load the watches object list

1. From either the Property Map tab or Scene Graph tab, check the checkbox next to any item in the list view
2. You will now be able to see the object in the Watches tab

How to use the radial dial widget to adjust values:



1. Begin by highlighting a editable numerical text box
    a. Such boxes include INT and FLOAT
2. Once a editable numerical text box is highlighted, tough somewhere in the highlighted RED box
3. Once touched, a circle will form with a radial size of the distance between your finger and the vertex of the circle
4. Slide finger to begin adjusting values
    a. Closer to the vertex of the circle will result in larger increments of change while further from the vertex will result in smaller increments of change

5. For optimum results, follow path of circle while adjusting values

The next section shows how to make technical changes to both VTRemote as well as VirtuTrace to accommodate new types and operations for VTRemote.

Changes Required to VirtuTrace Application for OpenSceneGraph:

1. In virtutrace/remote/Messages.h
    - Add a string that represents a new operation to be supported and a description of the arguments it takes
    - New Node subclasses do not need to be added here
2. In WorkVisitor.h
    - If a new Node subclass is to be supported, an “apply(NewNodeSubClass)” method must be declared; includes are added as necessary to support the new type
    - New operation types do not need to be added here
3. In WorkVisitor.cpp
    - To support a new type or add new operations, a format similar to existing types should be used in the “apply(NewNodeSubClass)”, which must do the following:
        1. Use the “DebugFunctionGuard()” method
        2. Get the node id (the function automatically creates one if none exists)
        3. Iterate through the WorkOrder queue
        4. Process orders with an id that match the current node
        5. Supported operations are checked in an if/else if block using the string constants defined in Messages.h
        6. The processing block should throw a relevant exception type for any problems with a command issued for the node
        7. The catch block must report errors
        8. Remove orders from the WorkOrder queue after they are processed and increment the orders run for the traversal
        9. Traverse more of the graph as allowed via the “canTraverse()” method

Changes Required to VTRemote Android App for OpenSceneGraph:

1. In “res/layout/”
    - New xml layouts must be created to support new types; it is suggested to copy the layout and remove items as necessary. Files should be named after the type they are for; e.g. Geodes have a “geode.xml” and MatrixTransforms have a “matrix_transform.xml”
2. In “src/com/vrac/virtutrace/vtremote/VTRUtils.java”
    - Add a string that represents the operation to be supported and a description of the arguments it takes
3. In “src/com/vrac/virtutrace/vtremote/list/SceneGraphListItem.java”
    - To add new types the “getFragmentFromType()” method must check its type and return a new fragment of the relevant type
4. In “src/com/vrac/virtutrace/vtremote/fragments/scenegraph/”

- New types require a fragment to support their edits and must extend from SceneItemFragment; e.g. “MatrixTransformFragment.java” and “GeodeFragment.java”
- The rootView of the fragment must be inflated the layout defined in “res/layouts”
- The fragment must contain the objects necessary to edit its values; e.g. “EditText anEditTextName”
- References must be assigned to edit objects in the “onCreateView()” method; e.g. an EditText field would be assigned as “EditText anEditTextName = rootView.findViewById(R.id.anEditTextName)”
- The “onCreateView()” method must call “initView()” and return the rootView
- The “initView()” method will set default loading text fields in edit objects using a Bundle
- The “updateView()” method will take a Node created by the OsgParser and use the data inside to fill the edit objects; the OsgParser will most likely need helper methods to retrieve the specific data from the XML response (see OsgParser.getMatrix() or .getColor() for an example).
- The “updateView()” method must also add listeners to the relevant edit objects only once per fragment instance; this is determined using the “dataLoaded” property of SceneItemFragments
- Watchers must extend their relevant edit object watcher counterparts and use “VTRUtils.sendOsgCommand(nodeId, VTRUtils.command_string, argument_string)” to send commands to VirtuTrace

5. In “src/com/vrac/virtutrace/vtremote/OsgParser.java”
    - New helper methods need to be added to retrieve specific data from XML nodes

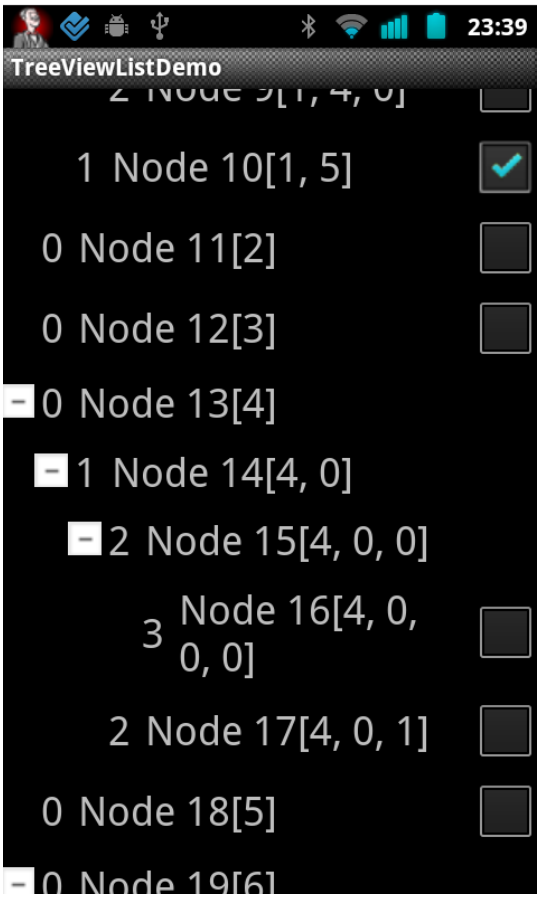
Changes Required to VirtuTrace Application for PropertyMap:

1. In RegisterTypes.h
    - If a new type is to be supported, a “vt::SerializableAny *type*ToSerializableAny(std::string newValue)” method must be declared; includes are added as necessary to support the new type
2. In RegisterTypes.cpp
    - A method matching the one declared in RegisterTypes.h must be implemented so as to return a SerializableAny for usage in the PropertyMap.
    - The method must then be registered in the registerAll() function in the form of PROPERTY_FACTORY.registerPropertyType("*demangleTypeName*",boost::bind(*type*ToSerializableAny,_1));

Appendix II: Alternative Designs

List View

As seen previously, the final user interface design shows the use of a list view as the main means of navigating through a scene to find objects. Originally, our idea was to use a tree structure because the objects in a scene are structured in a hierarchal manner.



Here is a mockup of an external plugin that we had planned to use:
Note: the plugin is known as “tree-list-view-android” and is available at the following URL:
https://code.google.com/p/tree-view-list-android/

As you can see above, the plugin could have been used to show a tree structure instead of showing a list view. However, after further discussion with the client, we felt that this view was not what was desired visually, so the tree structure was scrapped and the use of multiple listviews was implemented.

HTML5 vs Android

One question we have received from several sources is the reason we chose to create an app for a specific platform (i.e. Android) versus developing a cross-platform solution using something like HTML5 for use within a browser. This question is entirely valid and this alternative was briefly considered; however, we ultimately chose Android and believe it was the right choice.

As previously mentioned in this document, a main factor in our choice for using the Android platform was because our client had specifically requested it. Additionally, Android provides a lot of built-in functionality that is either simply not present in HTML5 or not widely supported by the most popular internet browsers. One example of this missing functionality is that there is not a clean way to create a persistent background thread which was necessary for our networking requirements. This realization was really the “nail in the coffin” and the last piece of evidence we needed to finally decide on Android.

Appendix III: Other Considerations

We would like to describe in this section some challenges that we encountered in the development of this application.

With the inception of the idea for this application, the client made it clear that this was something that had not been done before. While an iOS application existed that performed functions similar to VTRemote, it's functionality was not comparable, and as such, provided no aid in development. The biggest issues we had with such a project was any issue that you would see in a typical R&D (research and development) project. Not only did we have to implement the Android application, we also had to facilitate changes to VirtuTrace to allow for real time, direct communication between the Android end and the C++ (VirtuTrace) end. Thus, as we described before, our team was split amongst those of us working on Android and those working on the VirtuTrace end.

Any time multiple teams are working on multiple aspects of a project, issues arise with communication. Those on the Android side sometimes had difficulty implementing features without first knowing if the VirtuTrace side had its end of the necessary features complete. Thus, communication during this project became very vital early on.

We also did not know, especially during the first and much more theoretical semester, if a lot of the things we wanted to do (in terms of communication between the Android end and VirtuTrace end) was possible, and thus we had to do a lot of testing and debugging to ensure that communication was even going to be possible before we moved forward. As stated before, this was a total R&D project, so during the first quarter of the project lifecycle, we were designing our application on the basis that communication would be possible without ever ensuring, which as you can imagine, was frightening in it's own sense. However, due to technical assistance from staff of the VRAC and our technical advisor Kevin Godby, we were provided assurance that the features we wanted to implement would be possible.

Overall, the project was highly successful and we were able to realize all major features requested by the client.