Global Share System and Haptic Imprints

SK Semwal K Chandrasheker D Carroll A Deshmukh N Bastian
Department of Computer Science, University of Colorado, Colorado Springs
semwal@cs.uccs.edu

Abstract

We implemented the idea of haptic imprints for security applications Using the tools available at our disposal, our first priority was to provide access to information in a highly distributed environment. Global-Share project investigated the use of existing systems allowing access to information on the wearable visual displays available to us. SGI systems and a NOMAD display was available for our research. We used Vizserver software to display the output of an SGI system on the NoMAD display [5]. We then investigated ARToolKit® for our experiments. Haptic-Imprints use haptic devices, such as PHANToM forcefeedback device, to create a unique simple patterns which can be used to verify the haptic-signatures on-line. The need for *mobile* haptic devices with high fidelity is identified.

1. Introduction

Wearable and mobile computing poses special challenges in security research where humans in the loop system create special interaction issues. At the same time, wearable-environments allow us an opportunity to express, be creative, and quickly perform tasks which were otherwise impossible. At the minimum these systems are expected to understand verbal commands, provide feedback through a variety of output displays such as sound or displays. Wearable computing is the transformation of a user's personal space into an active area in which electronic assisted activities may be performed. This new area of research is rich with new ideas but it is far from being a mature field; yet acceptance of wearable devices in daily life has been slow at best, for example, watches [9] took several hundreds of years to be accepted as a device. In this paper, we describe our research efforts towards creating haptic applications geared toward physical security applications.

2. Global Share Project

In the Global-Share project [8] (Figure 1), our vision was to provide a technological collaboration between several individuals allowing them to interact with each other from anywhere in the world, and share important information using latest in wearable-computing and visualization research [7,10]. The long-term goals of the Global-Share project were:

(a) Information-Fusion: Combine the information from a variety of heterogeneous sources: collecting voice, text, images, video, 3Dimensional terrain and building data, and other sources such as remote monitoring of sensors in secure installations. (b) Present this heterogeneous information using novel displays [6].

The SGI™ Vizserver was used as it is a stable product for delivery of image and graphics information (Figure 2). The choice of Vizserver was guided by the following criteria which we had in mind: (a) should be able to provide accurate, real-time knowledge (b) should not require massive amounts of processing to occur at the client, yet be able to provide reasonable amount of functionality. (c) should not overwhelm the used with data
(d) should be able to provide delivery to disparate, portable devices. (e) The user should be able to manipulate the 3Dimensional data using the simple devices such as computer-mouse or touch-pad, and navigate that in real-time.

In the Global-Share system, our design focuses on removing all processing on the client side, except for display which would thus allow augmented reality devices such as NOMADs to be used for display

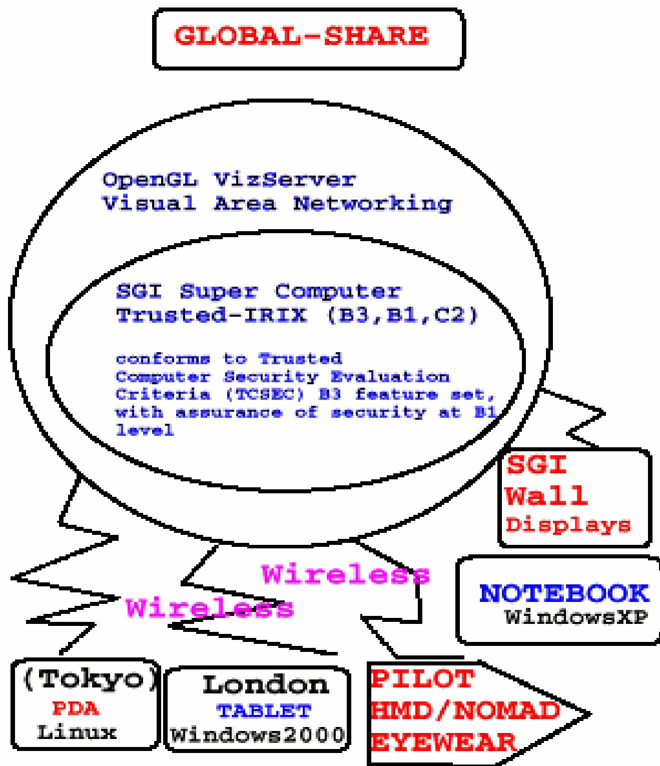

Figure 1: The System Diagram of the Global-Share Project.

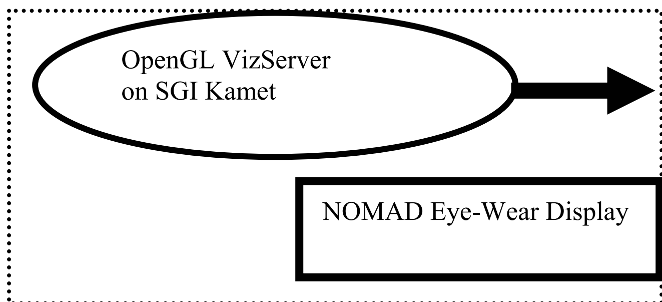

Figure 2: Vizserver arrangement used for our experiments.

We used the NOMAD™ Augmented Vision System as the client to be actually receiving data from the Vizserver (See Figure 2). Using these tools, we performed the following functions [4]:

(i) The SGI system runs a graphics program. (ii) The OpenGL Vizserver captures the frame buffer and sends it across the network. (iii) The client computer runs a client that accepts this frame (iv) The NOMAD Augmented Vision System is connected to this client computer and displays the image on its screen.

NOMAD Systems Study

We studied the suitability of the NOMAD system for our experiments. The NOMAD Vision system (Figure 2) is a hands free, heads up display unit. It provides the ability to superimpose digital information in front of one of the user's eyes, providing a virtual-view of a 17" screen on the retina. This unit provides mobility to the user, unchaining him/her from the desk. Using the combination of Vizserver installed[1] on one of our SGI system and the NOMAD system available as part of our Wearable Computing Laboratory, we were able to successfully demonstrate programs running on the SGI system to be simultaneously presented on a PC using Windows XP environment. Mouse-keys could be used on the PC to manipulate the OpenGL application running on the SGI system. There was no visual degradation of the OpenGL programs being displayed on the SGI system and that on the Windows XP PC. Next the output of the PC was connected to the NOMAD system and similar results were achieved.

The use of the NOMAD system for obtaining mobile information access was important for our goals. In our research, we intended to answer the following questions: First, would it be feasible to create a wearable system capable of mobile information access? In the event that it would be, what physical and psychological issues would confront users of such a system, and what means might alleviate any problems encountered? Finally, how broadly could such a technology be applied in contemporary society?

‡ email semwal@cs.uccs.edu

[1] We would like to thank Dave Lohman for help with installing the VizServer software on the SGI.

The first answer was relatively easy to obtain, with the construction of a basic wearable system using as its backbone a custom software package called InfoSys. The system developed for the NOMAD Study can be driven by any 802.11-compatible PDA capable of executing Microsoft DirectX applications and utilizes the Microvision ND1000 (NOMAD) Head-Mounted Display. The system was field tested by 9 individuals, each without knowledge of the results obtained by others. Testing criteria covered a wide range of issues, from hardware issues to software comfort. Most of the software results fell into expected patterns, with testers suggesting small changes and additional features. However, many difficulties were encountered with respect to the hardware. First, the head-mounted display responded unpredictably when combined with diverse testing systems, possibly a side effect of operating the device in DirectX's Retained Mode(640x480, 8 bit). Unlike most monitors, there was no effective means of adjusting the use of screen pixels in the display, and so the users often had to cope with unintended visual difficulties.

Most surprising was the effect on the user's mobility. 8 out of 9 testers indicated that they had difficulty performing a basic equilibrium test, wherein they were asked to walk across a room on a line marked with tape. This result may be due to the nature of the display, which causes one eye to focus on the text generated by the software while the other focuses normally. These results led the test group to conclude that the system made them feel clumsy, a problem which would lead to the significant social burden of self-consciousness. Interestingly, this equilibrium issue was only a problem when testers were in motion – when they were stationary most stated that they felt comfortable.

A secondary impact that was entirely unanticipated was the effect of the display's shading system, which displays a monochrome image in 32 shades of red. 7 of 9 testers indicated that the red coloring was hard on their eyes and noted difficulty in seeing edges when two adjacent regions only differed by a single shade. Additionally, 6 of 9 testers indicated a strong correlation between the light's intensity and an inability to view the environment through the image, especially when the background lighting was dim. However, reducing the image's intensity also substantially reduces the visibility of shading. The testing implies that a wearable system of this nature would be best served to utilize only a subset of available intensity levels, adjusting with the brightness of the environment.

Finally, all of the testers expressed concern for the physical robustness of the system. Most felt that the hardware is too delicate to be used in an environment with rough conditions such as physical secuirty. The projection unit in particular was a matter of great concern to the users – they felt uncomfortable moving around with such an expensive and relatively heavy device only secured to the rest of the headset by just a small plastic clip. A more robust construction would be necessary for such a system if it were in common use.

The last question, that of application in the real world once the kinks are worked out, is answered quite simply – such a system might be used for nearly limitless purposes. Industrial users could utilize it to speed the accomplishment of certain tasks. Librarians and grocers could use it to make restocking their shelves more efficient. The military could use it to supply battlefield information to soldiers, reducing infantry friendly fire incidents substantially. And not least, private users could utilize it for everything from reading books on the go to accessing streaming advertisements on store servers. The positive social impact which a strong wearable system for mobile information access might generate is truly inestimable. As limitations of head mounted display system were identified, we started looking for camera based augmented reality tools for security applications. We found that ARToolKit is very stable software packages (and several sites and mailing list already exist) as it is available on multi-platform (PC/SGIs).
We implemented a OO-class called ARC control which the user might find useful. The class and its functions are defined in the Appendix with this paper.

3. Haptic-Imprints

The goal of Haptic-Imprints is to develop a computer security application that uses the haptic device [2,3] for verifying the user that is trying to log on to the system. We want to: (a) create an application that allows one to write a signature using the haptic device, (b) break the signature into two parts and copy a part of it on to a memory device, and (c) then join the two parts again and compare it with the original image. We used C++ and OpenGL along with the Ghost API. With our experiments described above we have achieved to use the haptic device to display the signature on an SGI machine (Figures 3 and 4).

Haptic interaction adds the sense of touch to virtual environments. We used the PHANToM forcefeedback device for our experiments. Details of PHANToM can be found elsewhere. Here we briefly describe it as it pertains to our experiments. The PHANTOM Premium 1.0 is a system capable of 6 degrees of freedom input and 3 degrees of freedom output. This model provides a workspace of 5 x 7 x 10

----

‡ email semwal@cs.uccs.edu

inches (12.7 x 17.8 x 25.4 cm) or the range of motion approximate to the lower arm pivoting at the user's wrist.

Figure 3: Phantom Premium 1.0

The PHANToM provides an added advantage over the mouse and it provides us with a three dimensional workspace and also acts as a force feedback device. The phantom is easy to maneuver due to its six degrees of freedom and hence provides a great deal of flexibility. When connected to a computer, the device works sort of like a tactile mouse, except in 3-D. Three small motors give force feedback to the user by exerting pressure on the grip or thimble. GHOST API Library was used for our experiments and came with PHANToM device. The main idea of haptic-imprints was implemented as follows:
(a) First store the user's haptic signatures using the ghost library and the PHANToM.
(b) Split the haptic signatures into two.

.We started with learning the use of the Ghost API and used a 2-D application using the mouse to build a ' paint like' application which was used to write signatures using the mouse. This program was ported to the SGI to make it run along with the PHANToM.

We then integrated the application with the Ghost API and the PHANToM to use the two dimensional program in the three dimensional environment. The thimble (Figure 4) of the PHANToM was used to select draw modes. One chooses what to draw by clicking the thimble in the appropriate object box. One chooses where to draw the object chosen using the thimble in the drawing area. In all the cases the object is drawn only once. To draw a second object, you must select the object to draw again. Points are a bit different: Once selected you may draw as many points as you like before changing drawing mode. To quit the program we use the ‘Esc’ button.

More exploration would be required in the following areas: (i) Clipping the haptic scene in to two parts. (ii)Joining the clipped parts and comparing it with the original signatures. (iii) We also need to develop comparison algorithm to take into account the changes in the signature patterns even though they are by the same person.

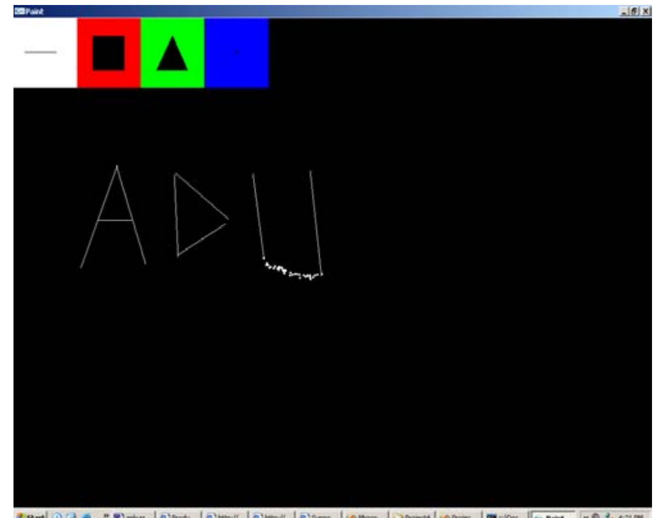

Figure 4: The haptics program using the PHANToM

4. Haptic Imprint within ARToolKit Discussion

User can store half-signature in a memory device and must supply the other half on-line to access the information. The system will have to merge the two signatures with the input supplied from the user. Finally, the user will be asked to provide the original signature and then compare the two parts with the original signature. Then provide him with the access to the computer The haptic device could be incorporated in ARToolKit experiments but we have not yet implemented this. ARToolKIt provides a capability of using cameras to look into real world. The real world can be annotated with simple planar-patterns which ARToolKit can recognize. Our idea is to display haptic-imprint is pattern form and use ARToolKit for recognizing the pattern. The extra layer of verifying haptic imprints and their image-equivalent would provide extra layer of security for our applications. The tactile sensors may be able to produce a unique output for each person, but transforming that into an image which an ARToolKit can recognize is a bit trickier. The images

----
‡ email semwal@cs.uccs.edu

can't be too complex, and ARTK won't be able to distinguish between images and patterns that look nearly identical. The haptic-data would need to be converted into planar images so that ARToolkit can distinguish it properly. In addition, we need to deal with mismatch of information: PHANToM can create arbitrary signature patterns whereas ARToolkit works off relatively simpler patterns.

ARToolkit may be useful for building security applications. This may be relatively simple, but powerful application for ARToolkit. Most people are familiar with laser grids - break a laser, the alarm goes off. This can be done in ARToolkit as well: break the detection of a marker, the alarm goes off. Train ARToolkit to see a marker, and start the alarm system. As long as it sees the marker, it knows everything is okay. If it doesn't see the marker, something must've disturbed it, and perhaps a robbery is in progress. This depends on whether ARTK can detect the same marker for long periods of time in low light conditions, and it may require multiple cameras positioned differently but it may be doable.

ARTK is a pretty useful tool, especially when one goes beyond its intended uses. Keep in mind, though, that ARTK can still be inaccurate and slow at times, and is not good for everything. Simple pattern recognition, though, seems to be good, and it can allow for overlays on the Nomad and other wearable screen devices without the hassle of learning image recognition. Performance is decent on the right system, just be careful about putting too many patterns into ARTK. Integrated with other systems, ARTK can be a real time-saver.

5. Summary and Further Research

Our research showed that systems that sharing information globally can be quickly assembled using existing products, and in our case this was done using SGI products. NOMAD displays or eye wear systems displays can be cumbersome when high mobility is desired such as in our application. New systems using nano devices could be an answer for resolving weight and other wearable issues. ARToolkit worked well in our experiments. We were able to use PHANToM force feedback device for implementing a simple haptic-imprint. Several research directions have emerged (a) Haptic devices for mobile/wearable applications, and (b) integration of haptics with existing AR tools such as ARToolKit. We plan to pursue both of them in future.

6. Acknowledgements

We wanted to thank Dr. Bill Ayen and NISSC for supporting the seed grant. Dave Lohman was critical assistance in dealing with SGI issues of our project. This research is partially funded from the research sponsored by the Air Force Research Laboratory, under agreement number F49620-03-1-0207. The US government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory of the U.S. Government

7. References

[1] I Poupyrev, DS Tan, M Billinghurst, H Kato, H Regenbrecht, N Tetsutani, Developing a generic Augmented Reality Interface. Pp 44-50, IEEE Computer March 2002.
[2] M.A. Srinivasan. Haptic Interfaces. In N. Durlach and A. Mavor, editors, *Virtual reality: Scientific and Technological Challenges*, chapter 4. National Academy Press. Report of the Committee on Virtual Reality Research and Development, Research National Council, 1995.
[3] Gabriel Robles-De-La-Torre and Vincent Hayward. Force can overcome object Geometry in the perception of shape through active touch, Nature, 412(6845): 389-91, July 26, 2001.
[4] The Importance of Trusted IRIX, www.sgi.com/pdfs/3185.pdf SGI's OpenGL VizServer, www.sgi.com/software/vizserver
[5] NOMAD Augmented Vision System, Microvison, www.mvis.com/products
[6]BP Center of Visualization, University of Colorado, Boulder, www.bpviscenter.com
[7] T Kato, T Kurata, and K Sakue, VizWear Active: Towards a Functinally Distributed Architecture for real-time tracking and context-aware UI, pp. 162-161, IEEE Computer Society, IEEE Conference on Wearable Computers (2002) Seattle, WA.
[8] Haptic-Jackets and Wearable Visual Displays for the Secure Global Share System by SK Semwal, a project proposal funded by NISSC, Summer 2003.
[9] Thomas L Martin, Time and Again: Parallels in the Development of the Watch and the Wearable Computer, pp. 5-11, IEEE Computer Society, IEEE Conference on Wearable Computers (2002) Seattle, WA.
[10] S Park, I Locher, A Savvides, MB Srivastave, A Chen, R Muntz, S Yuen, Design of a Wearable Sensor Badge for Smart Kindergarten, pp. 231-238, ISWC2002 proceedings, IEEE Conference on Wearable Computers (2002) Seattle, WA.

----

‡ email semwal@cs.uccs.edu

8. Appendix: ARToolKit Resource

The details of how we used ARToolKit [1] in creating a new wrapper-class (ARControl) and a subset of important functions is described below. This appendix may provide a starting point to researchers who are want to use ARToolKit for their experiments. Ofcourse large amount of information and other papers already exist for ARToolKit. Here is the ARControl class which we developed:

ARControl

One of the things which we have done with ARTK is to condense the ARTK functions into a streamlined, simpler interface. This interface is called ARControl. Instantiate a derived class of ARControl, call init and start, and the program is ready to roll, without having to worry about the ARToolKit calls in every program.

ARControl is an abstract class. One can inherit from this class in order to use its functions. This was done because many of the functions which we want ARTK to do are there, except one: How to draw the scene. The scene drawing method, used in the call to argMainLoop, is application-specific, and is thus virtual. So, inheriting and making this function is all that is necessary to use ARControl.

ARControl also handles patterns and pattern loading. This can be done from the command line. ARControl takes some parameters when the user calls the init function. These are the same as what the main function takes for convenience. These string values are passed onto ARTK, for example, -width=352, etc. A special command line parameter that has been implemented takes in pattern names and widths and loads them from file. Afterward, it puts them in a linked list for easy iteration. The parameter looks something like this when running a program:

Model -width=352 -height=288 -channel=0 -LoadPattern=Data/A.pat,160+Data/B.pat,160+Data/C.pat,160+Data/D.pat,160

Our interest is in everything after the -LoadPattern here. Once ARControl notices the LoadPattern Parameter, it knows that a list of pattern files and widths is coming. After the = sign, a pattern file is read. After the next comma, it reads a width. If the user wants to specify more patterns, it separates each pattern by a + and specifies the patterns as before. When referring to these patterns from the internal linked list in ARControl, pattern 0 is the first in the list, pattern 1 is the second in this list, and so on.

To get a frame from the user-implemented draw function inside ARControl, call the getFrame function. It will return a boolean of false if it failed. If successful, it stores a pointer to the frame in an internal class variable.

To get a pattern that ARControl loaded, call getPattern. It returns a special struct (Pattern) containing the pattern data. Or, if one is inside a function which was overriden, just use PatternList.getNext() to get the Pattern pointer from the list directly.

When you get the frame, it looks for markers and does most of the common per-draw stuff already. It fills an array of marker information like before. Iterate through the list of patterns, comparing their id's to the id's from the array of marker information. In this way you can react to each marker ARTK detects. This makes things better for the programmer.

When the ARControl derived class is destroyed, it makes a call to its cleanup function so as to exit ARTK gracefully. Also, if a program wants to exit abnormally, like making a call to the exit() function, you can call ARControl's cleanup() function to assure it does the proper shut down procedure.

Using ARControl, this is what the main function would look like:

```
int main(int argc, char **argv)
{
 ARCModel Model;
 Model.init(argc, argv);
 Model.start();

 return 0;
}
```

ARC Code is simple, convenient, and brief. All it requires is to make a drawing function, and derive the class as follows:

```
class ARCModel : public ARControl
{
 public:
  ARCModel();
  ~ARCModel();

  void swapMarkerLists();
  void makeMarker(Marker *marker, Pattern *pattern);

 private:
```

----
‡ email semwal@cs.uccs.edu

```
  unsigned int mode;
  LinkedList<Marker> *markerList, *prevMarkerList;
  LinkedList<Line> lines;
  LinkedList<Marker> masterList;

  void processPlanes();

  void arcMainFunc(void);
  void arcKeyFunc(unsigned char key, int x, int y);
};
```

The derived class doesn't have to be this complex. It can contain only a simple draw function, for example:

```
class ARCStub : public ARControl
{
 private:
  void arcMainFunc(void);
};
```

Thus using ARControl can make one's life a bit easier. There's more in ARControl, but the above is the basics which one needs to know to use it.

ARToolKit Startup

Following are the details which we used for initializing and using ARToolKit for our experiments. This is a brief summary of the functions that startup ARTK, as well as the overall ARTK startup process. This probably isn't a complete list, so please check [1,7] as well.

The first initialization call to ARTK is arVideoOpen:

arVideoOpen(char params[]) - Starts the process to open the video device. params is a string that can be used to send additional initialization options to ARTK is desired. If this string is empty, it'll default to a 640 by 480 camera on channel 1. If one wants to change these, construct a string similar to what one would do on a command line. For instance, using the string "-width=352 -height=288 -channel=0" will tell ARTK to use a 352 by 288 camera on channel 0. If these values do not work and camera can not be opened, use the "-debug" option to get some information on your camera. The return value indicates if the camera was opened correctly. If it didn't open, it will return a value < 0, otherwise it opened okay.

It's might be okay to not mess with the string parameter and just leave it empty, as was the case for our SGI machine in the wearable computing laboratory. On other machine such as home PC, it may be required to use them to open a camera. It's also good to get info on the camera by using "-debug", as it will print to standard output camera and other details. This call has to succeed. If not, you can't use ARTK with your current configurations and may have to change some parameters in your system a bit.

The next call is used to get the video resolution.

*arVideoInqSize(int *width, int *height)* - Returns the video resolution in the variables pointed to by width and height. Returns a value less than 0 if the call failed.

All this does is to provide the resolution of the video camera. This is needed for another function later.

Next, we'll need to tell ARTK where to locate the main camera file. This file holds some more details on the camera, or how to handle the camera.

*arParamLoad(char param[], int v, ARParam *wParam)* - Loads the camera parameter file from the location held in param. Usually the string is just "Data/camera_para.dat" The programs seem to work fine without setting other parameters. Returns a value less than 0 if it fails.

The next calls take more ARParams and the sizes saved earlier. They could initialize ARTK with more options, allowing the programmer to change the window/camera sizes.

arParamChangeSize(ARParam *wParam, int xSize, int ySize, ARParam *cParam)
arInitCparam(ARParam *cParam)

At this point, the patterns and markers are usually loaded. Next, the cParam is initialized.

```
#ifndef __APPLE__
 argInit(&cParam, 1.0, 0, 0, 0, 0);
#else
 argInit(&cParam, 2.0, 0, 0, 0, 0);
#endif
```

Apparently, there's a big difference between Apple and other machines here requiring a different setup. Up next, things get started, finally.

arVideoCapStart() - Starts... video capture.
argMainLoop(Pointer to mouse, keyboard, and drawing functions) - ARTK (really, GLUT) calls these functions when a mouse event, key press, or redraw is issued, respectively. The first two can be NULL, but the last shouldn't, as it we would usually want to draw something. Basically, it calls those GLUT functions which set up how to handle these events.

----
‡ email semwal@cs.uccs.edu

After that, the window with the video's feed shows up, unless you changed how ARTK draws to this window. To draw the screen with new stuff, swap the buffers, similar to GLUT.

argSwapBuffers() - Swaps buffers for GLUT, and essentially draws the screen.

Patterns

Patterns that the ARTK can use have to have certain properties:

1. Black and white only
2. Square
3. Thick black border

There are a few others that makers of ARTK and we recommend:

4. Use only asymmetrical images
5. Don't use images that are too complex
6. Using simple letters (A, B, C, D...) doesn't work too well, unless they have more detail.
7. If you're going to use some patterns in a specific environment, make and calibrate these patterns for this environment, even if you already have the pattern data files.

To load the patterns, all you have to do is make a call to ARTK with the name of the pattern.

*arLoadPatt(char *name)* - Loads a pattern into ARTK and searches for it when told to. The string parameter is the name of the file with the pattern in it, generated by the mk_patt program. It is assumed to be in the Data directory. The pattern also has to be specified in the object_data file.

The object_data file needs to have 3 lines added to it for each pattern that wants to be recognized: Pattern name, pattern location, and pattern width. Usually one just leaves the pattern files in the Data directory. The pattern name isn't used often. The width tells ARTK how wide the pattern should be interpreted as. This can be any width, but it may affect how ARTK places the marker in the scene. It's good to be consistent. If a pattern is twice as wide as another, it should have its width specified as twice that of the other. It is less confusing to just use patterns of the same width. Last, the first non-comment token in the file is a number containing the number of patterns in the file. ARTK docs state this is required. This may not be entirely necessary as the location and width of each pattern is specified at the time of creation in the program.
Now we have a set of patterns that ARTK knows to look for in its video. How do we respond when a pattern emerges in a frame? First we need to grab a frame from the video feed.

arVideoGetImage() - Returns an ARUint8 pointer to the frame data. If null, wait a bit, then try to grab the frame again.

Next, display the image from the video feed. If you don't want the image to appear, don't call these two functions.

argDrawMode2D()
*argDispImage(ARUint8 *frame, int a, int b)* - Displays the frame from the video feed pointed to by frame. The other 2 variables probably represent some kind of offset, but they have not been used or changed, since no real need has arisen.

This next call is what actually looks for the markers in the frame. It returns an array of patterns that it thinks matched in the frame.

*arDetectMarker(ARUint8 *frame, int threshold, ARMarkerInfo *markerInfo, int * markerNum)* - Find markers in the frame, using threshold to know how aggressively to look. Higher threshold may mean better results, but longer computation time. Returns a list of markers it thought matched the given patterns we made it look for. The number of patterns is in markerNum.

Last, set up ARTK for the next frame.

arVideoCapNext() - Called after done looking for patterns.

Now that a list of possible pattern matches has been given, we can iterate through this list to see if they match. Following is the code for matching these patterns in the ARControl class. It searches for a single pattern. Any other code that performs this search should look something like it.

```
Pattern *pattern = PatternList.getNext();

 for(i = 0; i < markerNum; ++i)
 {
  if(pattern->id == markerInfo[i].id)
  {
   if(cf == -1)
    cf = i;
   else
    if(markerInfo[cf].cf < markerInfo[i].cf)
```

---

‡ email semwal@cs.uccs.edu

```
    cf = i;
  } }
```
Basically, you compare the id's of the patterns. If they match, you know that that pattern was found in the frame.

It would be nice to know of a matrix that represented the position and orientation of a marker which the ARTK has found.

*arGetTransMat(ARMarkerInfo *info, double center[2], double width, double trans[3][4])* - Gets a transformation matrix (double trans[3][4]) of the pattern recognized in the frame. This represents its position and orientation. Center is used as an offset to further transform the matrix. Usually this is just 0,0 meaning the marker's center is where this matrix will transform to. You may also transform the width, altering the position and orientation further. Again, usually just stick with the width used when initializing the pattern.

Now, we'd like to convert the 3 by 4 matrix into something OpenGL can use, which is just an array of 16 doubles.

argConvGlpara(double source[3][4], double dest[16]) - Converts the ARTK transformation matrix (source) to an OpenGL matrix (dest) for use in the OpenGL matrix stack.

Now, we can overlay patterns in OpenGL simply by multiplying the current matrix with this one.

That's pretty much it for patterns. There are a few other things about patterns: One is that for each pattern searched for, four patterns are created. Each pattern is oriented in a different cardinal direction (right side up, upside down, sideways left and right. So, when searching for 4 patterns in a frame, ARTK is actually comparing 16 images to the frame. Keep this in mind for performance issues.
ARTK may have a function to directly deactivate patterns but that was not tested. Perhaps check arDeactivatePatt if this is needed. It might be worth looking into, since unloading patterns when they're over with could really help performance.

There is another part of this library called arMulti which deals with multiple patterns at once. We did not find any good documentation on this. Apparently, this library takes multiple patterns and puts them in a single coordinate system, or something similar. It requires another helper file, under the Data/Multi directory.

If more than one pattern with the same image is detected in the scene, there will be no way to distinguish between these patterns. Thus, the last pattern it detects for this image will be the only one it uses.

Please note that the above Appendix describes some of our experience with ARToolKit. A large community of ARToolKit users and mailing list exists. Please contact Dr. Mark Billinghurst for being on this mailing list.

----
‡ email semwal@cs.uccs.edu