3/6/2016

Expanse

System Design

Group 13
Allen Tung
Andrew Chang
Anthony Wong
Dmitriy Kozorezov
Joshua Chan
Marc Tabago
Yue Yang

14:332:452
https://github.com/joshpaulchan/expans

Individual Contributions Breakdown

Allen Tung	General documentation, wrote pseudocode and test cases, organized project and merged reports sections. Helped create diagrams, and captions. Updated report 1 and beginning work on window capture software writing.
Andrew Chan	Contributed to documentation. Continued research on gesture tracking using Kinect as well as coding for gesture tracking.
Anthony Wong	Documentation; control input via a black box input from a simple server.
Dmitriy Kozorezov	Contributed to Documentation. Worked on WindowObj controls testing, Applying Three.JS to virtualizing workspace.
Joshua Chan	Wrote pseudo-code for micro-algorithms in Algorithms section; currently implementing the server and setting up the inter-process communication infrastructure so that the gesture and window feeds team can communicate with the server
Marc Tabago	Contributed to documentation; began preliminary development of Kinect-based gesture tracking while continuing looking into hardware/software capabilities
Yue Yang	Contributed to part III, IV and VI. Starting to work on coding.

All team members contributed equally.

		Allen	Andrew	Anthony	Dmitriy	Joshua	Marc	Yue	Total
RESPONSIBILITY LEVELS	Project Management	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 1: Interaction Diagrams	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 2: Class Diagram and Interface Specification	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 3: System Architecture and System Design	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 4: Algorithms and Data Structures	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 5 : User Interface Design and Implementation	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 6: Design of Tests	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%
	Sec. 7: Project Management and Plan of Work	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%	100%

Table of Contents

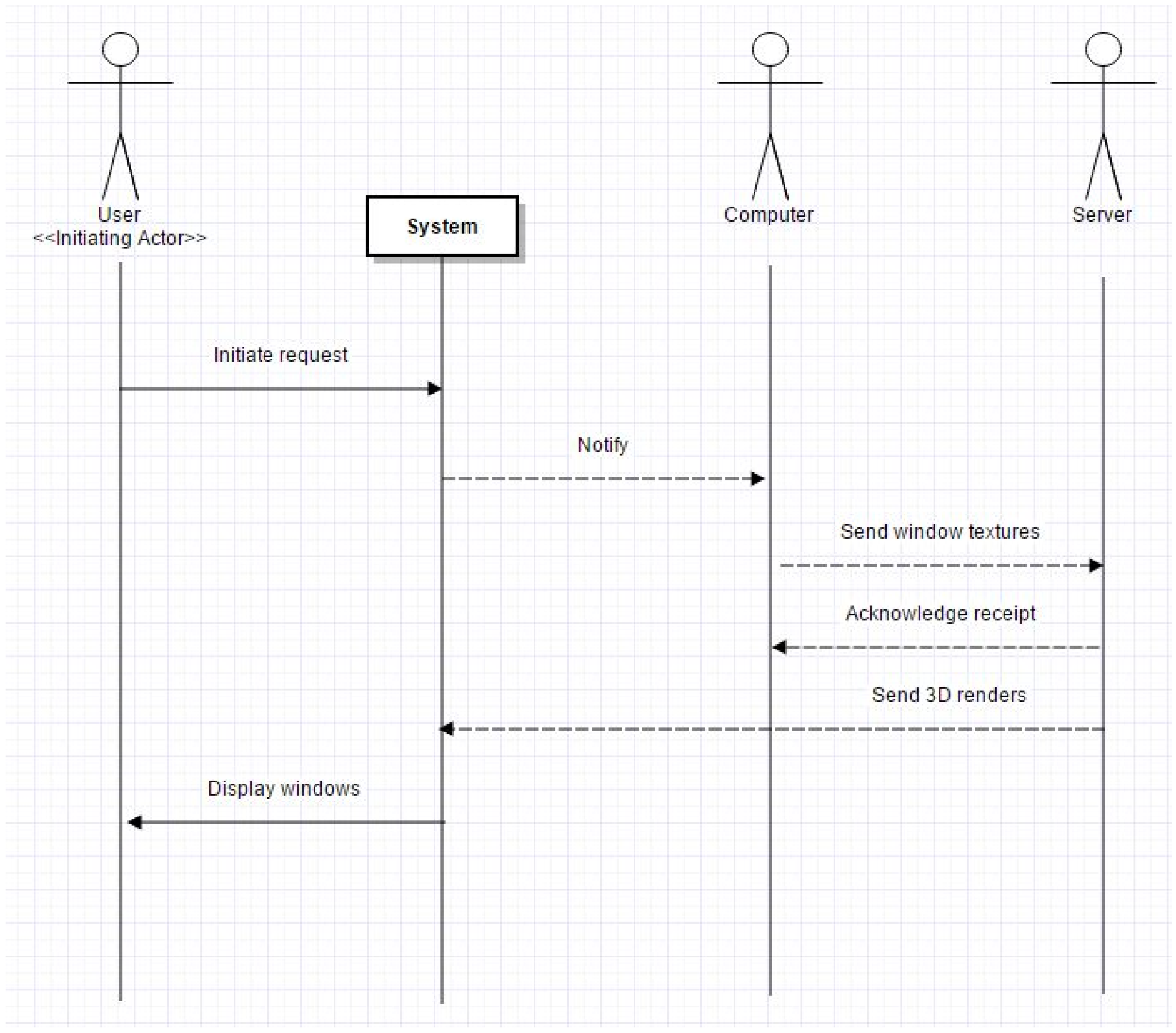
I. Interaction Diagrams

We chose to create two separate programs: One, to capture the window textures from the computer, and two, a server, that will send the window images (and other data) to the client HMD.

Because we have two programs to do the capturing and sending, respectively, our design follows the Expert Doer principle, somewhat akin to the Unix philosophy, in which one component is responsible for, and does, one major thing. Our design also follows the High Cohesion principle, in which the computational effort each program requires is distributed fairly equally.

Having multiple controllers means that the Low Coupling principle is followed; rather than having one central controller, some complexity in the communication network is added to increase the number of components in charge of communication, and thereby spread the load of each component's communication responsibilities across a larger number of components. The alternative would be to merge all the controllers into one central dedicated controller, which simplifies the diagram, but forces the controller to handle a very large amount of communication.

A. UC-1

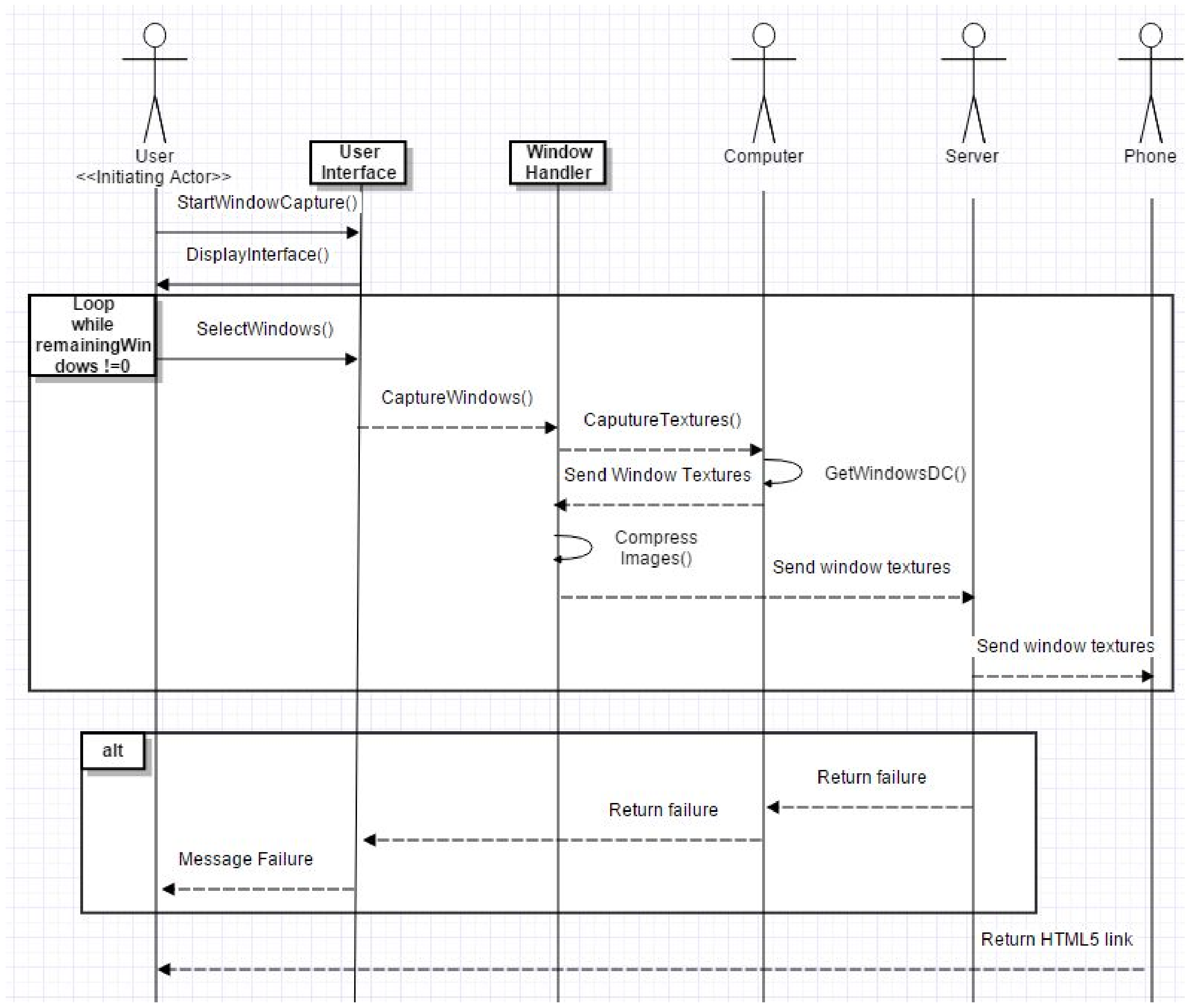


System Sequence Diagram for UC-1

The user starts virtualization by starting the overall program on the host computer and navigating to the app on her phone. The program on the computer starts capturing window screens and sending them to the phone, which renders the images in the 3d environment and displays them to the user.

There is a short period of time between the system receiving the initial request and displaying the window, but this should be in the order of a few microseconds. As a fundamental block of our program, this needs to execute flawlessly.

B. UC-1 & UC-3



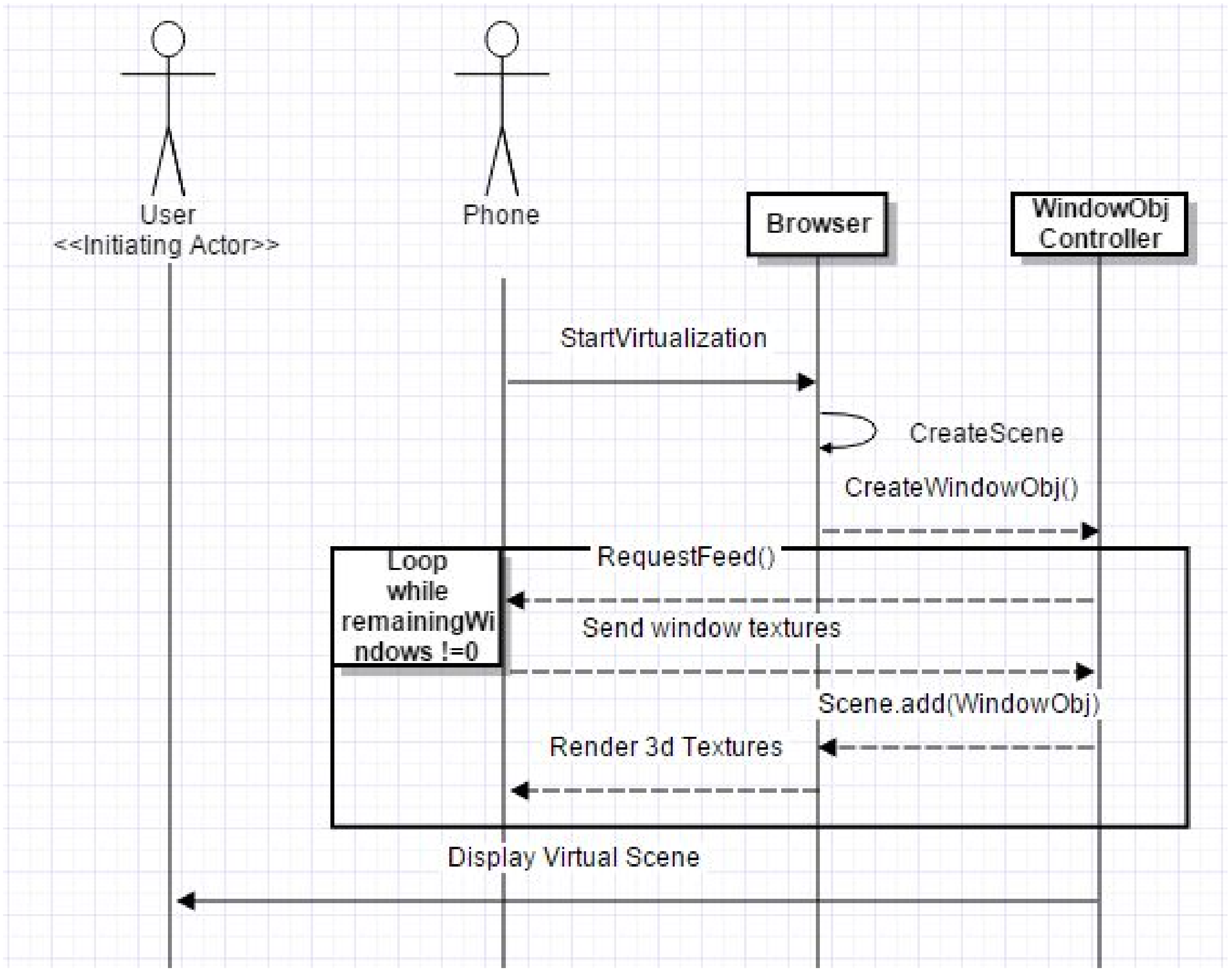
Detailed Sequence diagram part I

It begins with the User requesting to display windows from the desktop, which triggers a sequence of events ultimately leading to the windows hopefully displaying on the phone screen in 3D. At the end of this sequence diagram, the server will have sent all the window textures to the phone, which would process the textures and return a link to the user, which would automatically launch to display the 3D window renders.

As per the GUI that we specified in our RAD for Window Selection, the user has the option to change which windows are captured of all the applications running on the computer. The user would open the settings and navigate to the window that lists all the open programs.

Then, it would check on/off which windows will be grabbed. That command stops the window capture program from capturing those windows via an exception-list or blacklist, and the server will only send off whichever window images are received.

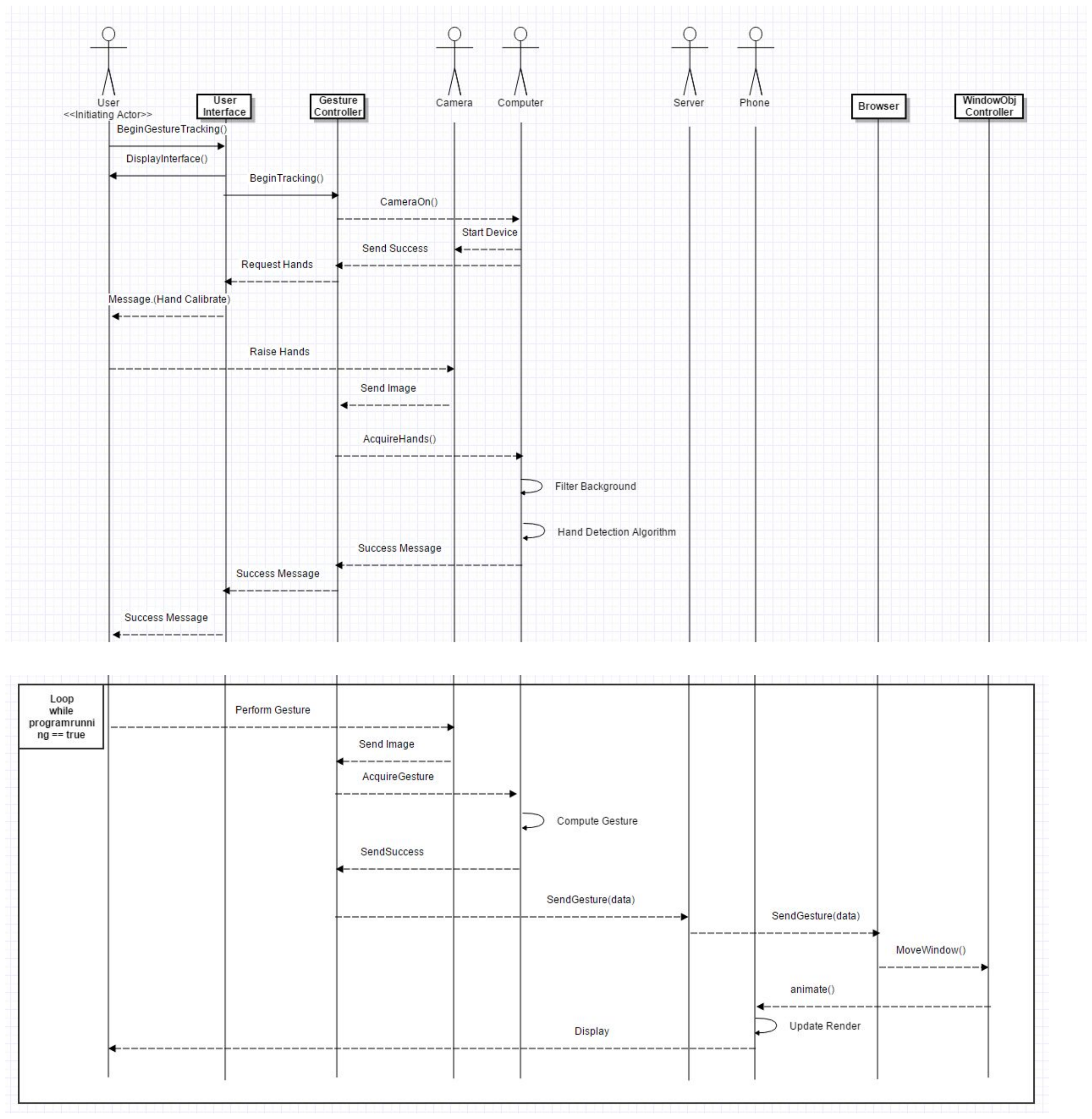
Analyzing this workflow, it’s clear that adding this ‘exception/selection’ functionality does not change the server, as it doesn’t need to know explicitly what images to ignore, and will only send what the capture program gives it. This does add extra computational burden to the capture program, but not much - just a block that stops the capture program from capturing all windows. In fact, one can argue that this might even relieve the capture program’s computational effort, since it won’t have to capture all windows anymore.



Detailed Sequence Diagram part II

The phone, having received the textures as detailed in the previous sequence diagram, will need to render the images in the browser, using webGL. The process by which this is done is detailed here. At the end of this sub case, the window object controller would display the virtual scene to the user, and the phone would simultaneously return a link to the corresponding renders.

C. UC-2



System Sequence Diagram for UC-2

As notable from the diagram, within the scope of the system, displaying changes to the workspace as a result of the user's actions can potentially take a long period of time. However, by maintaining both the High Cohesion and Low Coupling principles, the system should not be hung up at all during the gesture tracking process.

As an example, the tracking of the user's hand for gesture recognition and positioning purposes could have been within one module, but this would create an overreliance on one part of the implementation and would be likely to slow down the response of the system as a whole. In this case, the plan going forward is to create two modules of code, one for each of the duties outlined in the previous example. The diagram outside of the scope of the loop performs the hand tracking, while the diagram within the loop is set to perform any recognizable gestures and update all interfaces accordingly; having these similar ideas run individually of each helps to enforce the cohesion and coupling principles touched on previously.

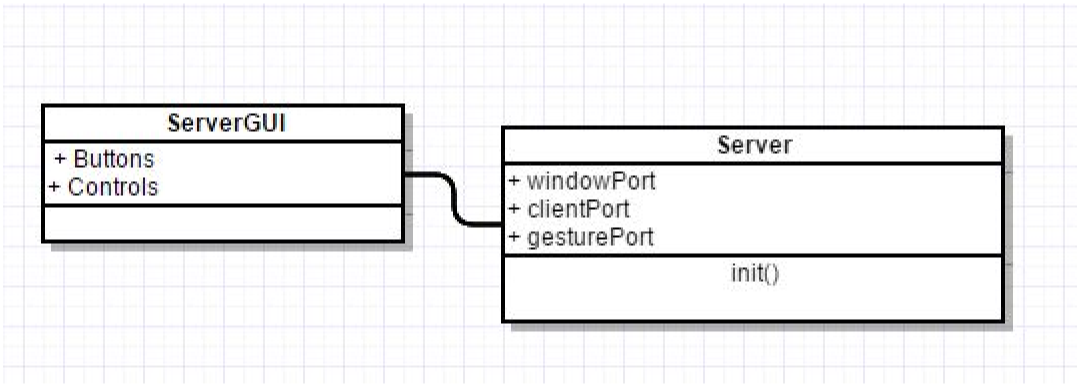
Responsibility weights: None / Light / Medium / Heavy

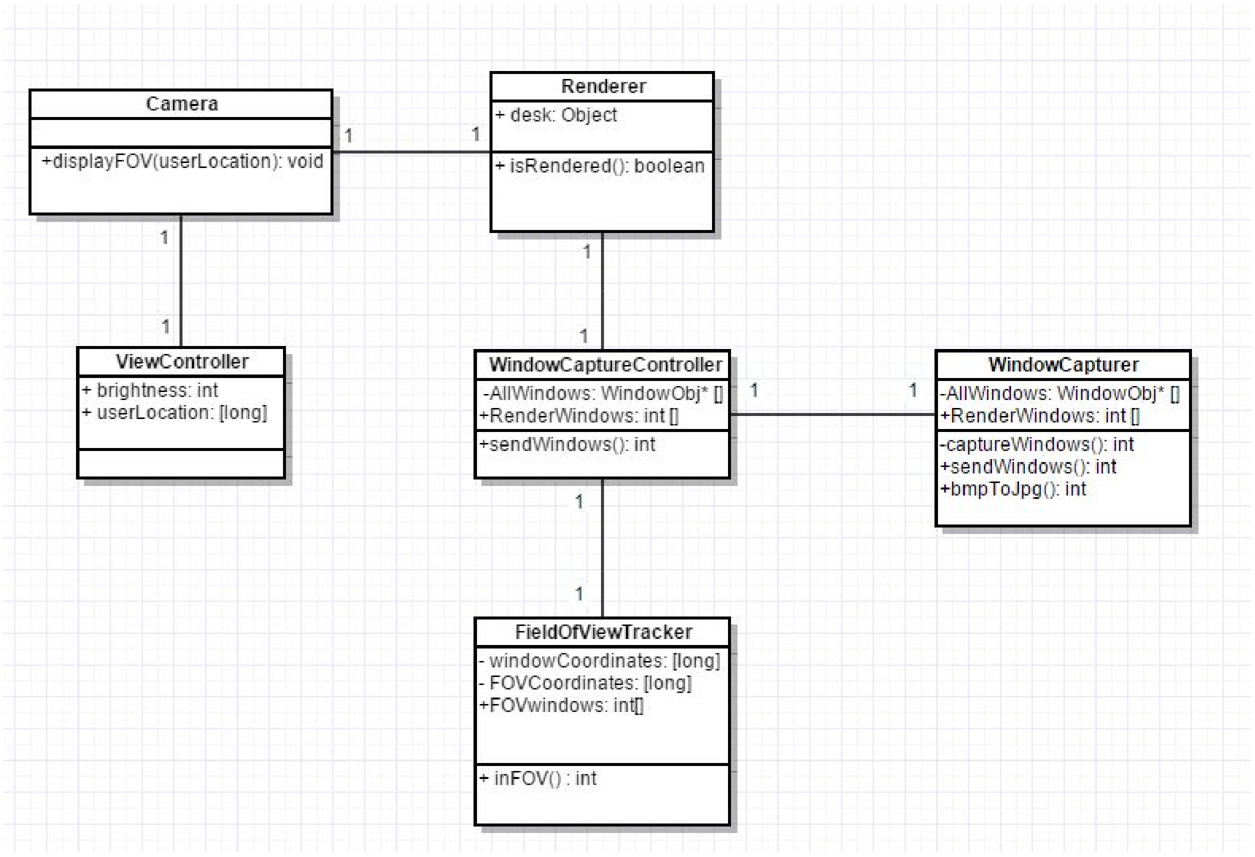
Responsibility Actor	Type 1 Knowing	Type 2 Doing	Type 3 Communicating
WindowObjController	Medium	Medium	Medium
ViewController	Light	Light	Medium
Camera	Light	Medium	Medium
WindowObj	Medium	Light	None
InputController	Medium	None	Heavy
Renderer	Light	Heavy	Medium
WindowCapture Controller	Heavy	Light	Medium
Screen	Light	Light	Heavy
Field of View Tracker	Medium	Light	Light
GestureController	Light	Medium	Medium
ExtractHand	Medium	Heavy	Light
GestureTracker	Medium	Light	Light
GesturePerform	Light	Medium	Heavy
GestureLibrary	Heavy	None	Light
HandTracker	None	Heavy	Medium

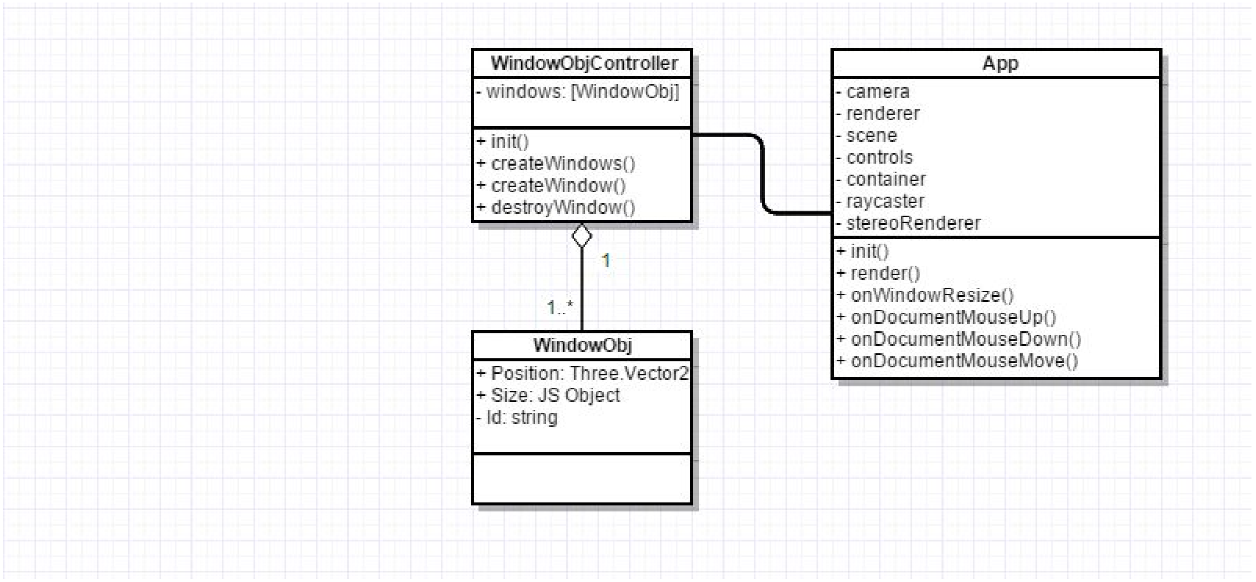
As we can see, the system was designed such that no actor would have two heavy responsibilities, and as much care was taken as possible to ensure that any one actor would not have too heavy of a responsibility. The key actors are those with both heavy and medium responsibilities, like the HandTracker or GesturePerform, and there was very little that could have been done to reduce their weight. In general, work was divided among ‘expert’ components of the system, where each expert component would have a single task to accomplish well. This increases the amount of communication required somewhat, but reduces the burden of each of the first two types of responsibilities on each actor.

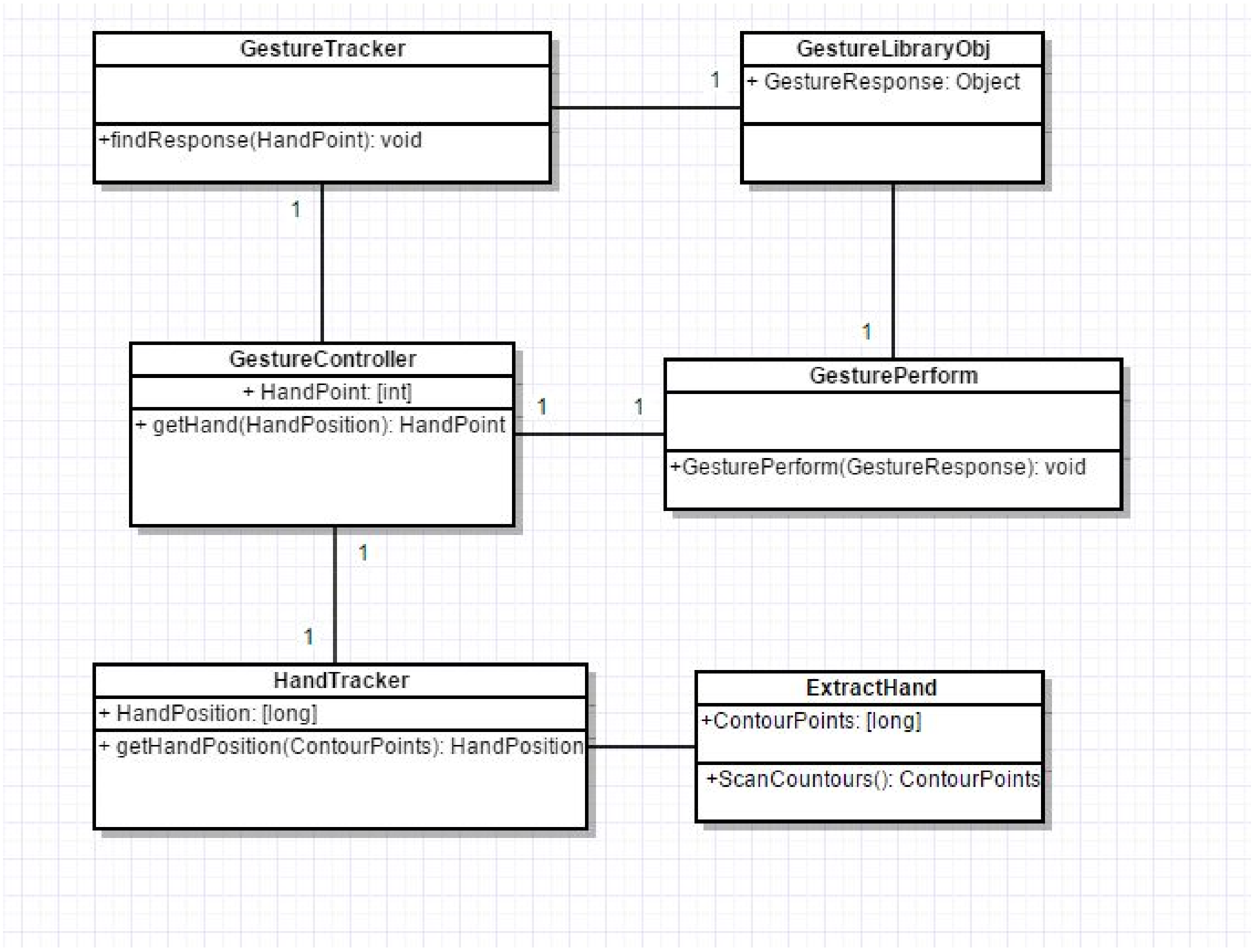
II. Class Diagrams and Interface Specification

A. Class Diagram






WindowObjController
- windows: [WindowObj]
+ init()
+ createWindows()
+ createWindow()
+ destroyWindow()
1
1..*
WindowObj
+ Position: Three.Vector2
+ Size: JS Object
- Id: string
App
- camera
- renderer
- scene
- controls
- container
- raycaster
- stereoRenderer
+ init()
+ render()
+ onWindowResize()
+ onDocumentMouseUp()
+ onDocumentMouseDown()
+ onDocumentMouseMove()


GestureTracker
+findResponse(HandPoint): void
1
GestureLibraryObj
+ GestureResponse: Object
1
1
GestureController
+ HandPoint: [int]
+ getHand(HandPosition): HandPoint
1
1
GesturePerform
+GesturePerform(GestureResponse): void
1
1
HandTracker
+ HandPosition: [long]
+ getHandPosition(ContourPoints): HandPosition
ExtractHand
+ContourPoints: [long]
+ScanCountours(): ContourPoints

B. Data Types and Operation Signatures

WindowObj:

The WindowObj class is the object responsible for holding data pertaining to each unique window feed and is designed in such a way that each WindowObj is independent of another.

Attributes

+size: JS Object	This is a hashmap with keys ‘width’ and ‘height’ that represent the width and height of the WindowObj
+position: Three.Vector3	The x,y,z coordinates of the object; the center of the scene is the origin
+obj: Three.Group	This is an a class defined by Three.JS that groups multiple 3D objets together. A 3D object is a geometry and a texture/material, where a geometry is a set of points that make up an object and a texture/material is how the space corresponding to a geometry should be styled. A group allows us to have multiple geometry & material pairs - one for the window images, one for a close button and one for a resize handle.
+id: int	The unique identifier of each object; used to reference the object

:
Methods

+getId():	Returns id
+getPosition():	Returns x,y,z position
+close():	Instantiates the window and does garbage collection
+setSize(x,y)	Given x,y input resizes the windowobj
+setPosition(x,y,z)	Sets the x, y, z of the windowobj
+update()	Updates the internal state of the object so it can be re-rendered

WindowObjController:

This class is responsible for tracking all WindowObj initiated and visible on the screen. It controls the lifecycle of WindowObjects.

Attributes

+windows: [WindowObj]	Array of WindowObj Objects

:
Methods

+createWindows([urls])	open a window once the user uses an app
+destroyWindow(id)	close a window once the user quits
+moveWindow(id, Vector2)	move the window when the user drags it
+focusWindow(id)	a specified window goes to top when more than two windows overlap
+minimizeWindow(id)	Minimizes the window and stores it on the virtual desk at the bottom of the display
+resizeWindow(id, JS Object)	Changes the size of the window when dragged from the corner

App:

App serves are the wrapper that controls the lifecycle of the virtualization program.

Attributes

+camera	camera object defined by Three.JS
+scene	scene object defined by Three.JS
+controls	control object defined by Three.JS
+renderer	render object defined by Three.JS
+container	container object defined by Three.JS to hold other variables.
+raycaster	raycaster object defined by Three.JS
+stereoRenderer:	effect object defined by Three.JS

:

Methods

init()	This initializes internal variables for the app
render()	This renders the entire scene, taking into account the camera, the renderer/effect, and the objects in the scene
onWindowResize()	This allows sets the controls for resizing the window through dragging a designated area of the object.

onDocumentMouseMove(event)
onDocumentMouseDown(event)
onDocumentMouseUp(event)
	These functions allow for the scene to be manipulated, for the user to be able to pan the scene and move the objects.

animate()	This allows for the object and scene manipulation.

GestureController

This class is responsible for updating/managing all gesture objects as well as controlling hand acquisition.

Methods

+getHand(HandPosition)	Returns hand position as a singular entity from its constituent x, y, and z coordinates

GestureTracker

This class is responsible for observing current hand positioning and signals when hand performs relevant gesture

Methods

+findResponse(HandPoint)	Uses simplified HandPoint to find a recognized, matching gesture command

GestureLibraryObj

This class stores all performable gestures and acts as a reference to user

Attributes

+GestureResponse: Object	Represents the computer actions associated with the recognized gesture that will be reflected in the user's view

GesturePerform

This class is responsible for sending a user/computer action corresponding to a recognized gesture

Methods

+GesturePerform(GestureResponse)	Returns the user/computer action associated with the recognized gesture

HandTracker

This class is responsible for sending the 3D coordinates of the hand

Attributes

+HandPosition	Represents the x,y,z coordinates of the hand

Methods

+getHandPosition(ContourPoints)	Returns the position of the hand

ExtractHand

This class is responsible for sending the 3D coordinates of the hand

Attributes

+ContourPoints	Represents the coordinates of the contours of the hand

Methods

+ScanContours()	Returns the contour points of the hand

WindowCaptureController:

This class is responsible for managing the capture of windows, sending the right captured windows to the right places.

Attributes

+CapturedWindows: int	Integer variable recording the number of recently updated windows ready to be rendered
-AllWindows: WindowObj* []	Array of pointers to captured WindowObj structs
+RenderWindows: int []	Array of AllWindow indexes, to which the AllWindow array will be used with to detail which windows are to be rendered

:
Methods

+sendWindows(): int	Send an array containing WindowObj structs to the renderer.

WindowCapturer:

This class is responsible for capturing windows, then sending the memory address of the captured windows to the controller.

Attributes

-AllWindows: WindowObj* []	Array of pointers to captured WindowObj structs
+RenderWindows: int []	Array of AllWindow indexes, to which the AllWindow array will be used with to detail which windows are to be rendered

:
Methods

-captureWindows(): int	Given the RenderWindows data, update the corresponding windows detailed in AllWindows
+sendWindows(): int	Send the pointer to an array containing the updated WindowObj structs.

+bmpToJpg(): int	Converts a given bmp to jpg. Used when updating AllWindows.

FOVTracker:

This class is responsible for tracking the window coordinates and size as well as the user's current field of vision, then determining what is and isn't in the user's current FOV.

Attributes

-windowCoordinates: CoordinateStruct[]	An array containing the coordinate structs of each respective window.
-FOVCoordinates: CoordinateStruct	A simple struct containing the coordinate x and y pairs of the current FOV, and the size of the FOV
+FOVwindows: WindowObj[]	An array of WindowObj structs, which are in the current field of view

:
Methods

+inFOV(): int	Regularly updates the tracked indexes of windows currrently in the FOV, whenever motion is detected

CoordinateStruct:

This class is responsible for storing the coordinate and size of each window.

Attributes

+Location_X: Long	X coordinate of window
+Location_Y: Long	Y coordinate of window
+Size_X: Long	X Size of window
+Size_Y: Long	Y Size of window
+windowNumber: Long	Index of the window this struct is referencing

C. Traceability Matrix

	Software Classes						
Domain Concepts	App	WindowObj	Window Obj Controller	Gesture Controller	Extract Hand	Gesture Tracker	FOV Tracker
WindowObj Controller			x				
ViewController							
Camera							
WindowObj		x					
InputController							
Renderer	x						
WindowCapture Controller	x						
Screen							
Field of View Tracker							x
BMPtoJPG							
GestureController				x			
ExtractHand					x		
GestureTracker						x	
GesturePerform							

GesturePerform							
HandTracker							

	Software Classes					
Domain Concepts	Window Capturer	Window CapturerController	Coordinate Struct	Hand Tracker	GestureLibraryObj	Gesture Perform
WindowObj Controller						
ViewController						
Camera						
WindowObj						
InputController						
Renderer						
WindowCapture Controller		x				
Screen						
Field of View Tracker						
BMPtoJPG	x					
WindowsAPI	x					
GestureController					x	
ExtractHan						

d						
GestureTracker						
GesturePerform					x	x
HandTracker				x		

As we can see from the figure above, several concepts were mapped directly to their respective classes. The ones that could not be mapped to directly will be explained below:

App maps the concepts related to overall system of virtualization and is responsible for concepts such as renderer and field of view tracker.

CoordinateStruct serves to organize the data sent between the FOVTracker and the WindowCaptureController, packaging the location and size data together in one struct

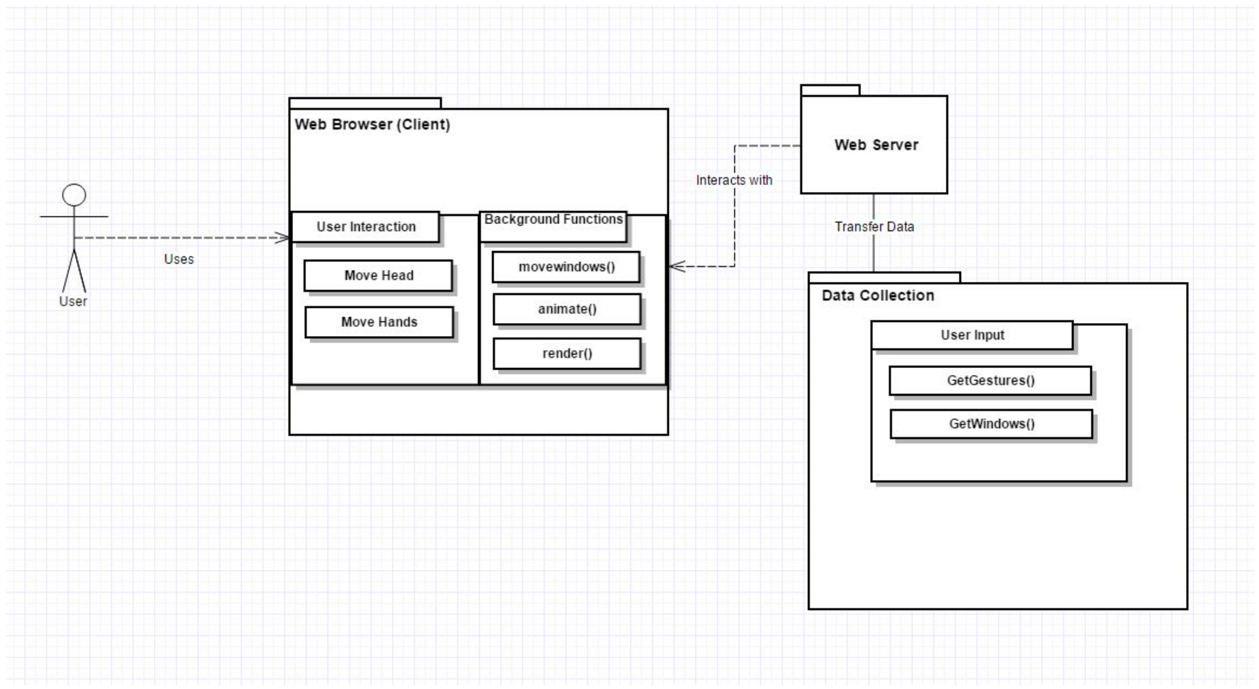
WindowCapturer evolved from WindowAPI and BMPtoJPG, turning the tools into a single class that would use said methods to capture windows and package them appropriately.

III. System Architecture and System Design

A. Architectural Style

The system uses a combination of mainly two architectural styles, the Model-View-Controller architecture and the Layered architecture. Starting from the user, the user's inputs are picked up by the gesture tracking hardware, which then communicates with several controllers that then interface with separate models for translating the user's gestures into actions which are reflected in the view, for the user, while in turn updating other controllers. The Layered implementation comes through with the user and the hardware being in the first layer, which communicates with the models and controllers in the second layer, which in turn communicates with a server that serves as the third layer, bridging the models and controllers to each other, as well as sending updates all the way up to the first layer for the user.

B. Identifying Subsystems



C. Mapping Subsystem to Hardware

The system uses a client-server architecture structure; due to the simplicity of the the architecture the hardware hierarchy is simple. The server is run off of the user's computer which transfers all relevant data related to gesture control and window capture application to the client. The data collection is done by a Kinect which captures the user's hands; the window capture is done via software on the user's computer. The virtualization subsystem takes place on the client which is the user's smartphone. By utilizing this distribution of work, we can provide the user the most optimal experience.

D. Persistent Data Storage

This system does not utilize persistent storage; each session is independent of any other session.

E. Network Protocol

The client-server architecture is supported by the Hyper Text Transport Protocol, commonly known as HTTP. The client loads the app from the web server using HTTP and standard network routing. We use HTTP/1.1, even though HTTP2 is much better at streaming data and handling concurrent data processing, because it’s still not widely adopted and the laymen’s documentation on HTTP2 is still very sparse.

The programs on the host computer (the server, window capture, and gesture capture programs) all computer using internal TCP sockets. Both the Window Capture program and the Gesture Capture program captures their respective data, then stream the data to the server using a socket.

F. Global Control Flow

Execution Orderness
Expanse is generally a procedure-driven system. The user is required to launch the Expanse program on their desktop, select their desired windows, and calibrate their hands before they can view and interact with their windows virtually.

Time Dependency
The system will make use of timers to keep a real-time image of the applications that are being visualizing. Timers will also be used to check response times between the server and client.

Concurrency
Expanse is not set to utilize multithreading at this time

G. Hardware Requirements

Computer Specification

	Min	Recommended
OS	Windows 7/8	Windows 7/8
RAM	2 GB	4GB
Graphical Processor	Intel HD 4400(DX 11 supported)	Nvidia GTX 660(DX 11 supported)
Processor	64 bit processor Dual-core 2.66Ghz	64 bit processor Dual-core 3.1 Ghz or better
Storage	Any	Any

Kinect

Color Camera	640 x 480 @ 30fps
Depth Camera	320 x 240 @ 30fps
Audio	16 bit audio @ 16kHz
Max Depth Distance	3.5 meters
Min Depth Distance	40 cm in near mode
Horizontal Field of View	57 degrees
Vertical Field of View	43 +/- 27 degrees
USB Standard	2.0
Supported OS	Win 7, 8

Smartphone Specification

	Min	Recommended
OS	Android KitKat 4.4 or higher	Android 5.0 Lollipop or higher
RAM	1 GB	3 GB
Graphical Processor	Adreno 306	Adreno 330
Processor	1.4GHz Qualcomm Snapdragon 410	1.8GHz Qualcomm Snapdragon 808
Storage	1 GB or higher	1 GB or higher
Display resolution	1280 by 720 pixels	1920 by 1080 pixels

IV. Algorithms and Data Structures

A. Algorithms

Virtualization

Although Expanse may seem like it requires intensive computation to adequately render and display the VR scene, a lot of the major graphical work is done for us by Three.JS. Three.JS has a well-defined application lifecycle that we take full advantage of, so we do not have to worry too much about creating a complex 3D rendering engine. However, we have decided to implement a few micro algorithms to improve usability of the application, pertaining to *Window Orientation* and *Layout/Distribution*.

Window Orientation

When a user repositions the window, its orientation (relative to the user) should change as well. It should stay perpendicular to the user's hand during the action, so that when a user pushes it off to the side, it won't just move linearly but instead emulate a circular path around the user. This function follows the mathematical model defined in the first report.

In pseudo-code:

```
def move(windowId, x, y, z):
    w = windowObjController.getWindow(windowId)
    w.setPosition(x, y, z)
    w.setOrientation(x, y, z)

// defined in WindowObj
def setOrientation(x, y, z):
    posVec = Vector(x, y, z)
    oldPosVes = Vector(self.x, self.y, self.z)
    theta = acos(posVec.xVec.dot(oldPosVec.xVec)/ ( posVec.length *
oldPosVec.length) )
    phi = acos(posVec.yVec.dot(oldPosVec.yVec)/ ( posVec.length *
oldPosVec.length) )
    w.setOrientation(theta + self.theta, phi + self.phi)
```

Window Creation & Distribution

When the application starts up, it is set to send the feeds of all the currently running applications on the host computer. This can clutter up the virtual workspace, so we decided to auto-layout the windows upon start-up and later creation. Windows will be created in a radial-manner, from the center.

In pseudo-code:

```
// defined in WindowObjController
def createWindows(windowFeeds):
    curLevel = 0
    For (i, feed in windowFeeds):
        If (i**2 > curLevel) then curLevel += 1
        w = createWindow(feed)
        R = Math.ceil(i / level^2)    // vector from center
        w.move(r.x, r.y, r.z)
```

FOV Tracking and Window Choice

Given the center coordinates and size of the current field of view, as well as the center coordinates and size of each window, we have all the information we need to decide whether or not to render a window. For the sake of efficiency, since this calculation must be repeated many times a second, we will use a simple scheme, and find the difference between the x coordinates of the centers of each window as compared to the center of the field of view. Of those that have a smaller difference than half of the sum of the field of view width and the window width, we will further filter by their y coordinate. We will filter out those windows which have a larger difference in their y values by over half the sum of the field of view height and the respective window height, and thus be left with only windows that are in the field of view.

In pseudo-code:

```
//Returns 1 if in fov, 0 if not
int[] inFOV(windowObj[] inq){ // inq for in question
     int[] result;
     int diff_x;
     int diff_y;
     result = malloc(sizeOf(int) * lengthOf(inq));
     setToOnes(result); //function to make every value in result 1
     //result will be an integer array with length equal to
     for(int i = 0; i < sizeOf(inq); i++){
         diff_x = FOV.center_x + inq[i].center_x;
         diff_y = FOV.center_y + inq[i].center_y;
         if( diff_x < ((FOV.x + int[i].x)/2)){
               result[i] = 0;
         }
         elseif( diff_y < ((FOV.y + int[i].y)/2)){
               result[i] = 0;
         }
     }
    return result;
}
```

Frame-rate Tracking

We will make two global variables to keep track of the number of updates we can accomplish in every second. The first will be the displayed framerate, and the second will be a number that counts up every second, and refreshes to 0 on the second, posting its previous value to the first global variable.

Gesture Tracking

As no gesture recognition algorithms exist natively within the Kinect for Windows SDK, it will fall upon the gesture tracking team to implement algorithms for every step of the gesture tracking process. To this end, we take some inspiration from the works of Daniel James Ryan of the University of Stavanger, who began an exploration into using the Kinect for purposes outside its intended use.

The gesture tracking process begins by determining the contour of any objects of importance (in this case, the hand in a forehand toward the sensor orientation); finding this contour starts with finding an initial contour pixel that stands out against the initial depth map mapped by the Kinect's camera. After locating this initial pixel, the contour tracking algorithm then searches for more contour pixels in a grid-like search pattern, stopping when it reaches one of two possible end conditions: either a contour pixel is discovered by the algorithm twice, or enough contour pixels are discovered to produce a usable contour. From this list of contour points, the positions of the fingertips can be determined by k-curvature detection. K-curvature detection uses the list of contour points gathered from contour tracking, a constant k, and a radian angle w (representing the average angle between fingertips) to find clusters of contour points that are individually less than w radians from each other in order to generate a rough outline of each individual fingertip. Once these fingertips are determined, it becomes a simple matter of running a Euclidean distance comparison between the stored gestures and the user input and then drawing the lowest distance path to determine the user's gesture input within the system.

B. Data Structures

Three.JS exposes some primitive data structures that the Virtualization engine has inherited from to create the client-side WindowObj and WindowObjController. The behind-the-scenes implementation of WindowObj is really just a group of geometries and materials covering those geometries, both of which are well-defined by Three.JS. Building upon those, we added internal methods to make these windows interactive, manipulable, and overall functional. These additions include movement functions, to reposition and reorient the window, initialization methods for creation, destruction methods for garbage-collection, and a few other useful methods.

The windows capture team shares a WindowObj structure with the virtualization team, and focuses on the CoordinateStruct associated with each WindowObj to manage the FOV tracking, and describe which windows to render that are currently in the user's field of view. The WindowObj structs are also used to store the textures for each window, prior to their rendering. For quick access and ease of management, we store the structures in an array, and use the indexes of that array to quickly manage which WindowObj structs we do and don't render.

The gesture tracking team uses the array as its primary data structure, storing lists of contour points from the input in an array, and comparing the resulting fingertip mappings to the gesture fingertip mappings stored as arrays within the system.

V. UI Design and Implementation

A. Review and Comparison to Initial Projections

Our initial GUI specifications have seemed to comply with all the needs that our separate teams seemed to require. We have decided to add a few more text overlays with debug information, such as frame-rate, but for the most part, the UI is just like what was designed.

B. Other

We aim for minimal user effort, accomplishing the key goals in a single step.

User effort:
Press the minimize button to minimize a window.
Press the icon for a window to bring it up in the current FOV.
Perform the gestures needed to elicit a system response.

VI. Design of Tests

A. Test Cases

Test Case Name: Window Render selection

Function Tested: refreshWindow()
Pass/Fail Criteria: Pass if only the correct windows are sent to the render server. Fail if any incorrect window is sent to the render server.

Test Procedure	**Expected Results**
Create windows completely to the left, right, above, and below the FOV.	No windows rendered
Create windows approximately around the border of the FOV- half in, half out.	All windows rendered
Create windows completely within the FOV	All windows rendered
Merge a few windows from each above case	Only windows from cases 2 and 3 rendered
Create window outside of the FOV. Then change FOV to view window.	No windows rendered, then 1 window rendered once FOV changed
Create window within FOV. Then adjust FOV to view nothing.	1 window rendered, then no windows rendered once FOV changed

Test Case Name: User Gesture Recognition
Function Tested: Ability of the system to track user hand, relate input to stored gesture
Pass/Fail Criteria: Pass: user's intended input is a gesture and is recognized as one
Fail: user's non-gesture input is recognized as a gesture, or user's gesture input is not recognized as a gesture

Test Procedure	**Expected Results**
Present nothing to the sensor (control)	No gesture recognized
Present non-hand object to the sensor	No gesture recognized
Present non-forehand view of the hand to the sensor, non-gesture position	No gesture recognized

Present non-forehand view of the hand to the sensor, gesture position	No gesture recognized
Present forehand view of the hand to the sensor, non-gesture position	No gesture recognized
Present forehand view of the hand to the sensor, gesture position	Gesture recognized
Present forehand view of the hand to the sensor, gesture position, outside of sensor's effective range	No gesture recognized

Test Case Name: WindowObj Creation
Function Tested: createWindow(feed,position)
Pass/Fail Criteria: Test will pass if a WindowObj is instantiated

Test Procedure	**Expected Results**
Call function (Pass)	WindowObj is instantiated
Call function (Fail)	Return error if invalid data is passed, feed already exists or position is invalid

Test Case Name: WindowObj Deletion
Function Tested: deleteWindow(windowObj)
Pass/Fail Criteria: Test will pass if a WindowObj is deleted

Test Procedure	**Expected Results**
Call function (Pass)	WindowObj is deleted
Call function (Fail)	Return error if invalid pointer is passed

Test Case Name: WindowObj Reposition
Function Tested: moveWindow(windowObj,position)
Pass/Fail Criteria: Test will pass if a WindowObj is moved to the correct location

Test Procedure	**Expected Results**

Call function (Pass)	WindowObj starts at a known position and if relocated to a correct position is successful
Call function (Fail)	Test fails Window does not move to the correct location or does not perform the desired effects.

Test Case Name: Virtualize Work Space/WindowObjController Functions
Function Tested: Testing if the workspace is compiled and virtualized.
Pass/Fail Criteria: Pass only if the workspace can be manipulated and virtualized. Fail if the workspace can not accept the user's input in workspace.

Test Procedure	**Expected Results**
User calls for a new window object to open.	**New window opens on desktop and virtually in workspace.**
User closes window in virtual space.	**Virtual window terminated and closed on desktop.**
User looks around the space.	**The workspace moves with respect to the user's camera.**
User drag virtual windows	**Virtual windows move only in the workspace, not on the desktop.**
User minimizes window in virtual space.	**Windows are minimized in workspace only.**
User tries to move camera behind window objects	**Workspace fails to look behind the windows.**
User tries to load workspace with no connection to server.	**Virtual workspace fails to render.**

B. Test Coverage

Expanse has critical distributed systems infrastructures that are essential to the functionality of the system. Our tests focus on measuring the performance of the virtualization engine on the client, testing about ⅓ of our whole system. We have also written a few tests for examining the performance of the IPC between window capture/gesture capture programs and the server, as well as tests measuring the efficiency and accuracy of the gesture capture program.

C. Plan for Integration Testing

Since we are using a distributed system, we will utilize bottom-up testing, locally testing each component on the same system to ensure we get proper input and output of each subsystem. The purpose is to isolate each component so that we can confirm that errors do not arise from each system. Next, the system will be tested across a distributed system, where the focus of our testing becomes reliable data transfer and latency. We test the responsiveness of the server and it's ability to properly transfer data from programs in C++, to another, in JavaScript.

D. Non-functional requirement Testing

One non functional requirement would be that we maintain a comfortable 30 frames per second refresh rate. We plan to track the refresh rate, and display it in an unobtrusive corner of the user interface.

VII. Project Management

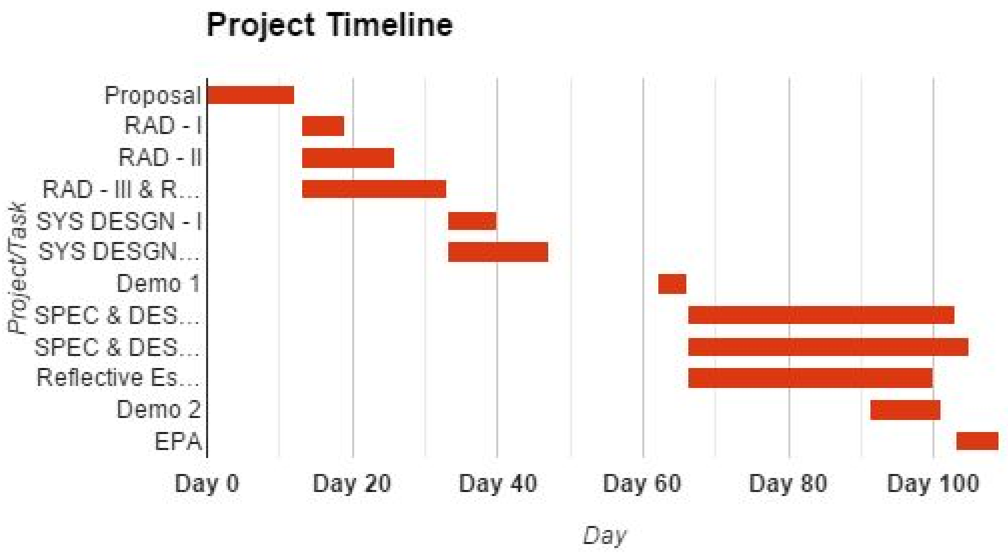
A. Merging the Contributions from Individual Members

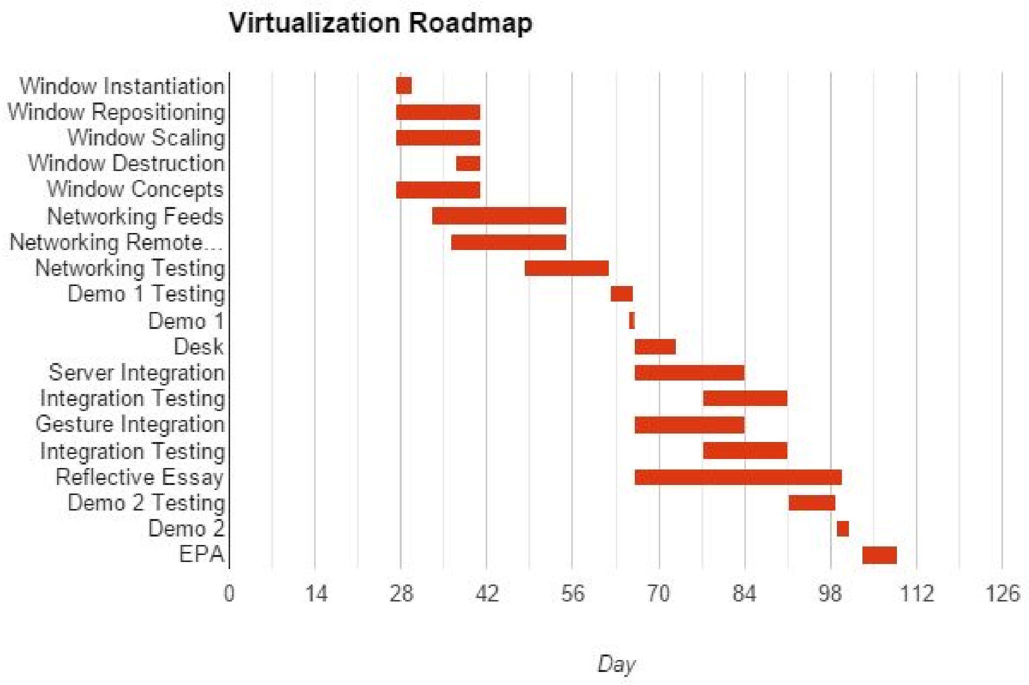
The report was compiled on Google Docs so that each individual member could contribute at their leisure, with real time synchronization. The previous two sections were merged with little difficulty, requiring only some renumbering of section titles, and a little formatting. One issue was that there was no point in time at which every member of the project could work on the report at once, but this problem was not too detrimental to the progress of the report, as the work each member contributed was instantly available at any point.

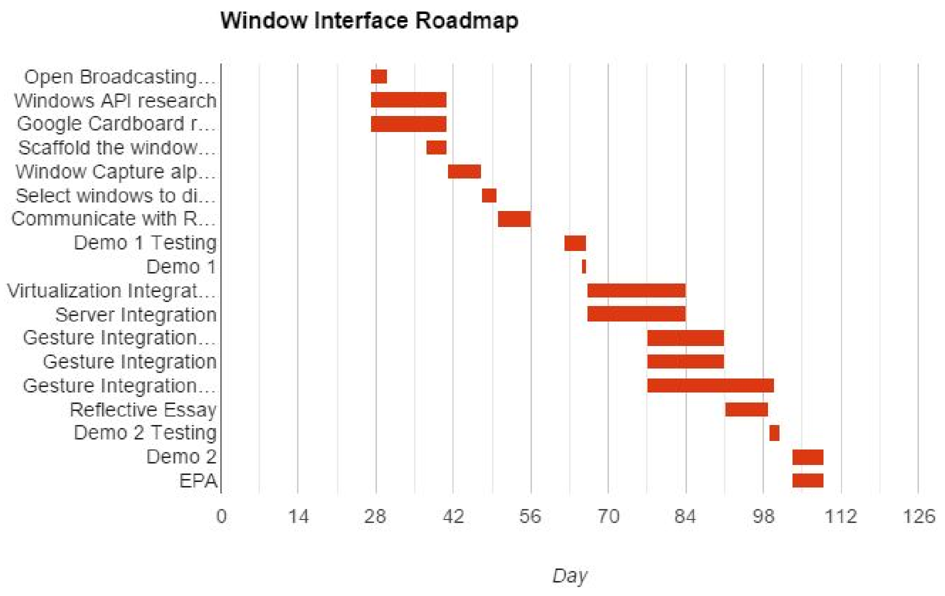
B. Project Coordination and Progress Report

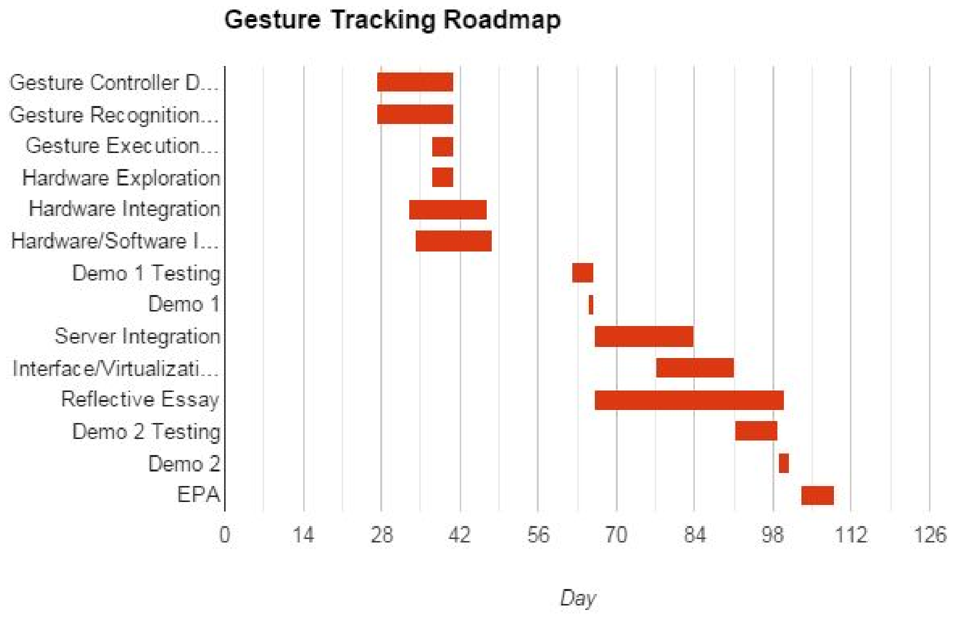
We're in the middle of writing the server/host programs, and plan to have functional (but non-integrated) programs by the 1st demo.

C. Plan of Work




Virtualization Roadmap
Window Instantiation
Window Repositioning
Window Scaling
Window Destruction
Window Concepts
Networking Feeds
Networking Remote…
Networking Testing
Demo 1 Testing
Demo 1
Desk
Server Integration
Integration Testing
Gesture Integration
Integration Testing
Reflective Essay
Demo 2 Testing
Demo 2
EPA
0
14
28
42
56
70
84
98
112
126
Day


Window Interface Roadmap
Open Broadcasting…
Windows API research
Google Cardboard r…
Scaffold the window…
Window Capture alp…
Select windows to di…
Communicate with R…
Demo 1 Testing
Demo 1
Virtualization Integrat…
Server Integration
Gesture Integration…
Gesture Integration
Gesture Integration…
Reflective Essay
Demo 2 Testing
Demo 2
EPA
0
14
28
42
56
70
84
98
112
126
Day



D. Breakdown of Responsibilities

VIII. References

http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf

https://msdn.microsoft.com/en-us/library/hh855359.aspx

https://dev.windows.com/en-us/kinect/hardware-setup

http://brage.bibsys.no/xmlui/bitstream/handle/11250/181783/Ryan,%20Daniel%20James.pdf