Final Document

VRAC Tactile Vest
Iowa State University
May 14-23

Team Members

Ryan Haack
Garrett Phelps
Ben Andry
Cyle Dawson
Jacob Cramer

Client/Advisor

Dr. Stephen Gilbert

Table of Contents

Project Design

Problem Statement

The VRAC is currently receiving funding from the United States Army for a virtual reality training simulator. The MIRAGE a mixed-reality research lab fitted with IR sensors and a fully-functional game engine. Applicants can use this simulator to experience combat simulations at a much lower cost and setup time than setting up an environment with paid actors.

The end goal of this project is to provide an API to communicate with physically-responsive hardware (called tactors) that are placed in particular configurations on applicants, and a graphical UI that will allow users to easily configure tactor placements virtually, as well as command the tactors (which will internally use our API). The current set-up uses a brand of restaurant pagers attached to vests, but interest in extending use-cases was expressed from the very beginning.

Along with the potential extra use-cases in mind, we were also instructed to leave room for the possibility of a unique concept of "patterns" that would allow for certain formations of tactor to be activated in a certain manner (e.g. a "star burst" pattern). The UI developed would also be responsible for configuring these patterns.

Finally, the API needed to be intuitive and complete enough to be "plugged-in" as necessary. In particular, we were told that the API would be used by the game engine to notify applicants when they were shot. Although we were not responsible for the calculations and determinations of which tactors to activate, we had to make sure that the API could be used in a way that was simple in this context.

Functional Requirements

Version 1 (December 2013)

V1.1 User will be able to choose from predetermined locations for the tactors on the vest/body
V1.2 User will be able to send commands with predefined buttons
V1.3 User can change the intensity of the vibration for each tactor
V1.4 Tactors must be configured before use

Version 2 (May 2014)

V2.1 User can place pagers to any appropriate location
V2.2 User can make custom patterns with UI

V2.3 User can make custom tactor layout and save custom tactor layout
V2.4 User can use other tactile attire (belt, wrist)
V2.5 Interface must be able to be "plugged" into any platform or system

Non-functional Requirements

- Detailed documentation, every method declaration and class
- Quick response time to signals sent to tactors
- Use the fewest possible transmitters to keep price low
- Basic soldier 3D model

Uses Cases

- User adds a specified tactor
- User drags a tactor to a part of the body
- User sends a signal to a tactor
- User clicks on "tactor Test" to test configuration
- User deletes tactor from body

Operating Environment

The primary operating environment for our program will be Linux but should be compatible with Windows. We will create a serial connection to the tactors and write our API using C++. The user interface will be coded using the Qt application to help us achieve our cross compatibility goal. Our API will be the communication between the MIRAGE game engine and the tactile vest.   Our API will be fully compatible with any application in need of serial connection to a device.

Assumptions and Limitations

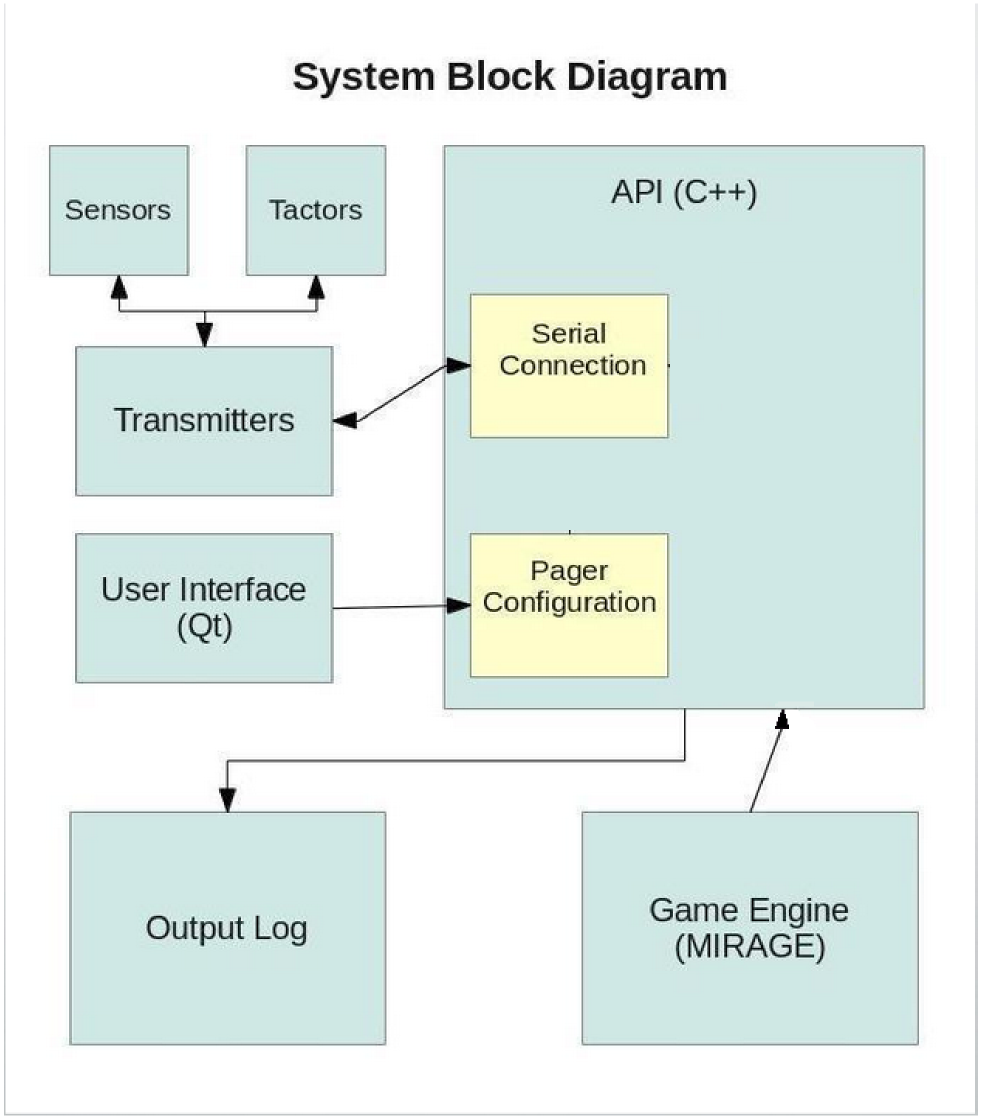
Assumptions

- VRAC has set up a git repo
- tactors, transmitter, and military vest will be provided
- Simultaneous signals can be sent with better hardware

Limitations

- Vibration strength of tactors
- Hardware restrictions for simultaneous signals
- Operating range of tactor approximately ¼ mile
- Battery life of tactor approximately 48 hours
- Export control regulations limit us to showing source code to only US citizens
- Final project deadline is May 2014

System Block Diagram



Block Diagram Description

Sensors - Obtains data from the subject and sends back to transmitter
Tactors - Responds to data from transmitter (ex. tactor)
Transmitters - Sends and receives data through serial connection
API - Specifies interaction of the game engine, user interface, and transmitters
Serial Connection - Takes the data from the game engine and pager configuration, converts the data into a serial format and sends it to the transmitter
Pager Configuration - Stores the layout of the tactors as received from the user interface
Game Engine - the MIRAGE environment
Output Log - A log of the events occurred and command given during the use of the API

Specification

Module Description

The API will control and access the hardware attached to our tactile attire (i.e. the tactile vest). The design will be simple, extensible, and minimal in assumptions.

On the lowest level, there are tactors and sensors. Users must be able to command tactors and check sensors. These tactors and sensors will be attached to some unit of attire. Each piece of hardware will be identified with a name, so when the user attempts to communicate with the hardware, he or she can do so without having to have direct handles to the hardware or remember complicated IDs.

On a higher level, users would much rather use a more abstract way of controlling the hardware. In particular, users would like to define a high-level pattern that describes which tactors need activated, potentially including a complex sequence of simpler patterns. Internally, these patterns would know which specific hardware to control and commands to run.

The interfaces are designed as follows:

Name: Sensor
Purpose: Read from a specific hardware sensor (i.e. temperature sensor)
Description: Sensor corresponds to a physical sensor and is responsible for retrieving data from it when check is called. In most cases a sensor would be associated with an Attire.
Methods:

- String:check() - Read the state of the sensor
    - Return the state

Name: Tactor
Purpose: Control a specific hardware tactor (i.e. tactor)
Description: Tactor corresponds to a physical tactor, such as a tactor, and is responsible for sending commands to that Tactor when command is called. In most cases a Tactor would be associated with an Attire.
Methods:

- boolean:command(String:command,HardwareInterface:device) - Send a command to the tactor via the HardwareInterface
    - Return true if the command successfully executed, false if the command

failed to execute.

Name: Attire
Purpose: Model a physical unit of attire (i.e. tactile vest)
Description: Attire corresponds to physical attire, such as a vest or belt. Such physical attire would have a number of tactors and/or sensors integrated with it. Attire will be able to add and remove both Tactors and Sensors, execute a pattern of Tactor commands, command a single Tactor, or check Sensors.
Methods:

- constructor(HardwareInterface:device) - Create an Attire with the specified hardware interface
- Tactor:addTactor(String:name, Tactor:tactor) - Add a Tactor with a specific name
    - Return the previous Tactor with that name, or void if none existed
- Tactor:removeTactor(String:name)
    - Return the removed Tactor, or void if no Tactor by that name existed.
- Sensor:addSensor(String:name, Sensor:sensor) - Add a Sensor with a specific name
    - Return the previous Sensor with that name, or void if none existed
- Sensor:removeSensor(String:name)
    - Return the removed Sensor, or void if no Sensor by that name existed.
- boolean:command(String:tactorName, String:command) - Send a command to a Tactor
    - Return true on command successfully executed, false if the command failed to execute.
- String:check(String:sensorName) - Read the state of a Sensor
    - Return the state
- boolean:execute(SimplePattern:pattern) - Execute a command pattern
    - Return true if the SimplePattern was executed successfully, false if the SimplePattern failed to execute.
- boolean:execute(ComplexPattern:pattern) - Execute a series of command patterns
    - Return true if the ComplexPattern was executed successfully, false if the ComplexPattern failed to fully execute.

Name: HardwareInterface
Purpose: Provide an interface for Attire to use to communicate with hardware (e.g. the transmitter via Serial)
Description: HardwareInterface corresponds to the connection to a physical device (e.g. transmitter) through which commands are sent to Tactors and Sensors. HardwareInterface

will be able to open and close a connection to a physical device, send data to that device, in the form of a C string, and read from that device.
Methods:

- boolean:Open(char*:toOpen) - Open a connection to the hardware device given by toOpen
    - Return true if the connection was successfully opened, false if the connection failed
- boolean:Close() - Close the open connection to the hardware device
    - Return true if the connection was successfully closed, false if closing the connection failed or if the connection wasn't opened in the first place
- boolean:SendData(char*:dataToSend) - Send the data given by dataToSend to the hardware device that is currently open
    - Return true if the data was successfully sent, false if the data failed to send
- String:ReadData() - Read data from the hardware device that is currently open
    - Return the data from the hardware device as a string or an empty string if the read failed

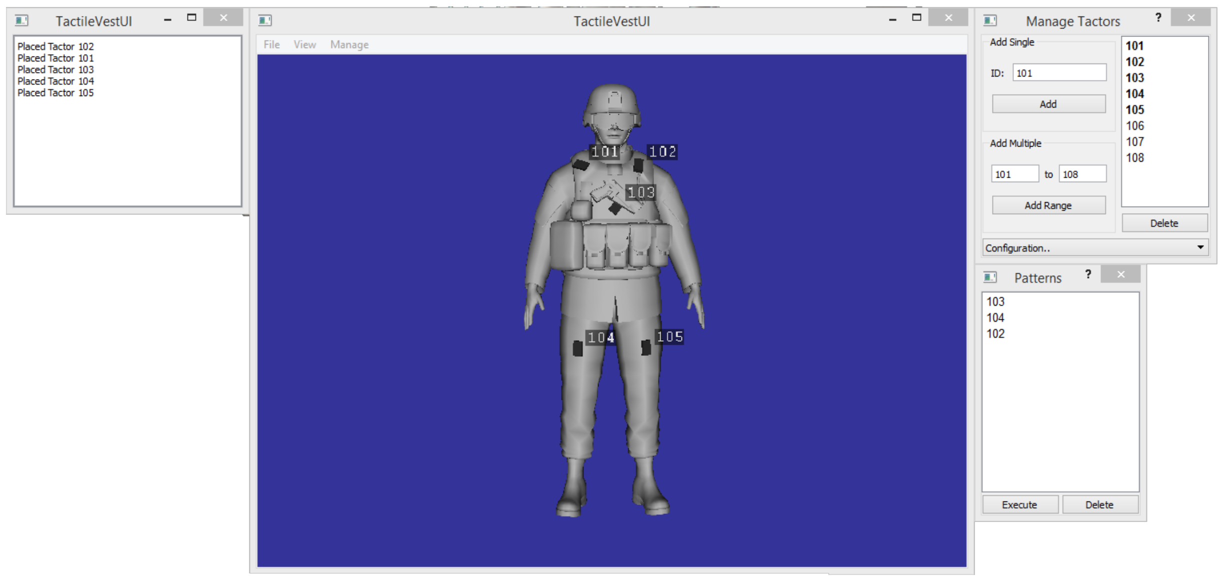
Name: SimplePattern
Purpose: Describes a set of Tactors to be commanded simultaneously
Description: SimplePattern corresponds to a series of Tactors to be given the same command simultaneously (or potentially in sequence as the hardware demands).
Methods:

- constructor(String:command) - Create a pattern with a specific command
- boolean:addTactor(String:tactorName) - Add the name of a Tactor to this pattern
    - Return false when a Tactor with that name already exists in this pattern, false otherwise.
- boolean:removeTactor(String:tactorName) - Remove the Tactor with the given name from this pattern.
    - Return true if the tactor was successfully removed, false otherwise.
- String:getCommand() - Get the command to run on the set of Tactors
    - Return the command
- String[]:getTactors() - Get the set of Tactor names to run the command on
    - Return the names of the Tactors

Name: ComplexPattern
Purpose: Describes a set of Patterns to be run asynchronously
Description: ComplexPattern describes a series of SimplePatterns to be run in sequence. The SimplePatterns are stored according to their delay after the start of the ComplexPattern.
Methods:

- Pattern:addPattern(int:time, SimplePattern:pattern) - Add a pattern to be run at a certain delay
  - The delay is measured from the start of the ComplexPattern execution, not the previous pattern execution
  - Return the Pattern previously set to run at this time, or void if none existed
- Pattern:removePattern(int:time,SimplePattern:pattern) - Remove a SimplePattern from the ComplexPattern.
  - Return the removed SimplePattern, or void if no pattern with that mapping exists.
- Map<int, Pattern>:getPatterns() - Get the patterns to execute and the delays at which to execute them
  - Return a Map of Patterns keyed by delay

User Interface Description



```
Placed Tactor 102
Placed Tactor 101
Placed Tactor 103
Placed Tactor 104
Placed Tactor 105
```

ID: 101

101 to 108

```
101
102
103
104
105
106
107
108
```

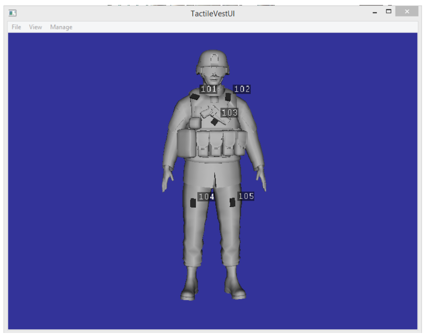
Configuration..

```
103
104
102
```

Using the powerful Qt framework, we were able to create a cross-platform, highly-functional “control center.” This application shows one way the TactileAPI could be used by allowing users to create and test tactor configurations. The first version consisted of a 2D image of a soldier, and a drag-and-drop interface. The user could add tactors to the configuration by dragging them to predefined locations on the image. The user could also trigger just one or all of the tactors added. See Appendix II for a full description of the original UI implementation.

Although this design was functional, we recognized the limitations fairly quickly. After speaking with the client, we determined that a more flexible implementation using a 3D model would be more adequate. We decided on using the popular OpenSceneGraph library, since it's capabilities and compatibility with the Qt framework made it ideal for our needs. In order to make space for the central component (the 3D model), many of the other UI components were broken out into their own windows or dialog boxes. The result was a more usable application, with each component having a clear distinction and purpose from the next.

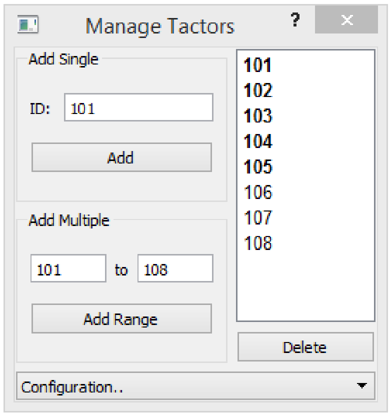
Configuration View



The primary focus of the application is the component we call the "Configuration View." Using this, users can add tactors to the Model by right-clicking anywhere, opening the Tactors menu, and selecting a tactor from the list. This list is based on tactors added to the tactor manager (described next), excluding tactors already added to the Model.

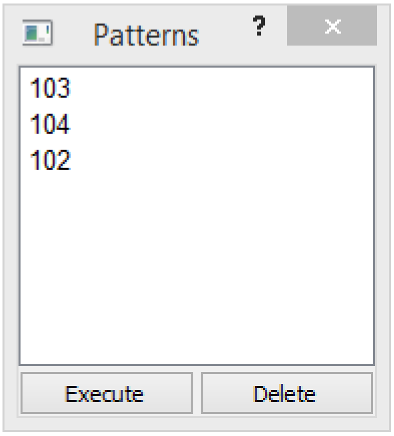
Right-clicking on a placed tactor brings up a menu describing things you can do with that tactor. Although limited in the current version, many more actions can be added. Each tactor on the Model has an identifying label, corresponding to the identifier given to the tactor when added to the tactor manager. Using these identifiers, users can easily see which tactors they are interacting with.

Tactor Manager



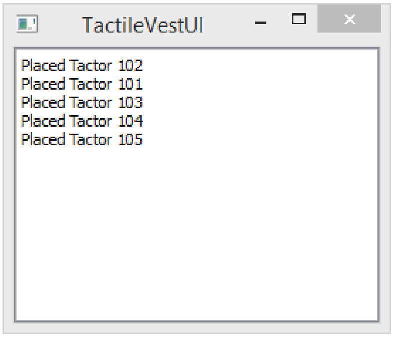
The "Tactor Manager" allows users to define what tactors are available for the configuration. Tactors can be added or removed as desired, with the option to add a range of tactors at once. This feature is particularly useful for trying out brand new configurations. Tactors already placed on the Model are bolded, so they are easily distinguishable. Configurations can also be loaded on the fly with the drop-down at the bottom of the window.

Pattern Manager



Patterns are a concept that allows the user to define a set of tactors to command as a unit. The "Pattern Manager" provides an intuitive way to use this feature. Due to hardware limitations, the TactileAPI can only control one of the tactors available at a time, so using this feature will activate tactors sequentially. However, the TactileAPI does not contain this limitation, thus allowing the Pattern Manager to become more powerful as better hardware options are explored. More information about handling these hardware limitations is provided in Appendix III.

Action Log



```
Placed Tactor 102
Placed Tactor 101
Placed Tactor 103
Placed Tactor 104
Placed Tactor 105
```

As the user works through this tool, certain actions are documented. This "Action Log" provides a view for the user to see what actions are being documented. Not only does this provide an accurate history of what the user has done, but this also allows the user to see actions performed by the software, as well as extra information about the actions taken. This feedback can help the user understand what goes on under the hood as the configuration takes shape.

Save/Open Configuration

Having to redo every configuration every time would be a frustrating task, so another important feature to note is the ability to load and save configurations. Using this feature, the state of all of the tools is saved in a standard XML format. The portability of this format allows configurations to be saved on one computer and opened on another, even if it's on a different operating system. This keeps the application cross-platform, which was one of the major goals from the very beginning, as discussed in the "Operating Environment" section of this document.

Testing

Test Specification

The following will be tested and verifies the correctness of the application

- Verify that a user can add tactors and then move them onto the body
- Verify a user can signal the desired tactor
- Verify that two tactors buzz at the same time when simultaneous signals are sent

- Verify a user is able to create a pattern of tactors to be signaled
- Verify that a user is able to save patterns for future sessions
- Verify that a user is able to move/remove tactors from the body
- Verify the output file matches the commands used during the session
-
- Verify that user outside of our team can complete all previous stated test cases

Tests

Stress Tests

When the project was started we started right off with stress testing the hardware of our paging system. We needed to ensure that our hardware was sufficient enough to complete this project.

To test this we attempted the follow:

1) Rapidly sending signals off to a single pager
2) Sending signals to multiple pagers simultaneously
3) Sending signals to multiple pagers sequentially in increasing intervals of 1,2, and 3 seconds

Results:

1) There is handshaking involved with the pager and the transmitter resulting in the pager ignoring all proceeding requests until it was finished with its first.
2) This was similar to the first result except that the transmitter was waiting for the reply from the previous pager before it attempted the next pager sometimes resulting in erroneous results.
3) Same of result 2 except with 1 second intervals the signals were being received correctly.

What this means is the hardware is not essentially not able to fully accommodate this project. We reached out to the LRS company in attempt to remedy this problem, but in the end we were instructed by our advisor/client to continue on with the current hardware, but adapt the API to accommodate similar transmitter and pager systems.

User Acceptance Tests

The client instructed us that we must be able to hand out our API with some minimal instructions to people involved at VRAC and have them send off basic commands to the pagers. In order to complete this we created some step-by-step user's manual to write some API commands to signal the pagers. We will then give these instructions to someone for testing.

Implementation

Basic Building Blocks

- C++ - Overall Header file for the API with a wrapper for both Linux and Windows libraries
- Qt Creator for the UI

Familiarity with platforms/tools

Languages

- **C++ -** The whole group is familiar with C code and object oriented programming. The project will be written in C++ and the majority of the group feels comfortable with the language.

Tools

- **Qt Creator** - We will be using Qt Creator to create the user interface for our project. None of us have used Qt Creator and Garrett and Ryan will focus on learning and using this new technology.
- **Git** - We are all familiar with and will use a Git repository for version control.
- **Redmine** - Our project has also been set up with a Redmine account where we will keep track of issues and current tasks regarding our project.
- **OpenSceneGraph** - This will allow us to show the user a 3D representation of a model.

Platforms

- **Windows/Linux/Mac -** We are all comfortable with windows but some of us are not familiar with linux or mac.This project will be cross platform but is intended to run on Linux. We will run and test our program on both a windows environment and a linux/mac environment. We learned from our initial testing that serial connection libraries are platform specific. We will have to use two separate libraries and at the same time make sure they don't interfere with each other.

Standards

C++11

C++11 extends the C++ standard library includes the core language of C++. This standard was important for our project because of the multithreading support.

Army Confidentiality

This project has received funding from the United States Army therefore there were privacy agreements we had to follow. Every member of our team had to be a legal United States citizen and we must not show any of our source code to any non United States citizens. We have made sure to not post any of our code in any public domains.

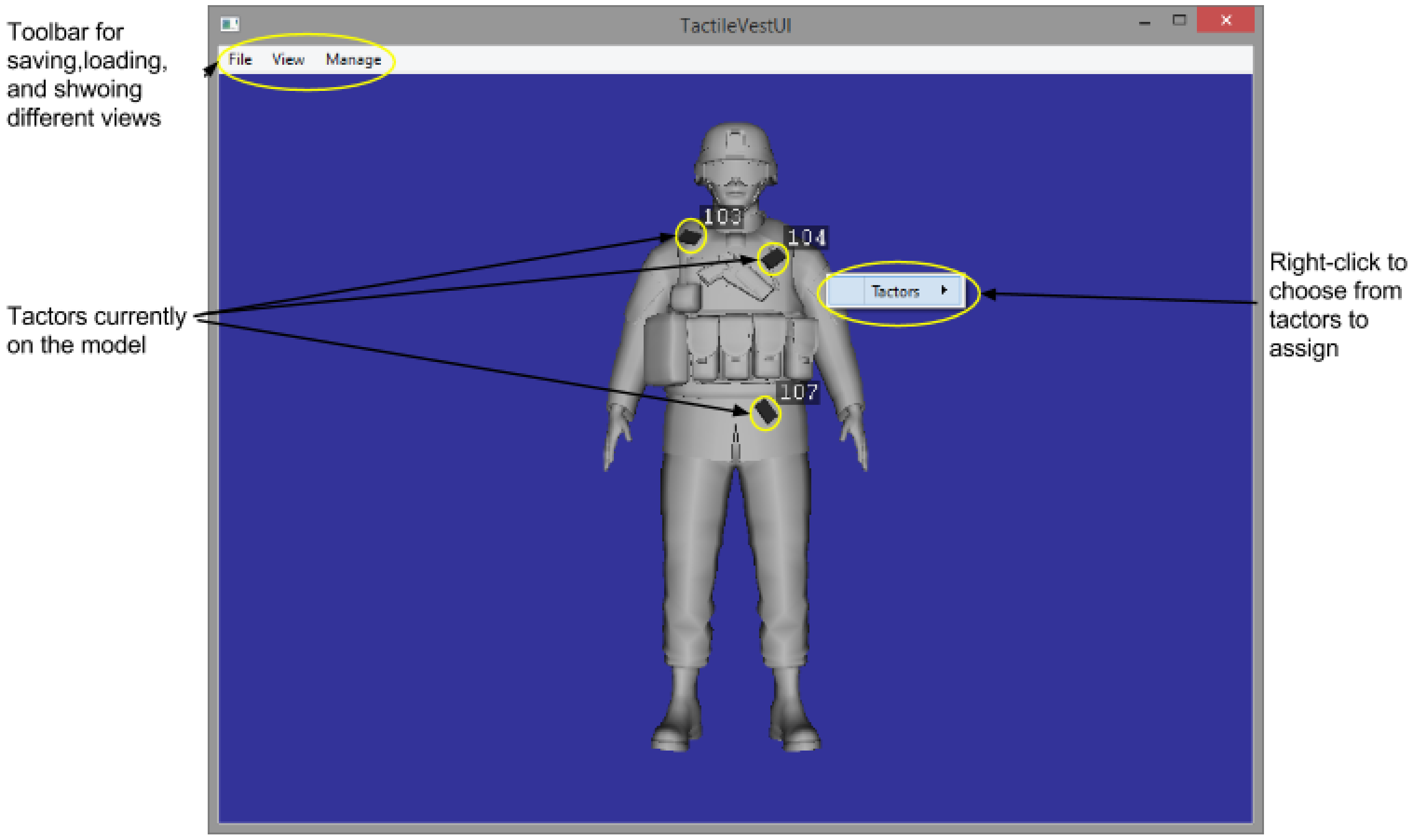
Universal Serial Bus (USB)

USB is a hardware communication standard for many types of electronics. In our case, the T74USB transmitter used for serial communication to the pagers. USB is common port on most computers so the transmitter may be used with most computers with the correct driver installed.

Appendix I: User Guide

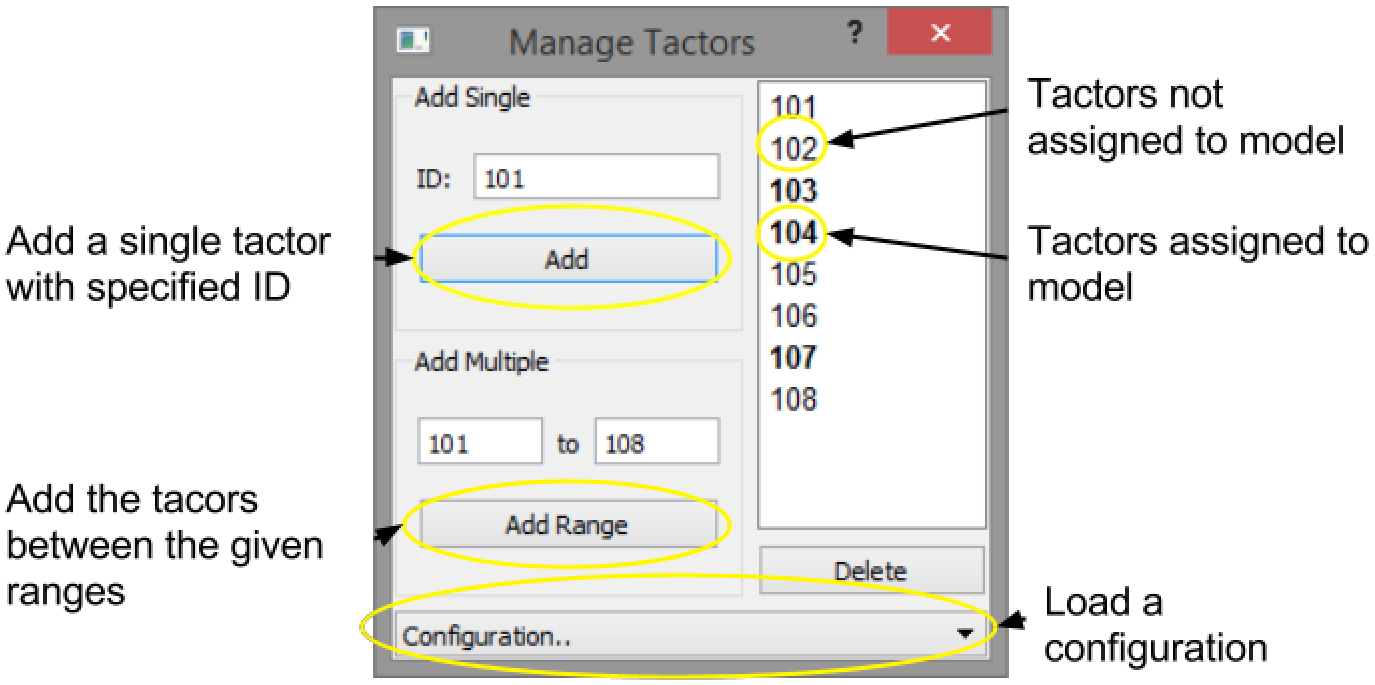
VRAC TactileVest UI Instructions

Main window


Toolbar for saving,loading, and shwoing different views
TactileVestUI
File
View
Manage
103
104
107
Tactors
Tactors currently on the model
Right-click to choose from tactors to assign

Shows the current model loaded and all tactors attached to the model.

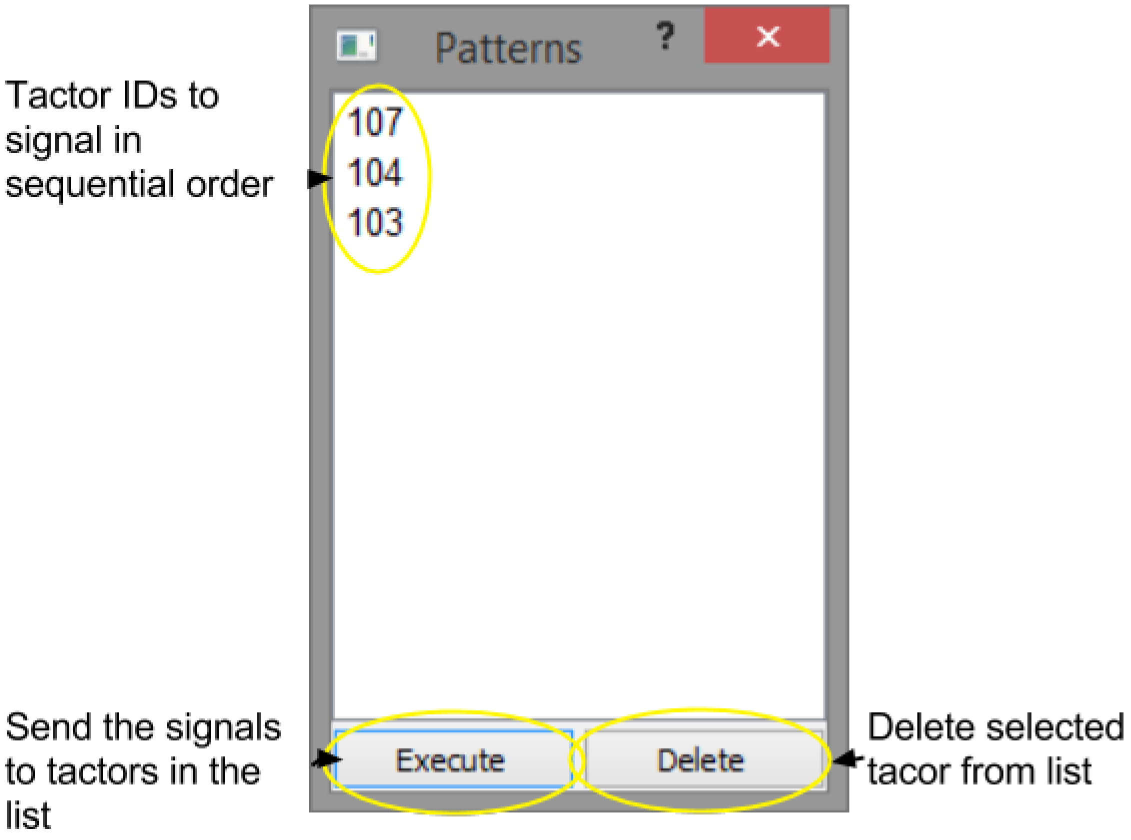
Manage Tactor Window


Manage Tactors
Add Single
ID: 101
Add
Add Multiple
101 to 108
Add Range
101
102
103
104
105
106
107
108
Delete
Configuration..
Tactors not assigned to model
Tactors assigned to model
Add a single tactor with specified ID
Add the tacors between the given ranges
Load a configuration

Used to create tactors to add to the model and also manage them. Has an option to load predefined configurations.

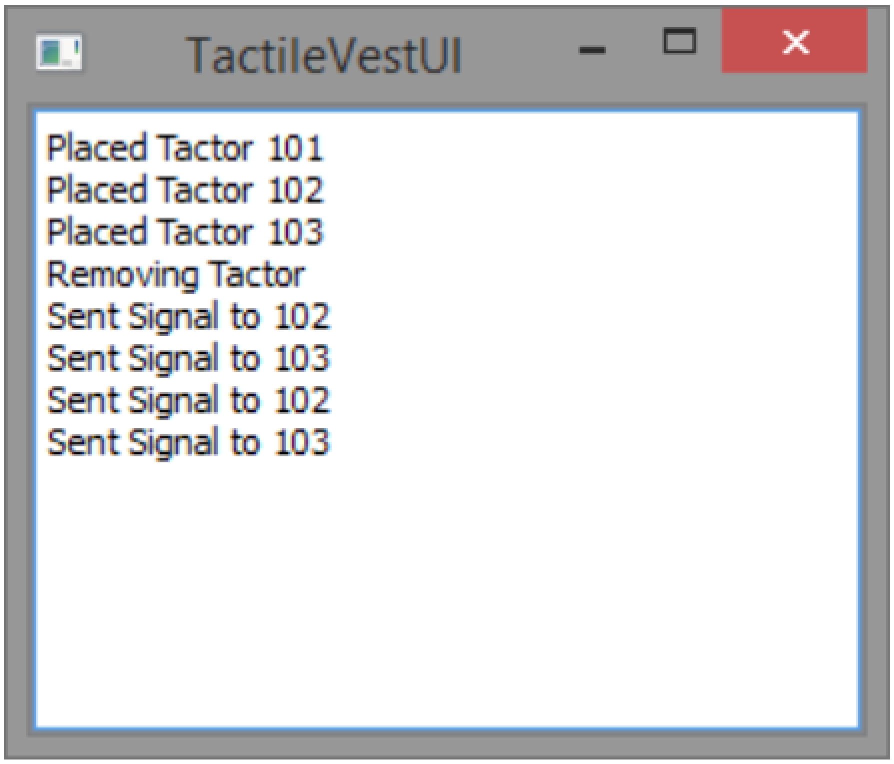
Patterns Window

Stores the patterns to execute in sequential order.


Patterns
?
×
107
104
103
Execute
Delete
Tactor IDs to signal in sequential order
Send the signals to tactors in the list
Delete selected tacor from list

Log window

Stores all actions completed in the current session.

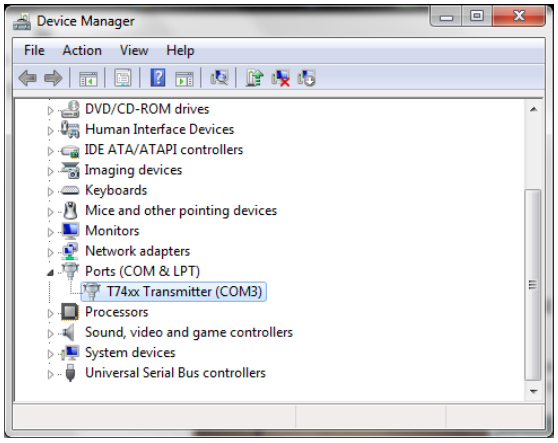


```
Placed Tactor 101
Placed Tactor 102
Placed Tactor 103
Removing Tactor
Sent Signal to 102
Sent Signal to 103
Sent Signal to 102
Sent Signal to 103
```

VRAC TactileVest API Instructions

Hardware Setup

- Go to pager.net/support/ and download the T74USB driver
- Plug the T74USB transmitter into a USB port on your computer
- Note the associated COM port with Device Manager

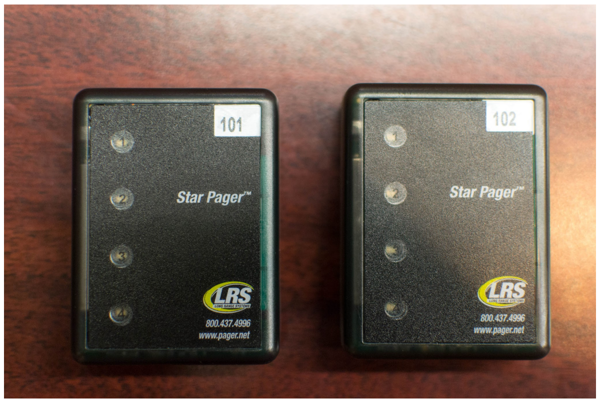


Software Setup

- Include the header file TactileAPI.h
- Make sure to compile with C++11 (set flags -std=c++0x or -std=c++11)

Pager Notes

- The ID of the pager is located on the upper right corner of the pager.



- This ID may not be correct if someone has previously reprogrammed the pager to another ID.
- The pagers we currently have cannot be signalled one after another very well. There is a period of time after a page is sent that you must wait for the pager and transmitter to perform their “handshake”. We believe the transmitter is waiting for some kind of reply signal from the pager.
- We recommend waiting at least 1 second before paging another pager.

Example Code to Page a Pager

Option1 is the most basic way to page one pager, while Option2 contains the concept of an attire object that is able to contain multiple pagers.

Option 1:

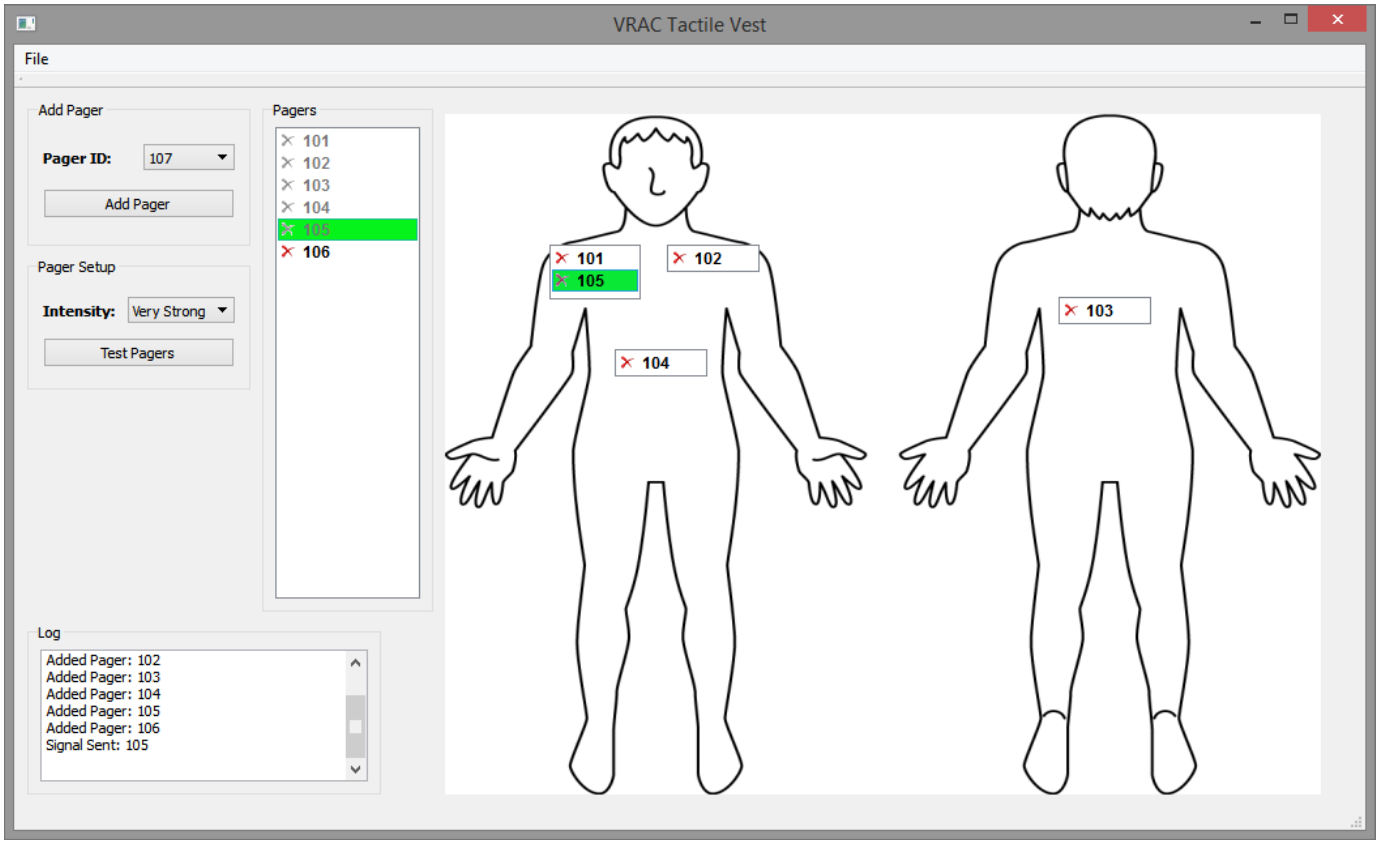
```
//Create a Serial object
Serial serialObj;

//Open the Serial object with the correct COM port parameter
serialObj.Open("COM1");

//Create a pager object and give it the ID of the pager you want to page
Pager pagerObj("101");

//Call the command function on the pager with the vibration intensity and
passing it a reference to the serial object. //Vibration intensity ranges from
1 to 4 (4 = highest)
pagerObj.command("3", &serialObj);

//Close the serial object
serialObj.Close();
```

Option 2:

```
//Create a Serial object
Serial serialObj;

//Open the Serial object with the correct COM port parameter
serialObj.Open("COM1");

//Create an Attire object and give it a reference to the Serial object
Attire attireObj(&serialObj);

//Create a pager object and give it the ID of the pager you want to page
Pager pagerObj("101");

//Call the addTactor function. Pass in a name and the pager object
attireObj.addTactor("pagerName", &pagerObj);

//Call the command function on the attire object. Pass in the name of the
pager to pager and vibration intensity
//Vibration intensity ranges from 1 to 4 (4 = highest)
attireObj.command("pagerName", "3");

//Close the serial object
serialObj.Close();
```

Appendix II: Initial Version

UI Design V1



Pager ID: 107

Intensity: Very Strong

```
Added Pager: 102
Added Pager: 103
Added Pager: 104
Added Pager: 105
Added Pager: 106
Signal Sent: 105
```

The initial UI design was a simple interface that allowed the user to simply drag and drop pagers that they have manually added. The pagers could then be signaled by double clicking on them within the respective spots on the human model. The pagers would also change colors and display where they are on the body when hovering over them with a cursor.

Client's Specifications

The specifications from the client depended on having the pagers be placed anywhere on the human model and also be able to create custom patterns to signal the pagers on the body. It was also specified to be able to change out the models and have the user create, for instance, a belt of pagers that could then be signaled.

Possible UI solutions

To solve the design specifications required by the client we attempted to create a movable list box to contain these pagers and also allow the user to add more list boxes to be placed on the body. This in theory would solve the specification of allowing pagers to be placed anywhere, but would create a drastic problem with space on the model becoming very limited as each spot would contain a list box. It was also very complicated to switch out 2D models and have these list boxes be saved across sessions.

Our final solution was to switch the whole UI to a 3D perspective with a human model that would allow the user to rotate and place pagers with a click of the mouse. The design we strived for was a very modular approach with optional windows around a main scene of the human analog. This would allow the user more freedom and overall a much cleaner look as compared to our V1 design.

Appendix III

What We Learned

More often than not, developers come out of a project wishing they knew at the beginning what they knew at the end. This statement holds true for us. This project had many powerful technologies involved, required specific skill sets, and had to remain abstract enough to remain usable across different environments, with both software and hardware being variable. Facing these challenges, our team learned some valuable lessons.

One lesson learned was if you are using technologies you haven't used before, it's worth the time and effort to play with them outside the scope of the project. Learning to deal with them in smaller, more targeted playgrounds helps learn the quirks of that particular technology so debugging in the grand scheme becomes simpler. This is especially true if you're mixing many different technologies (Qt, OSG, C++11, CMake, etc.), like we did.

Another lesson learned was to test the software on all the systems it is being developed for from the beginning. When issues do arise and a solution cannot be found within a team, it is wise to reach out to others that may have more experience with the technologies being used.

Also knowing each member's strengths and weakness so you know who you can ask for support. That also makes delegation of tasks easier, and make the project move along a little faster.

Future Plans

Plans for future features of the project have been discussed with our client and team.

The first major plan is to have better tactors and/or transmitter that allows for multiple signals to be sent simultaneously. Since this is a future plan that has been discussed multiple times at team meetings, we have set up our API in an abstract manner that it will allow for new equipment to easily be added and set up.

Another major future plan discussed is to add other tactors and sensors. For example, one of the sensors could be a heart rate monitor. Another tactor discussed was a heat based tactor that would add a heated signal to the tactile vest or apparel. Like the first future plan, our API is set up to abstract enough to add different types of tactors and sensors.

For the UI, a feature that has been planned is to add the ability to specify what UI events are logged and displayed in the side log. Another feature for the UI that has been planned to be added is the option to dynamically add and switch models, each with their own pager configuration.

Definition of Terms

VRAC	Virtual Reality Application Center
MIRAGE	Mixed Reality Adaptive Generalizable Environment
API	Application programming interface
Qt	A cross platform GUI application framework utilizing C++
Tactor	A piece of hardware to generate a tactile response.
Sensor	Obtains data from the subject and sends back to transmitter
Serial Connection	The process of sending data one bit at a time
Tactor Layout	The location of tactors on the vest/body
Tactor Pattern	The sequence of the tactors' vibration/lights
OpenSceneGraph (OSG)	Open source library for rendering 3D models with Qt support

Team Information

Name	**Major**	**Contact**
Ryan Haack	Computer Engineer	rahaack@iastate.edu
Garrett Phelps	Software Engineer	gdphelps@iastate.edu
Ben Andry	Software Engineer	bfandry@iastate.edu
Jacob Cramer	Software Engineer	jmcramer@iastate.edu
Cyle Dawson	Software Engineer	cjdawson@iastate.edu