ISU VRAC TACTILE VEST

Team May14-23 : Ben Andry, Garrett Phelps, Ryan Haack, Jacob Cramer, and Cyle Dawson
Client/Advisor: Dr. Stephen Gilbert

Last Edited on **12/4/2013**	**Ryan Haack**	**Version 1.1**

Version 0.1 | Ryan Haack | 10/29/2013
-Initial Document Creation, set up table of contents, outlined and added multiple sections
Version 0.2 | Cyle Dawson, Jacob Cramer | 10/30/2013
-Added API description
Version 0.3 | Garrett Phelps | 11/6/2013
-Moved API description into module description. Added content to User Specification. Added section Test Specification
Version 0.4 | Ryan Haack | 11/7/2013
-Added content to Test Specification and User Specification. Added section System Description
Version 0.5| Jacob Cramer | 11/7/2013
-Added detailed descriptions to module description
Version 1.0 | Ryan Haack, Garrett Phelps | 11/7/2013
-Added use cases, more test specification, formatted numbering
Version 1.1 | Ryan Haack | 12/4/2013
-Added section four, Implementation
Version 1.4| Garrett Phelps | 04/18/2014
-Updates to UI and adding OSG information. Replaced instances of tactor with tactor

Table of Contents

Project Breakdown
- 1.1 Problem Statement
- 1.2 System Description

Design Requirements
- 2.1 Functional Requirements
- 2.2 Non-functional Requirements
- 2.3 Uses Cases
- 2.4 Operating Environment
- 2.5 Assumptions and Limitations
- 2.6 System Block Diagram
- 2.7 Block Diagram Description

Specification
- 3.1 Module Description
- 3.2 User Interface Description
- 3.3 Test Specification

Implementation
- 4.1 Basic Building Blocks
- 4.2 Familiarity with platforms/tools

Definition of Terms

Project Breakdown

1.1 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.

This project is to develop an API for communication to off-the-shelf vibrating tactors on tactile vests that applicants would wear within the MIRAGE. These tactors vibrate whenever an applicant is “shot” within the simulation. The tactors could also be used as communication for navigation. Our main task is to set up an API and UI configuration so the tactors can be placed anywhere on the vest, which will then be able to identify the location of the tactor on the body. The API will be used to send signals to these tactors from a central command *i.e buzz shoulder tactor1*. The API will also need the ability to send multiple signals with predefined patterns and must be generic enough so that it can be easily called from any piece of software as a type of plug-in.

1.2 System Description

The project consists of implementing an API for communication purposes between systems, interfaces, tactors, and sensors. We also have to design and create a custom user interface using the cross platform Qt framework. The project is intended for VRAC personnel and military personnel. The goal is to have an easy to use application that allows an individual to interact with tactors placed on a tactile vest without any first hand knowledge of the code or tactors.

Design Requirements

2.1 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

2.2 Non-functional Requirements

2.2.1 Detailed documentation, every method declaration and class
2.2.2 Quick response time to signals sent to tactors
2.2.3 Use the fewest possible transmitters to keep price low
2.2.4 UI is simple enough to use without any knowledge of the code

2.3 Uses Cases

2.3.1 User adds a specified tactor
2.3.2 User drags a tactor to a part of the body
2.3.3 User sends a signal to a tactor
2.3.4 User clicks on “tactor Test” to test configuration
2.3.5 User deletes tactor from body

2.4 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.

2.5 Assumptions and Limitations

2.5.1 Assumptions

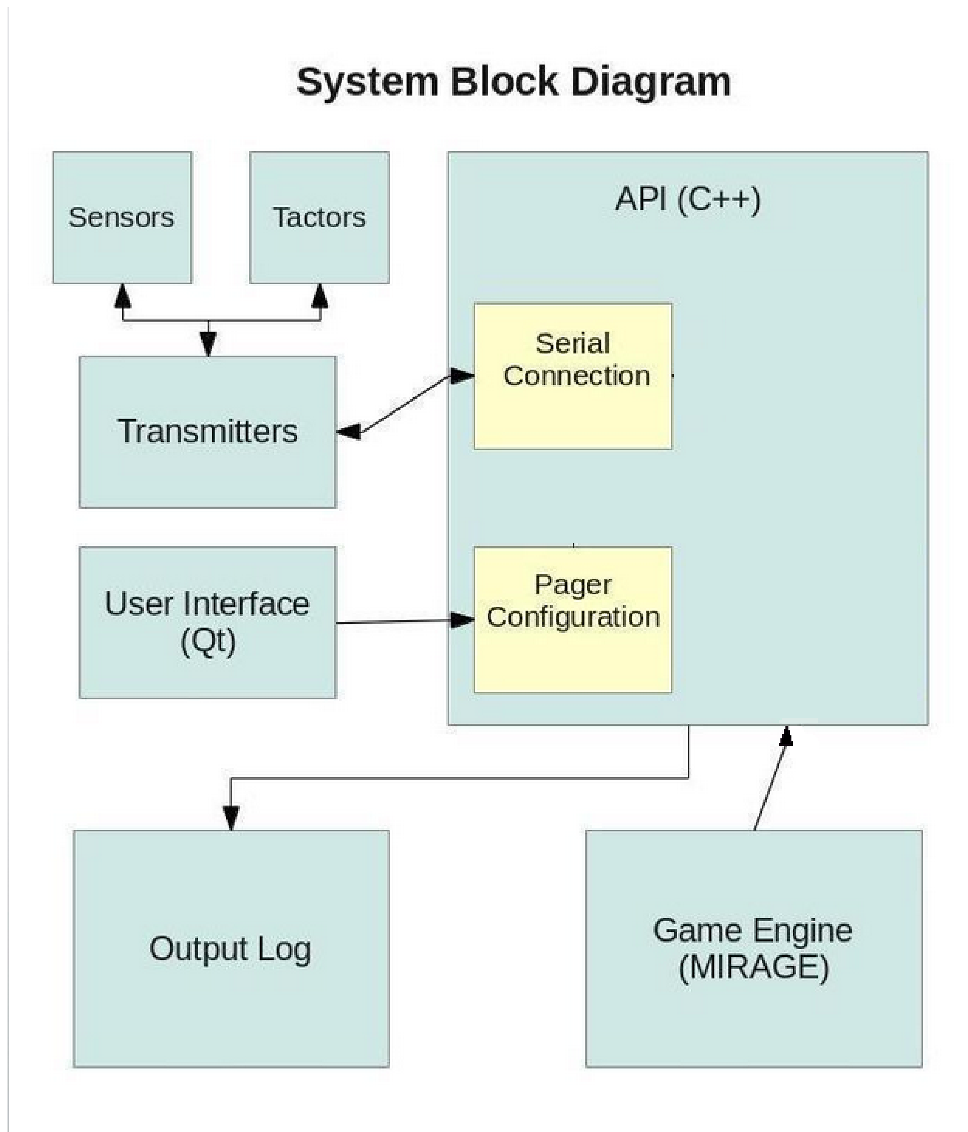
- VRAC has set up a git repo
- tactors, transmitter, and military vest will be provided
- Simultaneous signals can be sent with better hardware

2.5.2 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

2.6 System Block Diagram



2.7 Block Diagram Description

2.7.1 Sensors - Obtains data from the subject and sends back to transmitter
2.7.2 Tactors - Responds to data from transmitter (ex. tactor)

2.7.3 Transmitters - Sends and receives data through serial connection
2.7.4 API - Specifies interaction of the game engine, user interface, and transmitters
2.7.5 Serial Connection - Takes the data from the interpreter, converts the data into a serial format and sends it to the transmitter
2.7.6 Interpreter - Integrates data from the game engine into the tactor configuration and sends it to serial connection
2.7.7 tactor Configuration - Stores the layout of the tactors as received from the user interface
2.7.8 Game Engine Communication - Receives data from the game engine and sends it to the interpreter
2.7.9 Game Engine - the MIRAGE environment
2.7.10 Output Log - A log of the events occurred and command given during the use of the API

Specification

3.1 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) - Send a command to the tactor
    - 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:

- 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: 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:

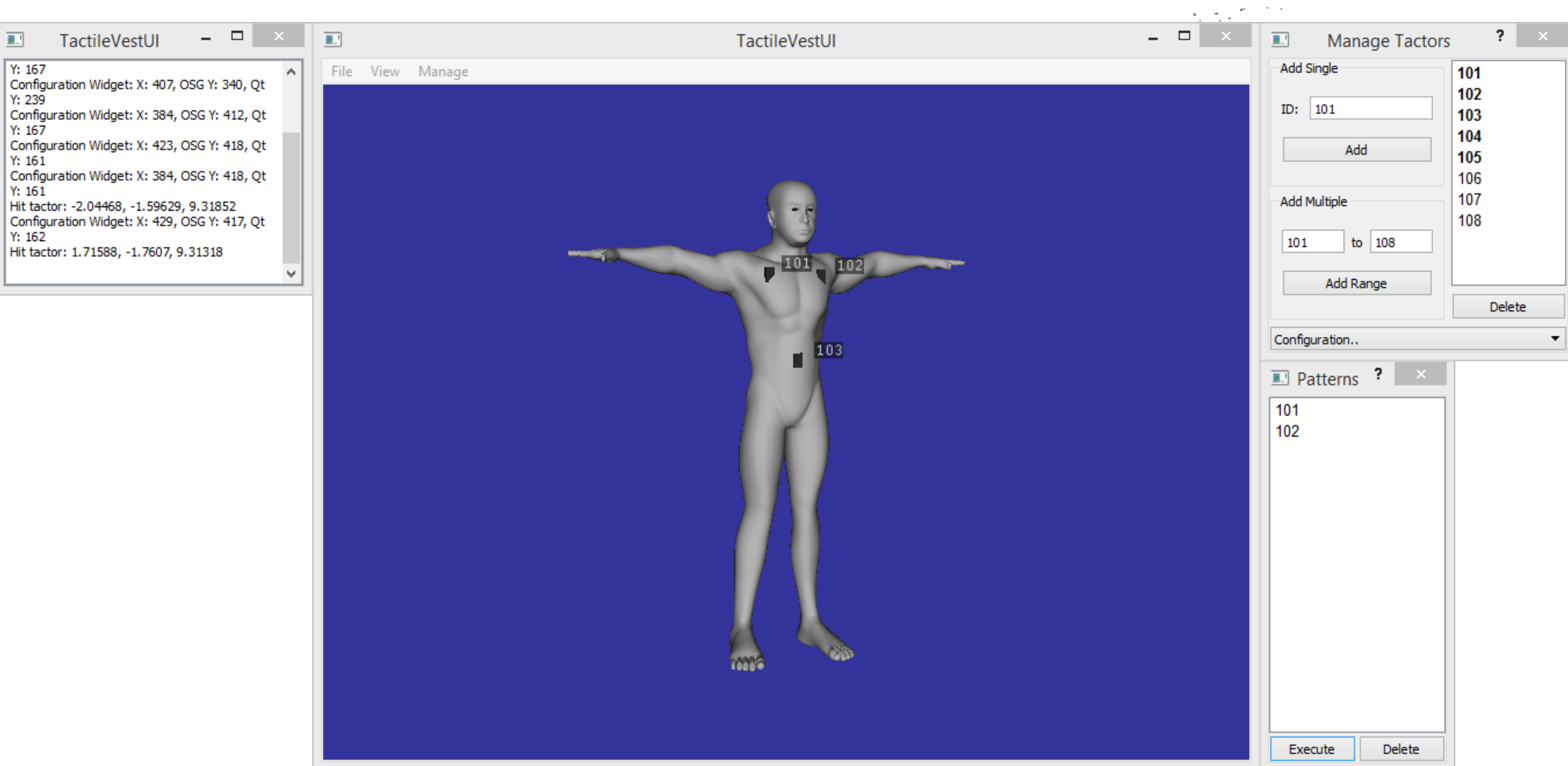
- Pattern:addPattern(int:time, SimplePattern:pattern) - Add a pattern to be run at a certain delay
    - The delay is measured from the start of the ComplexPattern execution, not the previous pattern execution
    - Return the Pattern previously set to run at this time, or void if none existed
- Pattern:removePattern(int:time,SimplePattern:pattern) - 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

3.2 User Interface Description



The user interface will be developed by using the cross platform Qt framework. The main screen a user will see is a human model along with the Log, patterns, and manage tactos window. Users will be allowed to add however many tactors they need and also to assign an ID to the tactors within the manage tactors. Once the tactors are created the user has the option of right clicking anywhere on the model and selecting what tactor to go at that location. Once this is done a tactor object is added to the model along with a label indicating its ID. Our first iteration (Version 1) will have pre-defined location on the body to place tactors, but in the future (Version 2) we hope to be able

to allow the user to place the tactors anywhere on the body. We also will have the ability to allow the user to load on the fly configurations that they have previously saved.

Model View

- Main display of a human model using Open Scene Graph.
- Model is able to rotate by use of the mouse.
- Tactors can be assigned to the body by right clicking

Manage Tactors Window

- Contains The current tactors for the session.
- Tactors can be added with a given id and also can be added in a range
- When a tactor is added it will be shown within the list and when a tactor is attached to the model it will be highlighted as "in use"
- Combo box with the ability to choose from saved configurations within current directory

Patterns Window

- Tactors can be added to the pattern by right clicking on a tactor on the model and clicking "add to pattern".
- Execute will send signals to the tactors in the list sequentially.

Log Window

- Every Time an action is made it is logged within this view.
- Once window is closed or if user wants to this log will be written to file.

Save/Open configuration

- The user has the option of saving the current configuration into an xml file.
- The user has the option to read in an xml file with the proper format into the current session to populate the Manage Tactors Window and also the Model View.

3.3 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

Implementation

4.1 Basic Building Blocks

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

4.2 Familiarity with platforms/tools

4.2.1 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.

4.2.2 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
- **Open Scene Graph** - This will allow us to show the user a 3D representation of a model.

4.2.3 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.

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.
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
Open Scene Graph	Open source library for rendering 3D models with Qt support