AVENTURA™
AWARENESS COMPUTING™

Aventura Gesture Engine Project

Johanna Smith, Matt Wesemann, and Charles Tandy

June 17, 2014

Table of Contents

Table of Contents.. 2

Introduction………. 3

Specifications and Requirements…………………………………………………………………………………... 3

System Architecture…… 4

Technical Design………. 5

Design Decisions and Integration……………………………………………………………………………………… 7

Lessons Learned…… 7

Results…… 8

Appendix A……… 10

1| Introduction

1.1 *Client Description*

Our client is Aventura Inc. – a company dedicated to increasing the efficiency and ease with which a clinician moves through a hospital environment. In a normal hospital, a nurse or doctor will approach a computer, sign in to use it, and then for the rest of the day that machine is essentially mandated to them. This becomes frustrating for the user when they must walk away from a patient room all the way back to their machine to input notes and record data. With Aventura's system, a clinician may log onto any machine using their proximity card and it will pull up their personal virtual desktop on that machine. The fantastic part is that it allows users to become more mobile- they can log out ("tap out" with their proximity card) of that physical machine but pull up all the same data immediately by "tapping into" another machine that will mold their same virtual environment. This system increases productivity in the hospital since no user is forced to return to the same physical place every time in order to do their work. It also removes the risk of logging out of a session because that exact same desktop with all the same information pulled up will still be there when you tap back into any other machine.

1.2 *Product Vision*

As mobile devices such as tablets become more popular and more frequently used in a work environment, Aventura saw an opportunity to allow their users even more physical freedom with their work station. Though a mobile device such as a tablet would not have a proximity card reader normally attached to it, it can still display the same virtual desktop that's been created elsewhere. With its portability, any clinician with such a device can still record any data from the last patient and pull up the next patient's information without ever having to stop at a stationary device. This is, again, a time saver and time is one of a clinician's most important assets. This is where our team came in. It's wonderful to have mobile access to patient information, but certain patient files (an x-ray for example) are actually meant to be shared with the patient. The mobile device loses its value when a doctor walks in with it, then is forced to log out and tap back into the larger monitor in the room for the patient's benefit. Our product, a gesture engine to handle any configurable touch input from the tablet and produce an associated action that is not native to the physical device, had the potential to resolve the stated issue. This gesture engine is really just an extendable version our client's request that our team develop functionality that would take in a three-finger swipe and cause screen sharing between the tablet and the other monitor in the patient room.

2| Specifications and Requirements

Our initial functional requirements were that we detect distinct gestures that are then sent to the Aventura server and return an action associated with that gesture. Our user story was that a mobile device could recognize a three-finger swipe and cause a screen sharing action with another monitor. Non-functional requirements were more heavily associated with security of a hospital system and their files. For example, our functionality had to be stringently tested to insure that it would not cause server-end problems (if the server goes down, the hospital cannot function) and also work to protect patient information. Adaptability was mentioned in order to support new gestures and in the future, possibly other platforms (such as an iPad instead of a window's Surface).

3| System Architecture

Our portion of the system as seen in Figure 1 is broken up into three main component- the gesture engine, the AV server, and the AV client (mobile and fixed devices).

The gesture engine, while running in the virtual machine, detects a gesture and sends it to the AV server. The gesture engine also functions as the server for a desktop screen share. This operates by having the AV server control the screen share server.

The AV server processes any given input and maps it to actions. It does this by delving into the database and mapping the input to a list of consequential actions. For example, when the gesture engine passes a gesture event, it takes that as an input. The AV server then sends a message to the AV client that contains the actions found in the database. Another thing that the AV server does is store user state for the desktop screen sharing.

The AV client then executes the actions that have been delivered from the AV server. The AV client also starts the screen sharing connector.

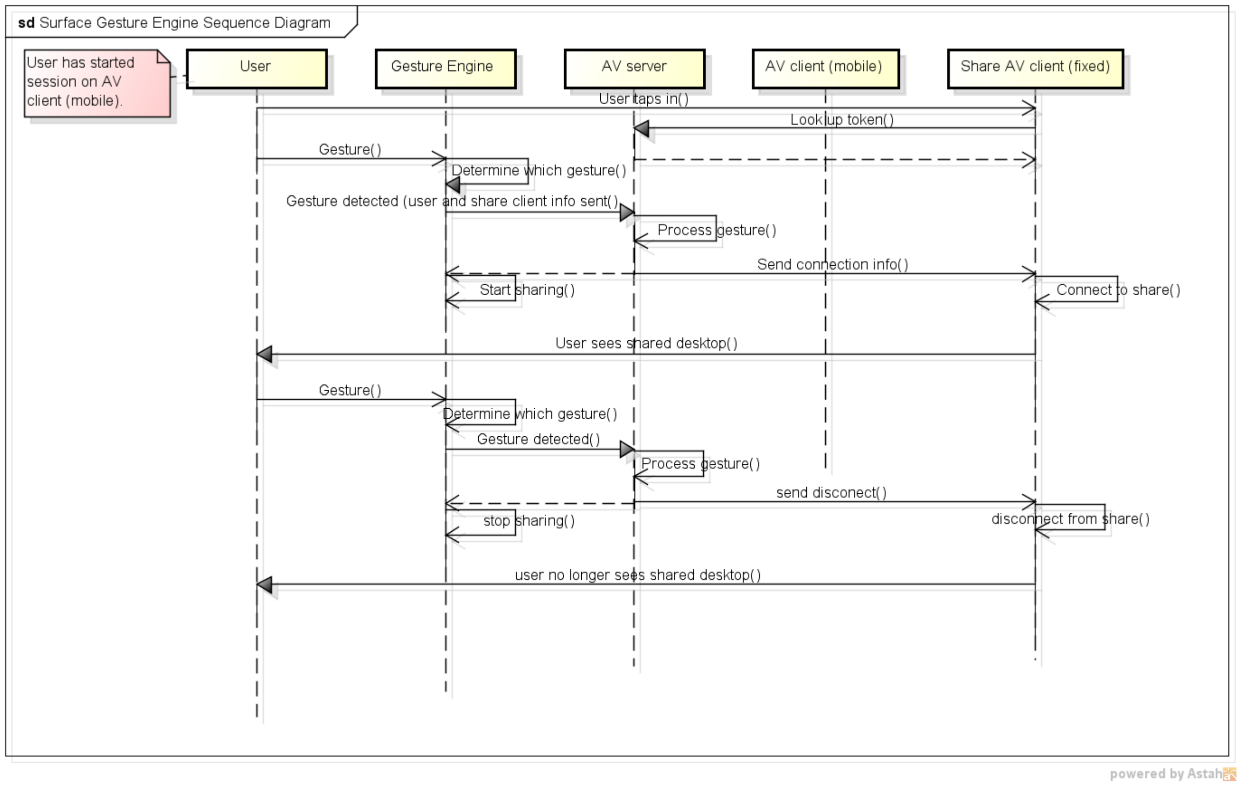


Figure 1: Gesture Engine Architecture through Sequence Diagram

4| Technical Design

4.1 *Input Detection*

We had to make many interesting design decisions around touch input processing. Our goal was to be able to listen to global touch input while also thinking about being easily extendible to future platforms. Unfortunately in Windows there are many ways of getting touch input but it is difficult to generalize it into a global solution. We started off by researching the different methods and as our research progressed we quickly started ruling things out.

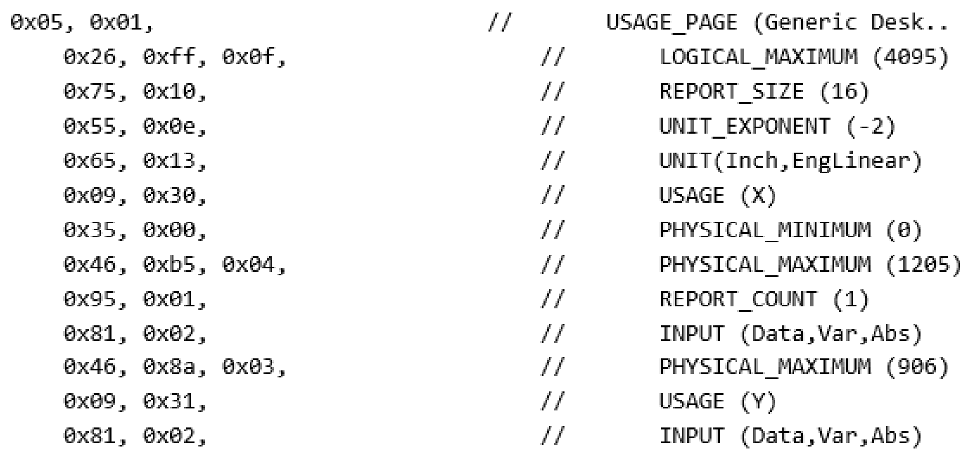
Generally, if you want to listen into something globally happening in Windows you register something called a Hook, which every time a certain event occurs your code will be called. There are two relevant events that could be used for touch: low-level input and application messages. Low-level input can be used to listen into all mouse and keyboard events as they occur before anything else can process them and your code will run in the context of your application. Surprisingly, because of legacy apps, touch events are converted into mouse events for those applications that do not handle touch events. However, it fails because it is difficult to tell if a mouse event was originally a touch event. There is a function you can call to see if it is a converted touch event but it must be called in a context that is unattainable here. Application message seem promising because as each window receives touch input we can intercept it and then process. This requires listening for the different touch messages that the windows can receive.

The first type of message is WM_TOUCH. This was added in Windows 7 and required apps to register themselves as being touch capable before getting these messages. WM_TOUCH is not global which means that we cannot use this message. When hooking into applications you can forward this message back to our application and then process it. This works well enough but few applications use it (and for good reason because it can be a pain to deal with) so it would need to be combined with other message types.

The next message type is WM_MOUSE. Legacy applications that ignore their touch message will get them converted into mouse events. In addition to the low-level mouse input you can listen to the mouse events in each application. Sadly, this code will run in the context of your app instead of the app receiving the input and therefore you are unable to tell if they were originally touch events. This alone ruins the chance of hoping to find a solution simply by listening into each application's message pump because this conversion happens with many apps people run today and we would miss all of that input.

The next message, WM_POINTER*, almost saves everything. These are a group of new messages added in Windows 8 that provide a rich, powerful, and easy to use input detection. The important ones for us are WM_POINTERDOWN, WM_POINTERUPDATE, and WM_POINTERUP. While not a global input, all apps will receive these messages. This means all apps will receive touch input we can listen for and easily process, but it still has problems. While all applications will receive the first WM_POINTERDOWN for a pressed finger, if they ignore this message they will receive legacy mouse events. And most bizarre of all, under some conditions an app may not receive any input messages at all! One such condition is Internet Explorer which uses DirectComposition which combines the input stack into the graphics stack for super low latency input response. With this last input method failing this completely knocks Hooks out of the running as a suitable input method.

There is another input method we looked into in the beginning but decided against it because of the higher difficulty and poor documentation. This method is Raw Input. With this method you get data directly from the driver. This means more work because instead of registering input types you register a specific device and then that driver sends you data which, of course, is in its own format. Luckily, this has all been standardized by an industry standard known as Human Interface Devices (HID). In Windows 8, touch screen drivers have to support this standard with a minimum number of supported options. In HID, data from drivers is given by descriptor reports with each driver specifying the layout of the report it gives. Then you can use Windows Driver functions that use this description to describe the report, listing what options are present and then extract the values for those options. After ruling everything else out, this was all that was left so we buckled down and finally got a working prototype. After getting it working, we optimized it and integrated this solution into our code. Figure 2 is a sample touch input report description. Here we can see information about how to get a specific data option of a report (USAGE (X)) and information about this option such as the byte size (REPORT_SIZE (16)) and the minimum (PHYSICAL_MINIMUM 0)) and maximum (PHYSICAL_MAXI MUM (1205)) values it can be.



```
0x05, 0x01,                    //      USAGE_PAGE (Generic Desk..
    0x26, 0xff, 0x0f,              //      LOGICAL_MAXIMUM (4095)
    0x75, 0x10,                    //      REPORT_SIZE (16)
    0x55, 0x0e,                    //      UNIT_EXPONENT (-2)
    0x65, 0x13,                    //      UNIT(Inch,EngLinear)
    0x09, 0x30,                    //      USAGE (X)
    0x35, 0x00,                    //      PHYSICAL_MINIMUM (0)
    0x46, 0xb5, 0x04,              //      PHYSICAL_MAXIMUM (1205)
    0x95, 0x01,                    //      REPORT_COUNT (1)
    0x81, 0x02,                    //      INPUT (Data,Var,Abs)
    0x46, 0x8a, 0x03,              //      PHYSICAL_MAXIMUM (906)
    0x09, 0x31,                    //      USAGE (Y)
    0x81, 0x02,                    //      INPUT (Data,Var,Abs)
```

Figure 2: Sample report description from a touch driver

5| Design Decisions and Integration

5.1 *Physical vs Virtual Machine*

Another major technical decision came from where we wanted to do all the input processing. Users of Aventura software do their work in virtual machines managed by Aventura. We could either detect the touch messages on each physical device or in the virtual machine using the information the Terminal Client has provided us. The three clients supported by our company are Remote Desktop (RDP), Citrix, and VMware. After testing what each connector does to touch input we had to rule everything but RDP because they mangled touch input into mouse messages. This prevents many users from using our new touch detection software if we run in the virtual machines. However, as Aventura adds support for new platforms they would have to re-implement touch detection code on every platform which is not ideal. One feature our client wanted was to be able to tell what app was focused when a gesture performed which would be more difficult if the touch detection was being done on the physical devices. We decided that overall things would be simpler running in the virtual machine as RDP working was a good enough solution and our client agreed.

5.2 *Screen Sharing*

There are two ways of displaying a session on a different device. We could either transfer the session on the mobile device to the secondary device or we could duplicate the screen on both devices. Transferring the session requires logging on and dealing with tokens while desktop duplication is much easier with new APIs added in Windows Vista. Since screen (duplication) sharing matches exactly with the scenario our client gave us and is much simpler to implement we are going with the desktop screen sharing route.

5.3 *Terminal Clients*

Of the three main terminal clients, our early testing discovered that both Citrix and VMware do not support passing through touch input while RDP does. Due to time constraints supporting other clients was not a high priority for this project as we focus on the Gesture Engine itself. Thus, as long as we had a popular working client we continued. Therefore, we chose to only currently support RDP. However as we switched to Raw Input (which comes in at device/driver level) it may be possible that the other clients are passing through the necessary data. Since we had already decided to only support RDP we have not tested Citrix or VMware.

6| Lessons Learned

6.1 *Research stories*

It is very easy to spend hours working on a piece of functionality only to find out that it won’t work with the project. Incorporating research stories into the project early on forces you to investigate the potential problems that may arise. This will prevent a tunnel vision effect where a developer is so focused on a certain action that they do not consider another path.

6.2 *Prototyping*

When working on implementing new functionality there can be unexpected behavior that can cause problems. Creating prototypes early and often helps to limit unexpected behavior when incorporating the new functionality into your code. Research stories and prototyping are closely tied. A good research story incorporates a prototype. The combination of these two will help to prevent unforeseen hiccups.

7| Results

7.1 *Performance*

When doing input processing you must be efficient because a simple swipe can easily generate hundreds of touch messages which can slow down a machine if processed incorrectly. We made sure that we were not doing any unnecessary work with our processing and the complexity of our algorithms were reasonable. One example is there were some situations in our Raw Input handler that we were looping over looking for the data we needed when we could instead just grab it directly.

There was also an inefficiency in the way we were storing a user's state in the database. Because of an existing code structure we were unable to load a user's state by their user ID but instead had to loop over all states looking for the correct user. We were able to fix this existing code which allowed us to switch to loading a user's state by their ID and removed all loops (linear to constant complexity!).

7.2 *Usability*

There are functions to add or remove Inputs and actions to the database. It will be fairly simple to implement a GUI using these functions so that a user can then modify their own input to action mapping.

There is a confirmation dialog that comes up when a user performs the gesture that maps to screen sharing. This dialog shows the user the location that they will be sharing with and allows them to choose whether they want to share the entire screen or just the application with focus. This makes it very simple for the end user to decide what they would like to share and verify the location that they would like to share to.

The gesture engine does not require the user's gestures to be perfect. For instance, a three finger swipe would occur if the user has three fingers down and one of the fingers moves past a given pixel threshold. This makes the gesture engine fairly user-friendly when taking in gestures.

7.3 *Future Work*

Our client is incredibly optimistic about the applications of this project *especially* in regards to sectors outside of the medical industry. The Chief Technical Officer of Aventura, Joe Jaudon, is most excited about its incorporation into conference room scenarios. As of right now, though people can duplicate their screens via VGA or HDMI cables, only one screen can be shared at once and input can really only be controlled from one end. With our screen sharing functionality, anyone who has “tapped” into the device being shared to as the most recent location, can share a specific window with an entire group, then anybody with access to that screen can edit the input. This allows for a much faster and more collaborative conference room experience. The same is true for the applications use in education- many people in a single classroom could be presenting a single window that displays their problem or they could be pointing out things they are confused about directly on the professor’s slides.

7.4 *Extendibility*

This project was created to accommodate a series of different inputs and can handle any number of consequential actions. The way we stored the input to action mapping in the database allows for a variety of inputs outside of just a touch gesture (for example a keyboard shortcut) and can accommodate a whole list of actions associated with a single input. In addition, the gesture engine can handle any number of points of contact (fingers on the touchscreen) and can be extended to any new gesture (such as a pinch).

Appendix A

Surface Gesture Engine Install document

Developer's Device:

The developer must have the most recent WDK on their device.

GestureEngine solution in GestureEngine Directory.

GestureEngineFrontend should be set as the startup project.

Build and run it.

User's Physical Device:

The device the user is swiping from needs to be registered as mobile on the AV server.

Virtual Desktop:

GestureEngine.dll, GestureEngineFrontend.exe and TouchHandler.dll should all be installed in EOF/Shared. This should all be added to the Apps and Agents Installer.

AV Server:

InputtoActionHandler:

Located in TISessionManager.

The Gesture mapping is written to the database from InputtoActionHandler.

Has all of the functions to add and remove mapping.

Message Handlers:

HandleGestureEngine takes in the touch events, calls the mapping to the actions, and sends the location information back to the HandleScreenSharingLocation message handler located in the GestureEngine project. This sends the location to the confirmation screen.

HandleScreenShareAccepted initiates the actual screen sharing session and is called from the GestureEngine project when the location is confirmed.

SoftClient:

ShareScreenConnector project must be initiated from the SoftClient.