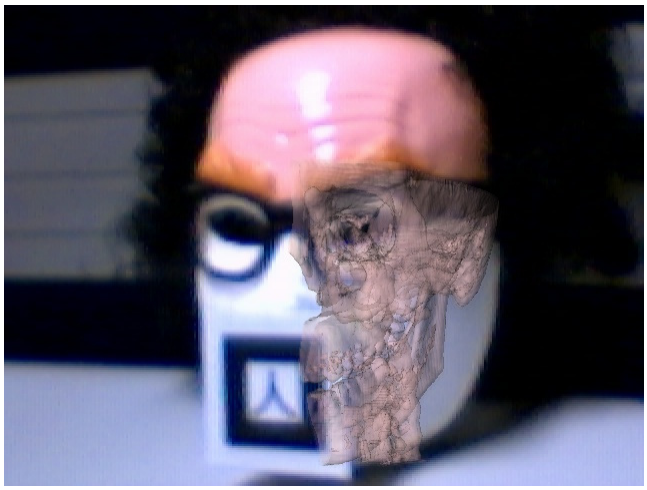
Augmented Reality:
Combining the ARToolKit, ITK, &VTK for use in a Biomedical Application



by David Sickinger

Computer Science Department
University of Auckland
New Zealand

CompSci 780 Summer Project Report
Supervisor: Burkhard Wuensche
January – February, 2004

Abstract

The goal of this project was to combine the ARToolKit, Insight Toolkit, and the Visualization Toolkit in order to implement a simple Biomedical Augmented Reality application. The result was what I refer to as the Augmented Reality Insight Visualization (ARIV) Software Suite that makes it easy to insert a visualization taken from medical imaging datasets into the video of a person. An example of a skull placed in the video of a facial mask is shown. The aim of this application is as a learning tool for kids along the lines of ARVolcano and MagiPlanet, both projects from HITLabNZ using the ARToolKit [Hit04].

Table of Contents

Section 1
Introduction

The goal for this research project was to explore the field of Augmented Reality (AR) by generating a Biomedical application. Last semester, in a Real-Time 3D Computer Graphics course, I began working with the ARToolKit which was downloaded from the HIT Lab at the University of Washington (www.hitl.washington.edu). The ARToolKit is an open-source vision tracking library written in C that can detect a cameras position and orientation relative to special physical markers. This makes it possible to use a web camera to capture video and to overlay virtual objects tied to the physical markers in real-time which can then be sent to the user's display. In a Visualization course, again from last semester, we were introduced to the Visualization Toolkit (VTK) from Kitware Inc (www.vtk.org). VTK is also open-source and is an object-oriented software system that contains hundreds of C++ classes for computer graphics and visualization. In addition to these two toolkits, I recently heard about the Insight Toolkit (ITK) which supports image registration and segmentation (www.itk.org). ITK is another open-source C++ toolkit that was sponsored by the US National Library of Medicine. ITK was created by six organizations (including Kitware Inc along with GE Corporate R&D, Insightful, UNC Chapel Hill, University of Utah, and University of Pennsylvania) to support the Visible Human Project that provides CT, MRI and cryosection images of a human male and female (www.nlm.nih.gov/research/visible/visible_human.html). Since I wanted to continue working in the Augmented Reality field for this project, I decided that it would be a great learning experience to combine these three toolkits. Both ITK and VTK are popular toolkits in the medical community and are well suited to support the development of a Biomedical AR application.

This report starts by giving a general overview of the Augmented Reality field, followed by a more specific focus on AR medical applications. Some basic research from my AR project of last semester is included in the general overview. Lessons learned from my experience with the ARToolKit is then shared to help a new user working with this software. Then just a small amount of background on ITK and VTK is given in the section that follows since there is plenty of documentation and numerous examples available from the web. The main section in the report on the *Implementation* appears next. This section was broken down into the stages used for building up the program with a discussion of each stage. The final program is then reviewed along with some concluding thoughts on how the project went. This report then wraps up with suggestions for future work and improvements.

Section 2
Literature Review

2.1 An Introduction to Augmented Reality (AR)

Azuma et al [Azu01] define Augmented Reality as a system that joins real and virtual objects into a real environment setting, happening in real-time, while registering (aligning) the real and virtual objects relative to each other. Figure 1 was defined by [Mil94] to show the possible mixed reality states. Milgram and Kishino address the subtle differences in defining these various mixed reality systems to allow discussions in the future. The authors define three factors to help distinguish between real objects and virtual objects: Extent of World Knowledge (EWK), Reproduction Fidelity (RF), and Extent of Presence Metaphor (EPM) [Mil94]. Azuma et al [Azu01] summarized [Mil94] to define the following mixed reality states:

- Virtual Environment (Virtual Reality) has virtual objects placed in a virtual world.
- Augmented Virtuality has real objects added to virtual ones in a virtual world.
- Augmented Reality has real objects combined with virtual objects, but the surrounding environment is real instead of using a virtual world.

Figure 1: Milgram's "Virtuality Continuum" (figure from [Azu01])

Monitor-based and other technologies are allowed using the [Azu97 and Azu01] definition of Augmented Reality which does not restrict display technologies. Some researches add the requirement that a Head-Mounted Display (HMD) must be used. There are two basic technology choices for combining real and virtual objects, either using optical or video technologies. Optical technology uses partially transmissive optical combiners in the HMD that allow the user to see the real world through them, but the optical combiners are also partially reflective allowing virtual images to be bounced off of them. Video see-through HMDs differ in that they combine a closed-view HMD with video cameras which provide the view of the real world. Video from the real world is blended with the virtual and sent back to the users display in the closed-view HMD. Azuma [Azu97] points out the advantages of both technologies. The advantages of optical displays over video are: they are cheaper and only require that the streaming video produce the virtual objects, though they reduce the amount of light the user sees from the real world they don't limit the resolution of real world objects, if they fail the user can still see, and there is no eye offset due to the placement of cameras. The advantages of video displays over optical are: they can completely obscure real world objects where

optical systems usually can not, the real and virtual delays can be matched by delaying the video of the real world, the real world can be digitized to provide more information for registration and other purposes, and it is easier to match brightness between real and virtual objects.

AR has an advantage over Virtual Reality (VR) in that is does not have high requirements for realistic virtual images since the user can still see the real world. Users can tolerate simpler virtual object models and for some AR system applications the use of color is not required. But tracking and sensing are more important in AR compared to VR due to the registration requirements [Bar01].

For a user to believe the illusion that real and virtual objects coexist, the objects must be properly aligned to a high level of accuracy. Several sources can introduce error, but they can be grouped into two categories which are static and dynamic. Static errors occur even when the user and objects remain still and can be caused by errors in the tracking system, system misalignments, and optical distortions. Dynamic errors occur due to system delays. It takes time to calculate the position and orientation, drawing images to the frame buffers, and for the images to be rendered to the display. These errors only become apparent when motion occurs [Azu97]. The latency times considered acceptable differ based on the application, but they are expressed in terms of milliseconds. In the [Bro99] paper, it states that Holloway [Hol95] found that a millisecond of latency translated into a maximum registration error of one millimeter. In 1999, the lowest achieved latencies were at best 40 to 50 ms [Bro99]. In medical applications, where surgeons could use AR to overlay images on a patient during surgery, this introduces too much error. But other applications do not have these same high demands.

For more information on Augmented Reality, a good starting point is the survey paper by Azuma [Azu97] along with the updates from Azuma et al [Azu01] which complements the survey with more recent advances. Another great source is the [Bar01] book where each chapter was handled by different individuals related to the AR field.

2.2 AR Medical Applications

I performed a search to see if anybody else had combined the ARToolKit with VTK and found that these two terms were mentioned together in only a few places. A posting at http://public.kitware.com/pipermail/vtkusers/2003-September/020243.html, which is in the vtkusers archives, asked if anyone had combined these toolkits but it had no replies. The individual who posted the question is from the University of Nottingham and there happens to be a five page PowerPoint presentation discussing the idea at www.crg.cs.nott.ac.uk/~cmg/Equator/Downloads/docs/enviro/2003-04-30/Data%20Visualization-alex-20030430.ppt. But this presentation seemed to be a proposal and I could find nothing else at this site indicating it had been done. Another vtkusers post inquired about AR applications with VTK but with no specific mention to ARToolKit (http://public.kitware.com/pipermail/vtkusers/2003-February/015956.html).

In [Rei03], it mentions that trying to adapt AVS or VTK applications to new interfaces like AR can be very difficult and it proposed using WireAR (based on WireGL from Stanford University) to allow any OpenGL output to be displayed in an AR setting.

CAVE's are virtual reality projection based systems and while not AR, I include here because a website shows a CAVE system using VTK rendered visualizations. Click on the *Applications* link at http://brighton.ncsa.uiuc.edu/%7Eprajlich/vtkActorToPF to see some cool screenshots. The *Parallel* link is also worth noting, it mentions that the VTK pipeline execution can be a bottleneck and provided a parallel example that can prevent a scene from "freezing" while a new isosurface is generated due to a selection being made (will see this happen in my application, press either the `F3` or `F4` function key).

Studierstube is software that supports AR development like the ARToolKit. There are two Studierstube AR projects listed at www.studierstube.org/research_master.php that are medical applications. The Augmented Reality Aided Surgery (ARAS) project can be found at www.vrvis.at/br1/aras and contains some screenshots and movies how it can aid a surgeon. The more recent medical AR project helps plan liver surgeries, refer to http://liverplanner.icg.tu-graz.ac.at/modules.php?op=modload&name=LSPSI&file=index to see several screenshots of that system.

The project steps used in an image guided surgery case are discussed at www.ai.mit.edu/projects/medical-vision/surgery/surgical_navigation.html.
Both screenshots and videos on the website demonstrate the process of constructing 3D models from biomedical imaging, using laser scanning to register segmented skin to a patient, the automatic registration process, and the enhanced reality visualization of a brain. Surgical instrument tracking can also be performed with medical imagery to show the position of internal probes.

Section 3
The Toolkits

3.1 The ARToolKit

As mentioned in the *Introduction* section, the ARToolKit allows a developer to overlay graphics on real-world video in real-time. Figure 2 was copied from HITLabNZ's website which shows the general steps involved (www.hitlabnz.org). Real world video is processed by searching each captured frame's binary image for a square shape which would represent the black border of a marker. The position and orientation of the potential marker is calculated relative to the camera and an attempt to match the potential marker is made with the marker template specified by the program. If a marker is identified, then the virtual objects that are tied to that marker can be transformed to match the markers alignment before being displayed. OpenGL can be used to render the virtual objects, or a VRML file of virtual objects can be passed as input to the ARToolKit.

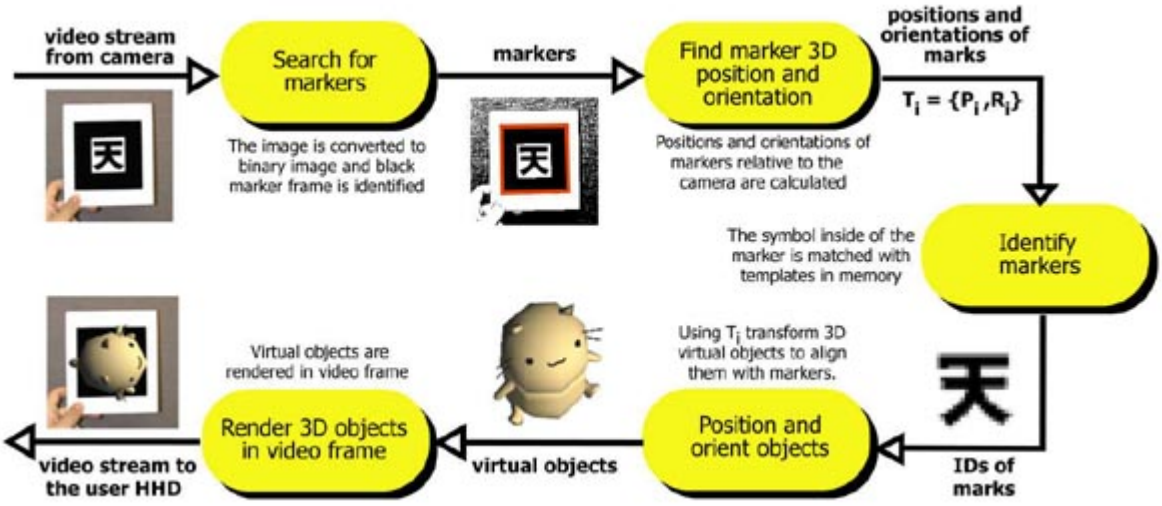


Figure 2: ARToolKit Diagram (figure from www.hitlabnz.org/blackmagic/how.htm)

3.1.1 Installation and Running the ARToolKit

I downloaded the ARToolKit from www.hitl.washington.edu/artoolkit/download.htm (Version 2.65 with VRML support). There is also a *Community* link at this website that gives access to a searchable mailing list as well as an online forum board. Since it took me some time to get up and running with the toolkit using .NET (Visual Studio 7) and because the ARToolKit code only contains a few comments, I included information here to help save time for a new user. After unzipping the source code, go ahead and double-click the *'Configure.win32.bat'* file which is found in the root folder. I recommend avoiding the *'ARToolKit.sln'* which is also at the root level since several warnings appear

about projects failing to open. It turns out that I never needed to use this solution once I became more familiar with what files I needed and the folders layout. Simply go to the *'examples'* folder and enter a folder of one of the three provided examples. Open the *'VC++ 6 Workspace'* file and convert to a *'VC++ 7 Project'* format. The projects *'Debug'* mode has the correct properties set up but the *'Release'* mode for the project does not. In order to compile the *'Release'* configuration, make sure to set the following which was taken from the *'Debug'* mode property pages:

C/C++ : General : Additional Include Directories : "../../include"
Linker : General : Output File : "../../bin/desired_filename.exe"
Linker : General : Additional Library Directories : "../../lib"
Linker : Input : Additional Dependencies : make sure have 3 *"libAR___.lib"* files

It is now possible to build a Solution in the *'Release'* configuration. The executable is placed in the *'bin'* folder found at the root level. The *'bin'* folder contains files and dll's that are necessary for the executable to run. Sometimes I copied these over to the same folder where the Visual Studio Solution was located so I could run the program from within the .NET environment. I should point out that in order to compile the code, the HIT Lab website states that in addition to the GLUT library, Microsoft's DirectShow SDK must be installed. I went through this process on my home computer after downloading Microsoft's DirectX 9.0b Software Development Kit (which was 218 MB) and it involved installing 5,261 files which took a long time on my machine. I now have a feeling the required files are already included in the *'bin'* folder, the *'DsRenderer.ax'* and *'DsVideoLib.dll'* files. I did not install DirectX 9.0b on the University computer I was assigned and had no problems compiling or running the executables (although was prompted to fun the following command: *'C:/WINDOWS/SYSTEM32/REGSVR32.EXE C:/* "*pathname*"*/DsRenderer.ax'*). So I recommend first attempting to compile the examples before going through this long installation process.

3.1.2 Camera Set Up and Calibration

Some type of video capture device must be connected for the executable to work. In my case, I used a Logitech QuickCam Express because it was the cheapest I could find on a list of supported web cameras found at www.hitlabnz.org/blackmagic/download.htm (I also downloaded the program that self-installs from this website about the America's Cup 2003 race and used it to try out my camera). I installed the Logitech camera driver provided with the camera before trying to run any executables. In the *'util'* folder at the root level of the ARToolKit, there are programs to calibrate a camera. However, I found the default camera parameter file found within the *'bin'* folder produced good results. The only consequences seem to be that I have to hold my web camera upside down (which I liked based on how I grip it) and I must check the *'Flip Horizontal'* box in the window that pops-up when the executable is launched (see Figure 3). An appropriate marker must be printed out prior to launching the executable in order to see results other than just a video stream. The marker files (in *.pdf* format) can be found in the *'patterns'* folder. In order to identify what pattern is required, look for a line like the following in any of the example C code files:

```
char *patt_name = "Data/patt.hiro";
```

In the case above, print out the *'pattHiro.pdf'* file. For better results, attach the pattern to a piece of cardboard so that as the pattern is moved around in the air it remains flat which produces better tracking results.

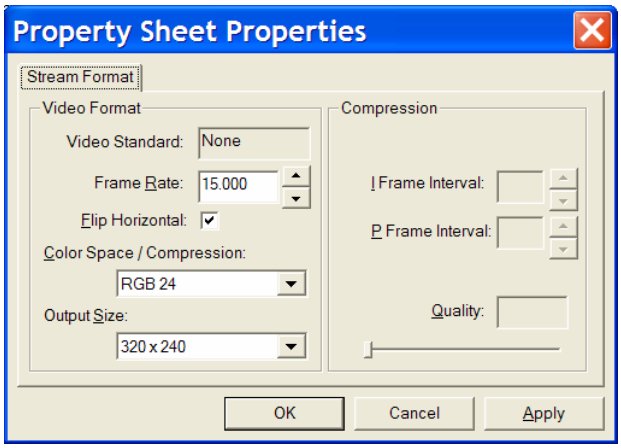


Figure 3: Camera Settings Pop-Up Window

I experimented with the settings found in Figure 3. Tried setting the frame rate to *'30.0'* as suggested by the instructions for the America's Cup 2003 program but I found even the simple demonstrations always reported less than 15 FPS (frames per second) with my setup and web camera. The FPS value is reported by hitting the `Esc` key to quit the program. I also tried various *'Color Space / Compression'* settings but saw no noticeable difference at the time. After quitting the program I always had an *'ASSERT Failed'* box pop-up which could be an issue related to the my setup and hardware versus anything with the ARToolKit.

3.1.3 ARToolKit Examples

The *'simpleVRML'* example renders a UFO when the marker is detected. Just a few modifications are needed in order to use a different VRML file for displaying other virtual objects. This example first reads in a *'bin/Data/vrml_data'* file. Opening this file shows that another file called *'ufoFly.dat'* is listed which can be found in the *'bin/Wrl/'* folder. The first line of the *'ufoFly.dat'* contains the name of the VRML file that is used which can be changed to another VRML filename. Just make sure the new VRML file is placed in the *'bin/Wrl/'* folder and run the simpleVRML executable again. I found that I needed to scale the two VRML files that I tested with by a large factor and had to experiment with the orientation and translations settings for good results. I tried following the *'ufoFly.dat'* file method for changing the scale, orientation, and translations but preferred using OpenGL to handle these changes. The OpenGL commands glScale, glRotate, and glTranslate can be added in the `draw_object(_,_)` function right after the `glLoadMatrixd(_)` call found in the *'examples/simpleVRML/simpleVRML.cpp'* file.

The OpenGL application *'examples/simple/simple.c'* renders a solid cube tied to the marker. More interesting virtual objects can be rendered by making additions in the `draw(_)` function. Basically what is happening is that the `mainLoop(_)` function grabs a video frame and draws that frame to the buffer. It then tries to detect markers by calling `arDetectMarker(_,_,_,_)` and if no markers are found, then a `glutSwapBuffers()` call is made and the loop returns. If a marker is found, then the transformation between the marker and the real camera is calculated before calling `draw(_)`.

After getting a feel for what is going on in the ARToolKit, it becomes relatively straightforward to create more interesting examples using the OpenGL approach. I was able to combine my Physically-Based Animation from an Advanced Computer Science course assignment with the ARToolKit. Also combined two tutorials taken from the www.GameTutorials.com website, one was of an animated Quake character and the other was a height-field combined with skybox example. HITLabNZ's website contains many projects involving the ARToolKit [Hit04], including some that are aimed at kids as an educational tool.

Note that the lighting in a room and the size of markers has an impact on the accuracy of the ARToolKit tracking results. A simple rule of thumb mentioned on the ARToolKit forum by Mark Billinghurst (HITLabNZ director) is that the tracking will work until the camera distance to marker exceeds ten times the width of the marker. There are also several parameters that are set in the *'include/AR/config.h'* file which I needed to reference at times when developing my own examples. For a deeper understanding on the transformations that the ARToolKit finds, refer to [Kat99a].

3.2 ITK and VTK

Since there is plenty of documentation, examples, and help available for both ITK and VTK from the respective websites as well as books about both toolkits, I will not go into much background here. I did find myself opening up and flipping through [Avi03], [Ibá03], and [Sch02] several times throughout the course of this project. Also made heavy use of the searchable mailing lists found at the respective websites for each toolkit. There is Doxygen (www.stack.nl/~dimitri/doxygen) generated API documentation for VTK that exists in a compiled HTML format that is easy to search. API documentation also exists for ITK but currently lacks the search ability and I found it more difficult and frustrating to navigate. The ITK website had some nice tutorials that were especially useful when it came to combining ITK with VTK.

VTK is a very large system that contains over 700 C++ classes. VTK's website lists many of its advantages but does also point out that VTK is not a super fast graphics engine since it is designed to be device independent and has dynamic binding. Based on the coding standards used for the VTK classes, the constructors and destructors are protected which prevents the C++ new and delete commands from being used. VTK objects are allocated via the static member function `New()` which adds a level of

indirection (similarly use `Delete()` for deallocation). This allows reference counting techniques to be used. Through VTK Object Factories, the function `New()` may return a more appropriate instance of a derived class, such as `vtkOpenGLRenderer` being returned when `vtkRenderer` is called on a Win32 system due to the `vtkGraphicsFactory` (useful notes at http://noodle.med.yale.edu/~papad/seminar/pdf/lectureb_3.pdf). Since I modified `vtkOpenGLRenderer` (see Section 4.1), other VTK applications on a Win32 system will be impacted if they link to this same VTK source build.

A difference exists between how ITK and VTK objects handle the `New()` operator. VTK uses raw pointers to hold the results from `New()` and thus `Delete()` must be called. But ITK makes use of SmartPointers to hold the results from `New()` and automatically takes care of releasing memory at the appropriate time and so `Delete()` is not used (useful posting at www.itk.org/pipermail/insight-users/2002-August/000728.html).

ITK was officially released in October of 2002 (ITK 1.0) and is a much newer toolkit than VTK. There were two releases of ITK in 2003 (1.2, 1.4) and the latest official release occurred during this project, in January of 2004 (ITK 1.6). I found it more challenging to use this toolkit, especially with the time frame given for this summer project. There is a new book set to be released in May that may prove useful for people developing future ITK applications (title will be *Insight into Images, Principles and Practice for Segmentation, Registration, and Image Analysis*, edited by T. Yoo).

There is another software system that is needed to support the ITK and VTK installation, and it is called CMake (www.cmake.org). It is an open-source cross-platform build system that was developed as part of the ITK project. CMake is used for the source code installations of ITK and VTK and it would almost be impossible to build them without. If using .NET, CMake basically creates a .NET Solution with the proper settings that can then be opened and compiled. I also created a CMake script file for my project and the entire setup process is described in Appendix A. It is possible to create a .NET Solution for my code without using CMake as long as all the project settings and links are properly set, but the CMake script makes it much easier. I found the [Mar03] book to be useful in a few cases but probably could have just used the CMake website.

Section 4
Implementation

I decided to approach the programming in stages. My goal was to build up and test each stage, where each stage would add a level of functionality that I would need for the final deliverable that I had in mind. It seemed natural that the first and most critical stage was to combine the ARToolKit tracking capabilities with the visualization capabilities of VTK. Without at least this ability, it would be hard to justify continuing. At times during this project, it seemed that more of my effort was spent performing detective work than actual programming, especially during the first two stages. This project required tracing through several layers of code to find what needed changing or experimenting with to understand the effects. In order to run the program, please refer to Appendix A which gives details on what files are needed, the toolkit versions used, and the camera hardware that was used in the development of this project. Note that the full paths to files from root directories are no longer always given in the following sections.

4.1 Merging Video with VTK Object Rendered to the Same Window

This stage took quite some time to develop, mainly because it forced me to become familiar with the structure of both the ARToolKit and VTK. The ARToolKit is composed of several files that each contain several functions that usually are closely related to the file's name they are found in. I also had to use the VTK documentation extensively to help in the understanding of the class hierarchy.

I felt at the beginning that I had two paths that I could pursue to merge the video feed with the VTK actors. The first option involved finding a way to display the VTK actors on the window that the ARToolKit opened and used. The second option was to pass the captured video frame pixels and the transformations calculated by the ARToolKit as input to the VTK pipeline. I decided to start with the first option because at the time I was not sure if I would run into problems with keeping the VTK pipeline running smoothly. I was successful with getting this first approach to work. As it turns out, I use the second approach to pass the pixels as input to ITK for processing and then the results are sent a new VTK window for displaying (see Section 4.9).

One of the keys to getting the first approach working was finding that the ARToolKit opens a GLUT window in `argInit2(_)` in *'gsub.c'*. So after some investigating and finding the method `SetParentId` for a `vtkRenderWindow`, I grab the handle of the GLUT window and then open a `vtkRenderWindow` and use the `SetParentId` call so it uses the same window.

It is helpful to understand how the ARToolKit handles the rendering process before proceeding any further, and the ARToolKit example *'simple.c'* is a good reference to look at. In the `mainLoop(_)` of this example is a set of calls to `argDrawMode2D()` and

`argDispImage(_,_,_)` which are both found in *'gsub.c'*. These functions take care of drawing the pixels just captured from the camera to the buffer. Then all potential markers in the captured image are found and investigated to see if a match to a known marker shape exists. If no marker is found, then the buffers are swapped and the program continues on to process the next image. If a marker was detected, then a call to `draw(_)` happens. The `argDrawMode3D(_)` just loads an identity matrix into the `GL_MODELVIEW` matrix and `argDraw3dCamera(_,_)` basically sets the proper `glViewport` size and loads the `gl_cpara` (camera parameters) into the `GL_PROJECTION` matrix. Notice there is no `glClear(GL_COLOR_BUFFER_BIT)` call typically found because that would clear the real-world image background. The `argConvGlpara(_,_)` sets the array `gl_para[16]` into a column-major matrix representation of the calculated transformation between the marker and real camera since the `glLoadMatrix` command expects a column-wise matrix. This is then loaded into `GL_MODELVIEW` before using a GLUT command to draw a solid cube. So in order to render the VTK actors, I needed to replace what happens in the `draw(_)` function with VTK equivalent commands.

For early experimentation, I made use of simple VTK cube and pyramid actors. Once I had the GLUT window and `vtkRenderWindow` tied together and displaying the video feed, I just took a small step by adding a VTK cube to be rendered. But the result of this test showed that the video feed would never show because the VTK renderer uses a background screen color when any actor gets added. So this required a modification to the `vtkOpenGLRenderer` class to prevent `Clear(_)` from clearing the `GL_COLOR_BUFFER_BIT` with a background screen color. After recompiling the VTK build and placing the new dll's in the proper folder, I finally had simple VTK actors appearing in the video feed.

4.2 Getting VTK Objects to Track Marker

My first attempts at getting this stage to work involved just using the matrix returned from the `arGetTransMatCont(_,_,_,_,_)` call, which is the transformation between the marker and the real camera. But when I did finally manage to get the VTK actors to track a marker (my attempt was a hack at this point), there was a noticeable drift in the VTK actors as the marker was moved around in the field-of-view of the camera. The translations and rotations behaved properly, i.e. if the marker moved to the left then the VTK actors would move to the left and so forth. But the VTK actors were not fixed firmly to the marker. I had never bothered to calibrate my web camera (which can be done by using programs found in the *'util'* folder of the ARToolKit source), and this fact was beginning to bother me. The drifting behavior puzzled me because I had not seen this in any of the other ARToolKit programs that I tested. I noticed another matrix called `gl_cpara` being used in the *'examples/simple/simple.c'* file and thought I should account for this matrix somehow as well, but was not sure on how to proceed.

It was not until I found Piekarski's [Pie01] posting on the ARToolKit mailing list archive that the pieces started to fall into place. He was trying to use the ARToolKit as a 3D tracker in an application and initially ran into problems, afterwards he posted what he had

learned. There is a default `gl_cpara` matrix that is used if the camera is not calibrated and it is used to set the `GL_PROJECTION` matrix. If the `gl_cpara` matrix is not used as in my first attempts, then the result is poor registration. But as long as this matrix is used in conjunction with the matrix from the `arGetTransMatCont(_,_,_,_,_)` call, registration results are good. This is due in part because the process of detecting the markers offsets them by this `gl_cpara` distortion and then the `GL_PROJECTION` brings everything back in line before rendering is performed. So even if the camera is distorted and has not been properly calibrated, the results are nice for applications that only involve image overlay as in most of the applications that are using the ARToolKit. However, 3D tracking applications (that require finding distances) must use a calibrated camera. Piekarski's [Pie01] posting contains some more tips for supporting a tracking type application.

With this new information in mind, I restarted my approach to this stage and found that I had to modify the `vtkCamera.cxx` and `vtkCamera.h` files so that I could set the view transform matrix (added `DSSetViewTransformMatrix(_)` method) and the perspective transform matrix (added `DSSetPerspectiveTransformMatrix(_)` method). The `gl_cpara` matrix only needs to be passed once to the `vtkCamera` using the `DSSetPerspectiveTransformMatrix(_)` call in `init(_)`. In the `mainLoop(_)`, the matrix from `arGetTransMatCont(_,_,_,_,_)` must be passed every time to the `vtkCamera` using the `DSSetViewTransformMatrix(_)` call. Also had to make changes to the *'gsub.h'* and *'gsub.c'* files from the ARToolKit so that I could gain access to the `gl_cpara` matrix (added `argVtkReturnGlcpara(_)` function).

Originally I was deleting the VTK window and recreating it depending if a marker was detected or not, but this seemed expensive and an ugly way to handle things. Then I was deleting and adding actors based on marker detection, but again not the cleanest solution. As I became more familiar with VTK and how to look towards the parent classes for some of the most useful functions, I found I could simply toggle actor visibility on or off.

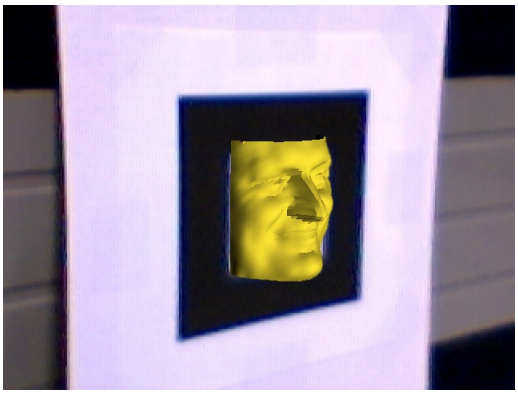


Figure 4: Fran Face Example A

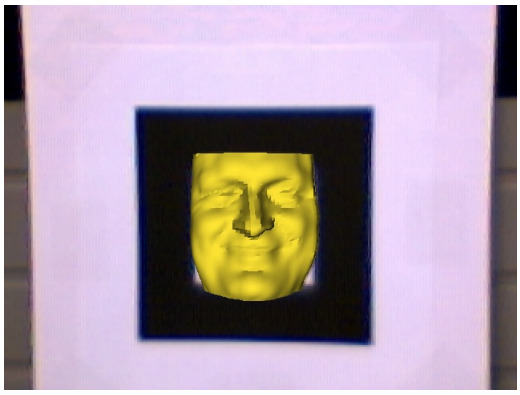


Figure 5: Fran Face Example B

At this point I decided to test out a VTK actor more impressive than a cube. Figure 4 and Figure 5 show the Fran face dataset that appears in some of the VTK examples. Deciding on this dataset caused me some initial grief by making me believe I had more problems to

solve. It turns out this dataset was acquired with an orientation of Z up, Y to the left, and X into the screen. Thought I was doing something wrong because the face would not track the marker properly (unlike the cube which was well behaved) until I set the actor's orientations properly (useful notes that discuss the way images can be acquired are at http://noodle.med.yale.edu/~papad/seminar/pdf/lecture5.pdf).

Near the end of this project, I made an attempt to use a cleaner approach for handling the changes to the `vtkCamera` and `vtkOpenGLRenderer` objects, but failed with my first try. VTK Object Factories can extend VTK at run-time and would allow me to replace the default `vtkCamera` and `vtkOpenGLRenderer` with my own versions that get compiled in with my application. This way I could build the VTK source files without swapping in any file changes. The biggest advantage would be that other programs that link to the VTK build would then be unaffected by my changes.

4.3 Resizing and Moving VTK Objects

I decided that my approach to positioning a dataset relative to a marker would have the user pressing keys to translate, scale, and rotate the actors into the desired place. The capability for the user to be able to save and load this transformation for future use would also be added. Then my ITK implementation idea would handle making any small adjustments in case the marker was moved relative to a person's face (see Section 4.10).

The position, orientation, and scaling of VTK actors is handled by a `vtkProp3D`. This class also allows a user transform to be set which gets concatenated with the actor's internal transformation. This user transform is initially set to the identity matrix and through various key presses (see Appendix B for Key commands) I change the translational and rotational values. A scale value is stored which can also be changed by key presses. Then just before rendering, calls are made to have the VTK actors update their user transform and scale value.

After a user is satisfied with the placement of the actors, it made sense to allow them to save these settings for later use. At first I looked within the vtkActor class on a way to do this using the print statements but it appeared way too much information would be written out including the datasets. Since only need to save the scale value and user transform for the actor, I decided to manually write and read the data to a file. The `F11` and `F12` keys handle the saving and loading of this information.

4.4 Handling Key Input

The ARToolKit handles key presses with the `keyEvent(_,_,_)` function already. However, it was more involved than originally planned to get the special keys (such as the cursor keys and `F1` - `F12` keys) that I wanted to work. The `Esc` key was already being used by the ARToolKit to quit the program and to report the FPS before exiting. The hex value of the `Esc` key is used in the check of equality against any key press, where letters

use their char value. I tried adding support for other special keys after finding a key code chart that gave all the hex values. But checking for the extended scan code of these keys in hex format did not work. Adding a `vtkRenderWindowInteractor` to watch for special key presses does not work as it takes control of the window and interrupts the `mainLoop(_)` flow. So I traced the `argInit(_,_,_,_,_,_ )` call from within the `init(_)` function and found it was defined in the *'lib/SRC/Gl/gsub.c'* file. After following a few more calls within this file, found a `glutKeyboardFunc(_)` call within the `argInitLoop(_)` function. This GLUT command was listed in [Woo99] but no explanation was given on how to handle special keys. After some searching on the web at www.opengl.org, found the special callback function `glutSpecialFunc(_)` defined that handles the special function keys and directional keys. Note that the escape, backspace, and delete keys are being handled as ASCII characters by the `glutKeyboardFunc(_)`, and this is what caused me some initial confusion. So I modified both the *'include/AR/gsub.h'* and *'lib/SRC/Gl/gsub.c'* files which require the *'libARgsub.lib'* library to be rebuilt (Section 4.2 also had changes to this library). Also note that the `argMainLoop(_,_,_,_)` function that is called within `main(_,_)` now requires four parameters instead of three within any program using the ARToolKit. But special keys are now supported, and the arrow keys can be used to maneuver objects which is all I wanted and was the reason for this exercise. Please see Appendix B for a full list of Key commands currently supported.

4.5 Capturing JPEG Image Output

For report purposes, decided I needed a way capture JPEG's. I have found that using shareware programs to capture screen output from a video source does not produce the best quality images. So a search using VTK's documentation and searchable archived mailing list returned the `vtkJPEGWriter` class that just requires a `vtkImageData` object be set along with a filename to use for writing. The class `vtkWindowToImageFilter` reads data from a `vtkRenderWindow` and can be used as input in the pipeline to the `vtkJPEGWriter` object. This stage along with the Section 4.3 stage seem to be the only stages that were completed in the timeframe I planned on. A side advantage of supporting the capture of JPEG's is that if this develops into a learning tool for kids, the option could be provided to the user's to capture a screenshot they like and then either allowed to print or email themselves a copy.

4.6 Clipping Data Sets

The original Biomedical AR application discussed with my supervisor involved the visualization of half a human's brain overlaid on the video of person's head. In order to support this, a way was needed to manipulate the biomedical datasets to get only half (or some other subsection) of the dataset. After some investigating, I found a Tcl script example in *'VTK/Examples/VisualizationAlgorithms/Tcl/ClipCow.tcl'* that demonstrates a process to accomplish this. The class `vtkClipPolyData` is used with an implicit function as input (can be a simple `vtkPlane`) to decide what data gets clipped. The result in

Figure 6 shows the result of clipping the cow dataset with a plane where one side is shown in red and the other side is displayed in wire-frame (could just discard one side).

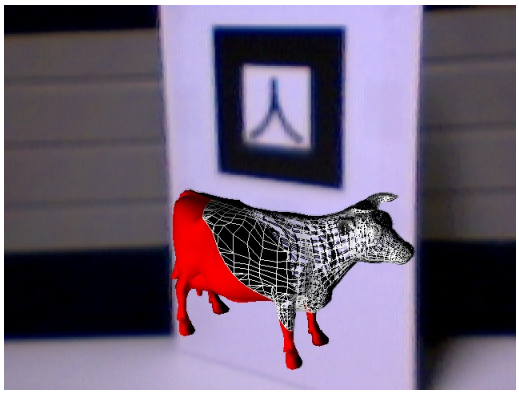


Figure 6: VTK ClipCow Example Tied to AR Marker

In my final implementation, the `F1` – `F4` function keys are used to select what halves of the skull are displayed (have quarter-resolution dataset and full-resolution dataset).

4.7 Marker Highlighting

For some of the different ITK applications I was contemplating, I needed a way to insure the marker plane was as perpendicular to the camera's view direction as possible before capturing any image to pass to ITK for processing. In order to accomplish this, I took an idea from the *'util/mk_patt/mk_patt.c'* file on how a new marker is created for the ARToolKit. It displays a red and green square around a new test pattern, which is simply the *'patterns/blankPatt.gif'* square with thick borders that contains a new shape inside, when the user holds the camera directly above. The user is supposed to orientate the camera until an alignment is matched where the lower left corner is red and the upper right corner is green. Then the user presses the left mouse button and is prompted to enter a filename to store the newly created marker. Refer to [Kat99b] for more details on this procedure.

My current implementation of highlighting the marker really just aids the user in aligning the X and Y coordinates so that the edges of the marker are horizontal and vertical. It is using the `marker_info->vertex[][]` which contains the X-Y coordinates of the four corners of the square detected. The midpoints between these corners are calculated in order to draw the red lines in Figure 7 (hard to see the two red lines in the figure, but they are inside of the green lines). The center point of the marker from `marker_info->pos` is used along with fixed offsets in both the X and Y directions to draw the green lines. The idea for this implementation was to grab a frame to pipe into ITK when the red lines fell between the green ones indicating a good X-Y alignment. But note that the vertex order from `marker_info->vertex[][]` is not fixed to the same vertices as the marker is

rotated around, so the bottom line is that the orientation is only known by looking at the shape inside the square (since four possible orientations exist inside a square).

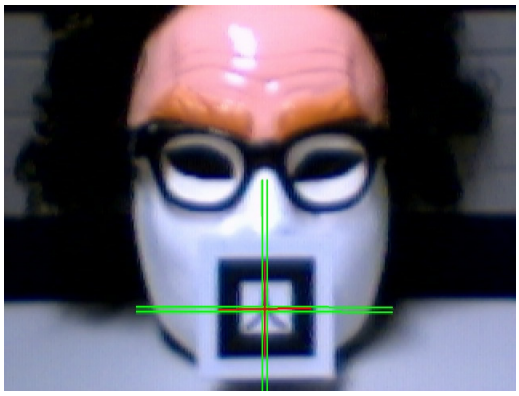


Figure 7: Highlighting Marker

I had to give this problem some more thought since it is possible to tilt the marker in such a way that the X and Y axis align but the marker is not perpendicular to the camera view direction. A better approach would be to compare the edge lengths between the four corner vertices to make sure they are within a tolerance of each other, thus indicating as close to a perfect square as possible and so camera direction is directly over the marker. For my *ITK Based Implementation Idea* (Section 4.10), this is all that would be needed and it would be restrictive to align the X and Y axis as well. Due to time constraints, I did not get around to implementing this type of marker highlighting.

4.8 Importing Images into ITK

My first thought on how to handle transferring images into ITK for the analysis planned in Section 4.10 was to use the same approach taken in Section 4.5 to capture JPEG's. Since ITK will only require a few snapshots at most for any of the applications I had in mind, it seemed like an easy idea to use the `vtkWindowToImageFilter` to capture an image into a VTK object and then pipe this to ITK for analysis. Any image result from ITK could then be piped back to VTK for viewing or the transformation calculated in ITK could simply be used to modify the VTK actor's user transform and scale value.

But it seemed that I could never catch the marker without the VTK actors being rendered as well, even though I tried several combinations of events (sometimes the actors would appear ghost like). At the time I was working on this stage I was using the `RemoveActor(_)` command when no marker was detected. A VTK `Render()` call is still needed to force the pipeline to update and prevent the VTK window from causing rendering lags in the program. I used a key press to turn on a flag to capture the screen when the marker was detected and to make sure all the actors were removed before calling the `vtkWindowToImageFilter`. I tried swapping buffers before this command and other various combinations but the actors always seemed to appear. When I added

some `Sleep(_)` statements and watched the behavior, it seemed that if a marker had been picked up and the actors had been rendered, when the marker became undetectable it seemed to take the VTK actors 3-4 frames to completely disappear (gradually got lighter like a ghost). Not sure if that was caused by the `Sleep(_)` command, or if something else was going on, or if my program had errors at the time. But the behavior in the program now seems consistent with other demos I have done with the ARToolKit in terms of appearance. However, if I had more time I would like to go back and investigate this some more. I would want to see if any commands are executed after a VTK `Render()` call is made before it completes the process. I added a `vtkMyRenderCommand` class in the ARIVSuite header file and left as a comment a `renderCallback` object in `init(_)` of the ARIVSuite C++ file.

I mentioned back in Section 4.1 the thought of passing data directly from the ARToolKit into the VTK pipeline, so I changed my thoughts on how to import images into ITK and decided it would be a good exercise to pass captured images directly into ITK.

4.9 Passing Images from ARToolKit to ITK to VTK

Started this stage by first trying to get the pipeline from ITK to VTK working. The [Ibá04] tutorial was helpful here, especially with the CMake scripts and indicating that the `itkImageToVTKImageFilter` class was needed to create the pipeline. Experienced quite a bit of trouble getting the *'InsightApplications-1.4/Auxiliary/vtk'* built on my home machine using CMake though, stated that I needed Python and Tcl installed. But think this was due to some earlier settings I selected when running the CMake scripts for the ITK and VTK builds (was able to proceed after installing Python and Tcl). However, the build on the University machine went smoothly (although switched to ITK version 1.6).

The video frame that the ARToolKit captures is stored in the `dataPtr` within the `mainLoop(_)`, which is an unsigned character pointer to the frame pixels. In order to test the ITK-VTK pipeline along the lines of what I would need, I made use of an ITK example that covered how to import image data from a buffer into ITK. The example creates a buffer of a binary sphere that is then imported to ITK, it can be found in '*Examples/DataRepresentation/Image/Image5.cxx'*. I incorporated this example into the ITK-VTK pipeline and got it to display the results in a new pop-up VTK window. From reading parts of [Ibá03] and researching the ITK website, I learned that currently only one image channel can be passed at a time from ITK to VTK. So to create a color image to display in the VTK window, I would need to pass the red, green, and blue channels separately and then combine them (which can be done) in VTK before displaying. As it turns out, the ITK registration filters also only use one image channel. So I just calculate the greyscale values from the `dataPtr` and pass that to the `itkImportImageFilter`. According to the ARToolKit file *'config.h'*, it appears that for Win32 systems the pixel format is stored as BGRA (blue-green-red-alpha) versus RGBA (red-green-blue-alpha). Note also that a GLUT window expects pixels to be defined from the bottom left corner but that an ITK image expects pixels defined from the top left and so reordering was required.

The camera captures images at a size of 320x240 and this is also the size of the new VTK window that pops up to display the pipeline results after pressing the *'i'* key. Notice that the window the ARToolKit opened is double the size and this is due to the second argument in the `argInit(_, 2.0,_,_,_,_)` call. A `vtkRenderWindowInteractor` was added to the new VTK window which halts the video flow in the other window until the user presses *'q'* or *'e'* to quit and return to the video window. The user can make use of the mouse to alter the appearance of the image in the new VTK window before closing this new window (basic image processing). Right now I do not subtract the time that the new VTK window is open from the FPS calculation in the ARToolKit, but this should be done in the future to get accurate FPS results.

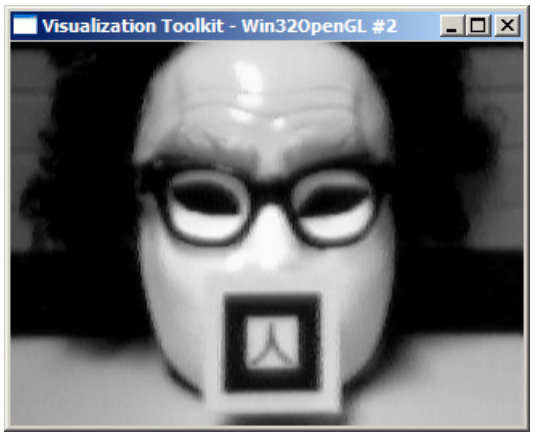


Figure 8: New VTK Pop-Up Window

After successfully getting this pipeline to work, I believe it was a better choice to have the ARToolKit handle displaying the video versus feeding every captured video frame into the VTK pipeline, especially since only one channel can be passed at a time. But I think this approach is the much cleaner solution for an ITK image analysis application. It was at the completion of this stage that the ARIV Software Suite was officially born (otherwise would have been the ARV Software Suite).

4.10 ITK Based Implementation Idea

My ITK integration idea would have involved a user translating and scaling a skull dataset relative to a marker placed on a person's face (or mask) until the results appeared good and then storing that transformation along with the marker's relationship to the face. In order to establish a relationship between the marker and person's face, I was planning on having the user draw a line by mouse clicking in the new VTK pop-up window after pressing the *'i'* key. The user would have been prompted to draw the line connecting the outside corners of the person's eyes, making sure to always draw from the left-to-right. Then the program would find the middle point of that line and define that as the center of a 2D coordinate system with a normal to the line as the other axis direction. Then using

ITK filters, this image would be segmented to find the marker and the position would be related to the 2D coordinate system (including orientation). At this point the information would be saved for later use since it would be written to a file. While this application would require user interaction, even the ITK website had a comment on how image analysis is hardly ever fully automatic. Also did some quick tests on the image of my actual face with the `vtkRenderWindowInteractor` which seemed to indicate that finding the borders of my face might be difficult to extract nicely and in a consistent manner. So this made the drawing-a-line by the user approach more appealing to me.

Now the user would be allowed to move the marker by either shifting the marker's placement on the face or by rotating it, but making an attempt to keep the marker as coplanar as possible to the stored version. By launching ITK again and drawing another line between the eyes, a comparison could be made to the stored transformation and the actor's user transform could be adjusted to account for the change in the marker. Then only some fine tuning in the Z-direction by the user would be required. But this would be very useful for a user when restarting an application (assuming marker coplanar enough to stored version). Unfortunately, due to time constraints and the learning curve for ITK, I was not able to get through this stage. I did find that the marker input files (i.e. *'Data/patt.kanji'*) repeat the 16x16 pattern of pixels three times in succession with slightly different greyvalues and that all four orientations within a square are provided. Originally I was planning on reading this data in as an ITK model to help find the marker in the image and to determine the orientation. But then changed thoughts to first finding all squares in the image based on an ITK example of extracting all circles in an image (see *'Examples/Segmentation/HoughTransform2DCirclesImageFilter.cxx'*). Then would look within the squares extracted for known marker shapes.

4.11 Bringing Implementation All Together

Considering the amount of time I had for this project, I decided to start with a skull visualization idea taken from the *'Examples/Medical/Cxx/Medical2.cxx'* file included in the VTK source download and expand from there. Isosurfaces are extracted from a volume dataset that is generated by reading in 93 2D-slices (images). The example showed how to use isosurface values corresponding to skin and bone for extraction. I combined this with the previous program stages to produce the different screen shots found in this section. I found it useful to use a mask of a person (actually use two separate ones that fit together) instead of trying to point a web camera at my face for testing and capturing screen shots. My program code is included on the CD that was turned in with this project and the main filenames to look at are *'ARIVSuite.cpp'* and *'ARIVSuite.h'*. Make sure to follow Appendix A which covers what files were modified for the ARToolKit and the VTK source builds to the ARIV Suite works properly.

Figure 9 and Figure 10 show the differences in quality between using the 93 quarter-resolution (64x64) 2D-slice dataset versus the 93 full-resolution (256x256) 2D-slice dataset. But it takes more time initially to process in the full-resolution dataset.

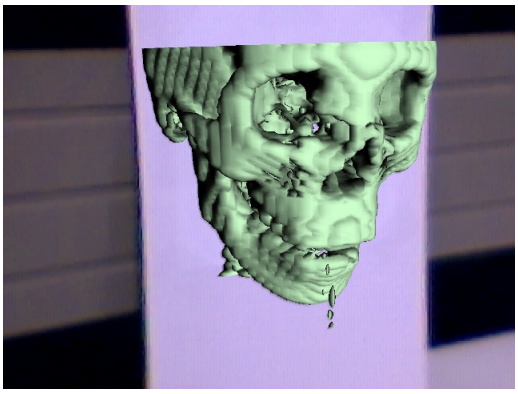


Figure 9: Quarter-Resolution Dataset

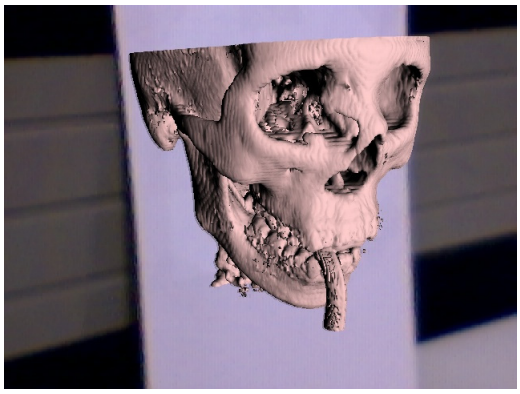


Figure 10: Full-Resolution Dataset

Figure 11 and Figure 12 each show the left side of the skull's in quarter-resolution and the skull's right side is at full-resolution. The same plane is used to clip both datasets.

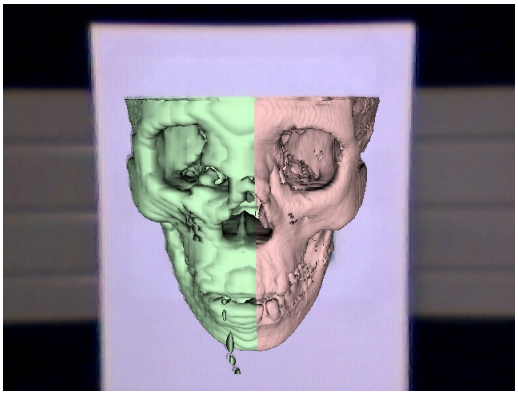


Figure 11: Resolution Comparison A

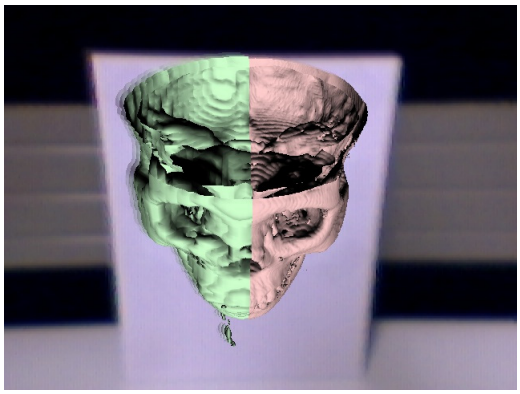


Figure 12: Resolution Comparison B

Figure 13 and Figure 14 just show two different views of the full-resolution dataset after clipping it with a plane (marker moved between the two Figures).

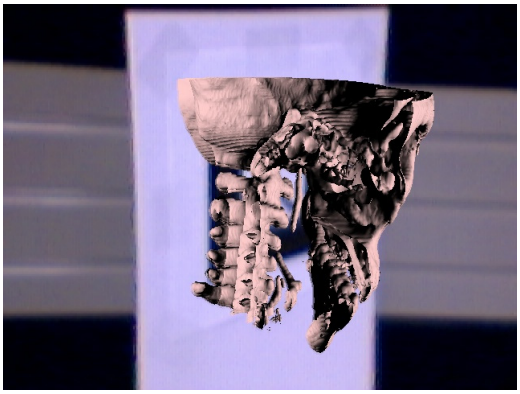


Figure 13: Full-Res, Half-Dataset, View A

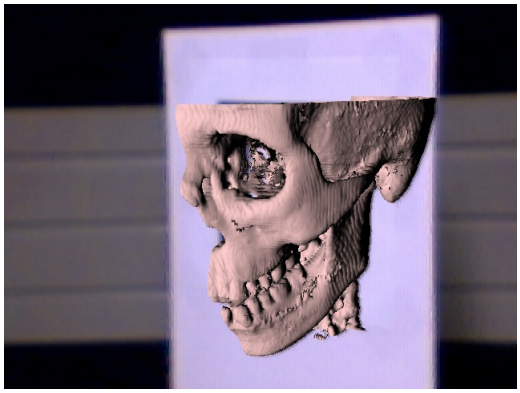


Figure 14: Full-Res, Half-Dataset, View B

The marker (*'pattKanji.pdf'*) that is attached to the mask in the following screen shots was printed at a scale of 60%, although the full marker size was used in the screen shots above that have no masks in them. Figure 15 and Figure 16 show the impact of changing the VTK actor's opacity value. View A has the default opacity setting of 1.0 while View B uses a value of 0.2 for the opacity.

Figure 15: Opacity Comparison View A

Figure 16: Opacity Comparison View B

Figure 17 and Figure 18 are personal favorites of mine from the snapshots that I took.

Figure 17: Personal Favorite View A

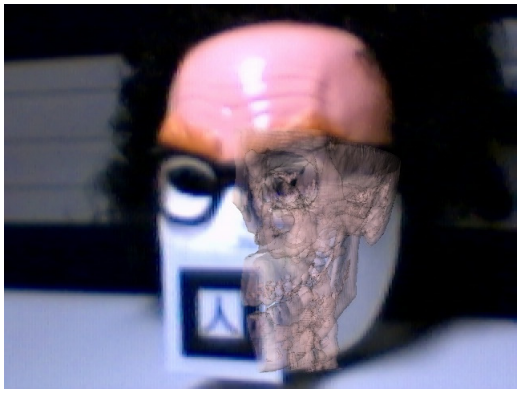


Figure 18: Personal Favorite View B

The drawback of having a visualization of a skull is that the eye sockets and the mouth stand out clearly if the user does not spend enough time resizing and moving the actors. In that sense it would be better to visualize a brain which would not provide the same visual cues of a mismatch.

Section 5
Conclusion

I set out with a goal of combining three toolkits to learn as much about them along the way as I could and to develop a simple Biomedical AR application. Early in the project, I had my doubts that the toolkits could be merged after many of my first attempts failed. But I felt a great sense of accomplishment after combining the ARToolKit with VTK and then finally with ITK. While I was not able to complete the main ITK based stage due to time constraints and the learning curve involved, I feel that overall the ground work has been completed to allow more impressive Biomedical AR applications to be developed in the future using these three toolkits together. In this learning process, I was forced to go and explore many files, classes, and functions to find the information that I needed. And this increased my understanding of especially the ARToolKit and VTK (just got started with ITK). I also liked the exposure that I got from working with large toolkits like ITK and VTK, it was good to see how they manage, track, and test so many C++ classes.

I feel that with some more hard work, this program could develop into a nice learning tool about the human body for kids that they would find fun to interact with. Of course kids can look at pictures of the brain and skull to gain an understanding of the human body, but I think it would really hit home if they could look through a HUD at one of their parents and see a brain or skull visualization superimposed over them. I plan to monitor how other AR projects, like HITLabNZ's ARVolcano and MagiPlanet [Hit04] that are already being shown in science centers and museums, are received by kids.

Section 6
Future Work

I had a chance to attend HITLabNZ's 2004 Annual Consortium meeting while working on this project and while there I noticed most of their applications make use of multiple markers to improve tracking results. This was especially true if the application used small marker sizes. I printed out the *'pattMulti.pdf'* page at a scale of 75% which made the six markers on the sheet fairly small. I then attached four markers to the eyeglasses of the mask and two on either cheek as shown in Figure 19. Believe this change would greatly improve the tracking results. Could develop a weighting function to use when several markers are detected in an image to get more consistent results between frames. Refer to [Owe02] for more insight on the fiducial markers and correlation coefficients.

Figure 19: Multiple Markers for Improved Tracking

The *'Examples/GUI/Win32/SampleMFC'* folder in VTK gives an example of a Microsoft multi-document interface. I was originally trying to avoid drop-down menus, extra windows, buttons, and sliders because I was not sure how this would fit into a HUD application. Again from the HITLabNZ Consortium meeting, there were demonstrations that had all of these. So extra features could be added to the program, perhaps allowing a user to select what datasets to view and allow them to interact in a separate window to make changes before viewing the AR result in another window. Could also allow a user to grab an AR frame to process it using VTK's volumetric ray caster to generate a highly detailed image in another window which could be saved as output. It is worth checking out the 3D Slicer software tool (www.slicer.org). It is open-source software that uses VTK for the visualization of medical data.

Definitely more with ITK processing can be added. Note that there is a VTK class for capturing video input called `vtkWin32VideoSource`. Perhaps try using ITK for tracking the markers and not use the ARToolKit. Maybe even try without markers and perform tracking using the face and eyes, and just render a few images (not real-time). See the posting at www.itk.org/pipermail/insight-users/2003-February/002401.html.

Incorporate BrainWeb datasets that can be obtained from either the BrainWeb website (www.bic.mni.mcgill.ca/brainweb) or ftp://public.kitware.com/pub/itk/Data/BrainWeb. Note that the `vtkMetaImageReader` may be needed which requires a later source version than VTK4.2 (or at least need this newer class).

Had plans to allow interactive clipping of the datasets but ran out of time to try this. Also wanted to experiment with saving the clipped datasets in .vtk format files after the `vtkContourFilter` is applied and see how much faster these datasets would load over the raw datasets. For the full-resolution dataset, extracting the skull can take over twenty-five seconds to load into memory on my machine.

Thought about changing the code to allow a user to save a series of frames in JPEG or ppm format but did not get around to it. But this would support a post-processing application to combine the frames into a video. Could try Multimedia SDK to generate the video and compare this against the quality of a movie generated by a shareware window capture program.

I am sure there are many more improvements that could be made, but that concludes my list for now. Definitely more testing and comparing results is needed, I had hoped to do some of this but just not enough time. But this might reveal improvements that can be made in combining the toolkits.

References

[Avi03] *The VTK User's Guide, Updated for Version 4.2*, by L. Avila, S. Barré, B. Geveci, A. Henderson, W. Hoffman, B. King, C. Law, K. Martin, W. Schroeder. Kitware Inc, 2003, pp. 324.

[Azu01] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, B. MacIntyre, "Recent Advance in Augmented Reality". IEEE Computer Graphics and Applications, 21(6):34-47, November 2001.

[Azu97] R. Azuma, "A Survey of Augmented Reality". In *Presence: Teleoperators and Virtual Environments 6*, 4 (August 1997), pp. 355-385. www.cs.unc.edu/~azuma/azuma_publications.html.

[Bar01] *Fundamentals of Wearable Computers and Augmented Reality*, edited by W. Barfield, T. Caudell. Lawrence Erlbaum Associates, 2001, pp. 797.

[Bro99] F.P. Brooks, "What's real about virtual reality?". IEEE Computer Graphics and Applications, 19(6):16-27, Nov./Dec. 1999.

[Hit04] HITLabNZ Projects Webpage, www.hitlabnz.org/index.php?page=projects.

[Hol95] R. Holloway, "Registration Errors in Augmented Reality Systems". PhD dissertation from the University of North Carolina at Chapel Hill, 1995. www.cs.unc.edu/Publications/PHDAbstracts.html#Holloway.

[Ibá04] L. Ibáñez, W. Schroeder, "Getting Started with ITK + VTK". Tutorial from www.itk.org/CourseWare/Training/GettingStarted-II.pdf.

[Ibá03] *The ITK Software Guide, Updated for Version 1.4*, by L. Ibáñez, W. Schroeder. Kitware Inc, 2003, pp. 539. Free download version available from www.itk.org/ItkSoftwareGuide.pdf.

[Kat99a] H. Kato, M. Billinghurst, "Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System". Proc. of the IEEE & ACM IWAR '99, pp. 85-94, 1999.

[Kat99b] H. Kato, M. Billinghurst, R. Blanding, R. May, "ARToolKit PC version 1.0". 1999. ManualPC1.doc from www.hitl.washington.edu/research/shared_space/download.

[Mar03] *Mastering CMake, Updated for CMake Version 1.8*, by K. Martin, B. Hoffman. Kitware Inc, 2003, pp. 157.

[Mil94] P. Milgram, F. Kishino, "A Taxonomy of Mixed Reality Visual Displays". IEICE Trans. Information Systems, E77-D, 12, 1994, pp. 1321-1329. http://search.ieice.or.jp/1994/files/e000d12.htm#e77-d,12,1321

[Owe02] C. Owen, F. Xiao, P. Middlin, "What is the best fiducial?". First IEEE International Augmented Reality Toolkit Workshop, Darmstadt, Germany, pp 98-105, 2002. http://metlab.cse.msu.edu/tracking-prepared/charles-owen-art02.pdf.

[Pie01] W. Piekarski, "Re: ARtoolkit Calibration Problems", mailing list archive. See www.hitl.washington.edu/artoolkit/mail-archive/message-thread-00133-Re--ARtoolkit-Calibratio.html.

[Rei03] G. Reitmayr, M. Billinghurst, D. Schmalstieg, "WireAR - Legacy Applications in Augmented Reality". www.ims.tuwien.ac.at/media/documents/publications/wirear2003.pdf

[Sch02] *The Visualization Toolkit, 3rd Edition*, by W. Schroeder, K. Martin, B. Lorensen. Kitware Inc, 2002, pp. 496.

[Shr00] *OpenGL Reference Manual, 3rd Edition*, edited by D. Shreiner. Addison Wesley, 2000, pp. 692.

[Woo99] *OpenGL Programming Guide, 3rd Edition*, by review board M. Woo, J. Neider, T. Davis, D. Shreiner. Addison Wesley, 1999, pp. 730.

Appendix A
Getting Program Up and Running

The University computer I was assigned is already set up to run my program. All that is missing is a web camera (and software driver if different than the model I used, which was a Logitech QuickCam Express). Print out the *'C:/dsic001/ARIVSuite/pattKanji.pdf'* marker and attach it to a piece of cardboard. Just go to the *'C:/dsic001/ARIVSuite/bin'* and launch the executable and the program should run.

The steps required to get this program running on a new machine are reviewed below. Some of the path names can be changed, but note that the CMake script for my program will need to be updated to reflect any pathname changes. I will include the source for the toolkit versions I used on the CD I submit with this project, space permitting (so check before downloading). Here are the major steps:

1) Install CMake by downloading the Windows installer from www.cmake.org which will have a name like *'CMSetup181.exe'*. CMake installation is pretty straight forward, refer to website for help. If nothing appears on the Start menu (due to permissions), launch *'CMakeSetup.exe'* from *'CMake/bin'*.

2) Install VTK next – Download the source files from www.vtk.org. I placed the source files in *'C:/dsic001/VTK-4.2'* and created a new folder for the binaries called *'C:/dsic001/VTK-4.2_b'*. Due to the modifications needed for the `vtkCamera` and `vtkOpenGLRenderer` objects, copy the three files found in *'replace_VTK-4.2_files'* folder on the CD to *'C:/dsic001/VTK-4.2/Rendering'* (files are: *vtkCamera.cxx*, *vtkCamera.h*, and *vtkOpenGLRenderer.cxx*). Open CMake and set the folders for the source code and binaries. Click configure, and then change *VTK_USE_HYBRID*, *VTK_USE_PARALLEL*, and *BUILD_SHARED_LIBS* to *ON*. Click *OK* and close CMake. Go to the binaries folder and open the .NET *'VTK.sln'* that was created, and then build the *ALL_BUILD* project in the Release configuration (can also build Debug). After the build is complete, go to the bin folder and copy all the dll's created and paste into *'C:\WINDOWS\SYSTEM32'*.

3) Install ITK next – Download the source files from www.itk.org. I placed source files in *'C:/dsic001/ITK-1.6/InsightToolkit-1.6.0'* and created a new folder for the binaries called *'C:/dsic001/ITK-1.6/InsightToolkit-1.6.0_b'*. Open CMake and set the folders for the source code and binaries. Click configure, then change *BUILD_SHARED_LIBS* to *ON* and since the examples take a long time to build, turn *OFF* the *BUILD_EXAMPLES*. Click *OK* and close CMake. Go to the binaries folder and open the .NET *'ITK.sln'* that was created, and then build the *ALL_BUILD* project in the Release configuration (can also build Debug). Note that the same build configurations must be used

in applications that use both ITK and VTK or problems occur. After the build is complete, go to the bin folder and copy over the one dll that is created and paste into *'C:\WINDOWS\SYSTEM32'*.

4) Install an ITK application next – Download the appropriate source files from www.itk.org, note that all the InsightApplications are not needed, just the Auxilary/vtk folder is all that I used. I placed these source files in *'C:/dsic001/ITK-1.6/InsightApplications-1.6.0/Auxiliary/vtk'* and created a new folder for the binaries called *'C:/dsic001/ITK-1.6/InsightApplications-1.6.0/Auxiliary/vtk_b'*. Note that I had to modify the CMake script file, so copy the file found in *'replace_InsightApplications_CMake_script'* folder from the CD and paste into the source folder. The problem I experienced on the University machines is that the CMake module *'FindVTK.cmake'* always found the VTK build on the P: drive. Now open CMake and set the folders for the source code and binaries. Click configure, then *OK* and close CMake. Go to the binaries folder and open the .NET Solution that was created, and then build the *ALL_BUILD* project in the Release configuration (or make sure to match the configurations of ITK and VTK builds).

5) I created a root folder called *'C:/dsic001/ARIVSuite'*. Copy the *'source'* and *'bin'* folders from the CD and paste these two folders inside this root folder. Open CMake and set the source code folder (*'C:/dsic001/ARIVSuite/source')* and binaries folder (*'C:/dsic001/ARIVSuite/bin')*. Click configure, then *OK* and close CMake.

6) Extract the *'ARToolKit2.65vrml.zip'* file to the root folder created in Step 5. Check to make sure the path *'C:/dsic001/ARIVSuite/ARToolKit2.65vrml/lib'* is valid, if another folder level was added due to the zip extraction then fix so matches. Two files need to be replaced in the ARToolKit. Copy *gsub.h* from the *'replace_ARToolKit_files'* folder on the CD and paste into *'C:/dsic001/ARIVSuite/ARToolKit2.65vrml/include/AR'* folder. Copy *gsub.c* from the *'replace_ARToolKit_files'* folder on the CD and paste into *'C:/dsic001/ARIVSuite/ARToolKit2.65vrml/lib/SRC/Gl'* folder. Within this folder open the libARgsub.vcproj and build the Release configuration which will then update the *'libARgsub.lib'* library which is found two folders up.

7) Now open the *ARIVSuite.sln* in *'C:/dsic001/ARIVSuite/bin'*. Should be able to compile the Release configuration and run from within the .NET environment. If prompted, may need to run *'C:/WINDOWS/SYSTEM32/REGSVR32.EXE C:/* "*pathname*"*/DsRenderer.ax'* which seemed to happen every time a new *pathname* was used on the University machine. Recommend changing the *'Linker-General-Output File'* property to place the executable directly in *'C:/dsic001/ARIVSuite/bin'* folder. That way the executable runs properly outside the .NET environment because it has access to the necessary data files.

Appendix B
Key Commands

KEYS	ACTION
ESC	Quit and report FPS
c	Toggle ARToolKit translation modes (continuous / one shot)
t	Change the ARToolKit threshold value
F1	Toggle on/off Actor - boneLeftQuarterSide
F2	Toggle on/off Actor - boneRightQuarterSide
F3	Toggle on/off Actor - boneLeftFullSide
F4	Toggle on/off Actor - boneRightFullSide
F5	Toggle on/off Actor - bone
F6	Toggle on/off Actor - skin
F7	Toggle on/off Actor - outline
F8	Toggle on/off Actor - faceActor
F9	Toggle on/off Actor - cowActor
F10	Turn off all Actors
F11	Save Actor transform to file
F12	Load Actor transform from file
Arrows	Translate Actor in X and Y (left, right, up, down)
PageUp	Raise Actor (translate Actor in Z)
PageDown	Lower Actor (translate Actor in Z)
q *and* w	Rotate Actor in X by (-1° / +1°)
a *and* s	Rotate Actor in Y by (-1° / +1°)
z *and* x	Rotate Actor in Z by (-1° / +1°)
[*and*]	Scale Actor by large increments (smaller / larger)
- *and* =	Scale Actor by small increments (smaller / larger)
d	Toggle on/off printing marker status in DOS window
i	Launch ITK which opens new window - Press either “q” or “e” keys to close this new window and return
j	Create JPEG image and save to file
m	Toggle on/off marker highlighting
o	Toggle on/off opacity setting
r	Reset Actor transform to default