Cross-Language Interfacing and Gesture Detection with Microsoft Kinect

Phu Kieu
Department of Computer Science and Mathematics,
California State University, Los Angeles
Los Angeles, CA 90032

Lucy Abramyan, Mentor
Section 317F—Planning Software Systems,
Jet Propulsion Laboratory, California Institute of Technology
Pasadena, CA 91109

ABSTRACT

Stage provides an immersive visualization environment for mission operations, in which we integrate Google Streetview and the Kinect Natural User Interface (NUI). The Kinect is controlled through a C++ dynamic link library, where it is interfaced to other environments: Java, Unity, and Teamcenter Visualization. Through the Kinect API, we retrieve the positions of various skeleton points, and use them to perform gesture processing and identification. The goal of the project is expand the uses of Stage to more than mission operations, and to explore the degree of seamlessness that we can achieve by using the Kinect NUI. This will highly depend on the types of gestures that are used to perform commands as well as the accuracy of the detection algorithms. We found that the learning curve of our system to be low since the number and complexity of the commands are low. However, other factors such as fatigue limit the feasibility of the NUI as a primary input method.

1. Background

Stage is an experimental visualization environment for mission operations. For it to serve this purpose, it should handle three functions: input, display, and control. We define the following terms as follows:

- Input—Collect and process data from device (robot, spacecraft, etc.)
- Display—Display data from device to user in a usable format
- Control—Process user's input and converts to commands recognized by device

As of now, the display function is fairly complete, and we have fabricated input by using Mars' ground data and Google Streetview. Control functions are purely client-side for the time being.

The distinguishing factor of Stage is that the display is 12 ft. high, 270 degree cylindrical shaped. Figure 1 illustrates the physical configuration of Stage. Three projectors are required to illuminate the whole screen. The advantage this set-up has compared to traditional display methods is that it maintains the relative position of objects.

The code-base of Stage is written in Java. The 3D environment is rendered and managed by the Ardor3D library. In 3D rendering, the maximum view angle a camera is allowed is about 179. A piecewise rendering trick is used to achieve the full 270 degree view angle.

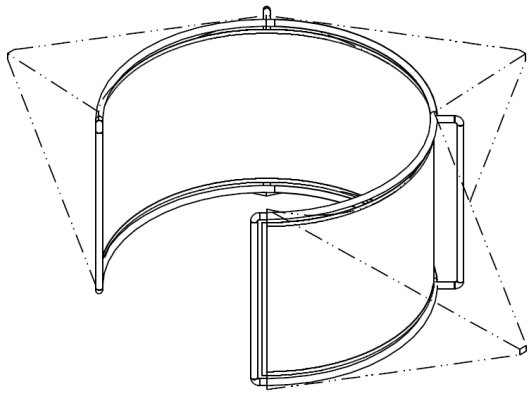

Figure 1. Stage Configuration

2. Google Streetview Undocumented API

We have found that the raw Google Streetview image data is not available through their public API. Therefore we access them through an undocumented API described by Jamie Thompson. It describes the sequence and methods on retrieving the raw image data of a given latitude and longitude pair. Note: There is a problem regarding the assumption of image dimensions (refer to Appendix A) in Thompson's article.

The result is a spherical projected image of the location desired. This spherical image can be displayed correctly by creating a sphere and mapping the Streetview image to the sphere as a texture within the 3D scene.

3. Google Streetview Integration

We have added support for Google Streetview in Stage. It takes a pair of latitude and longitude coordinates, and retrieves the image data from Google's servers. The image is processed and rendered with the 270 degree view geometry. Basically, the behavior of the program is similar to Google Earth, but with a wider view angle. This includes the ability to walk around various locations on earth, and the program will appropriately load and transition between images.

4. Cross-Language Kinect Interface

Stage supports keyboard, mouse, and Nintendo Wii Remote inputs. Now we would like to include Microsoft Kinect support. The challenge associated with this is that Stage is written in Java, and the Kinect API supports C/C++ and C#. We solved this problem by writing interface code on each end: Java and C++.

Our C++ interface is a dynamic link library (DLL) which uses the Kinect API and exports the Kinect functions. The memory mapping of the data structures is known, so any program that can call functions from this DLL should be able to retrieve data coming from the Kinect. We call this the wrapper DLL.

The Java side of the interface accesses the wrapper DLL through the Java Native Access (JNA) library. The JNA library allows us to load the DLL and call functions through the DLL and store the output to a memory block. The raw data is remapped into a proper Java-native data structure that we can use to perform gesture processing.

We use a similar architecture set-up to interface the Kinect to Unity, a 3D game development tool, and Teamcenter Visualization, a CAD design tool. We will focus more on the implementation details used for Stage, however. Refer to the figures found in Appendix B.

5. Gesture Detection

The method used to detect gestures in Stage is similar to the operation of a deterministic finite automaton (DFA). It is a clearly defined DFA in which at each state the system evaluates a condition, which can return either -1, 0, or 1. A value of -1 signals the system to return to the beginning state, 0 will result in no state change, and 1 will cause the system to advance a state. All states are ordered and there is no skipping of states or backtracking except for returning to the beginning state. A gesture is detected when the final state is reached.

Figure 2 is a DFA of a four state gesture. We generalize that a gesture with more states are more complex. For example, our implementation of a wave gesture consists of four states. The gestures used to control Stage are probably the simplest type of gestures possible, which are composed of two states.

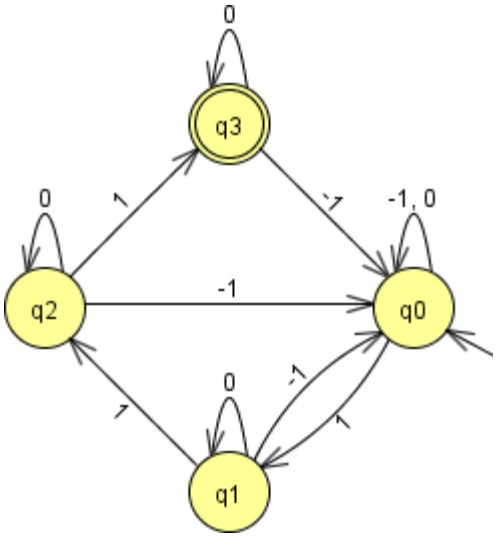

Figure 2. DFA of Four State Gesture

6. Challenges

Ardor3D was quite difficult to work with. The documentation was difficult to find that we thought it did not exist. It required inspection of the source code to understand the behavior when we needn't involve ourselves with the implementation details.

The gestures we used in Stage are simple. If we wish to implement more complex gestures, the deconstruction of the gesture will be difficult. The wave gesture we have implemented as a test did not have a good detection rate. It resulted in primarily false negatives. There is also the possibility of multiple gestures interfering with each other.

7. Discussion

In developing and testing the gestures, we have found that they can allow a user to finely control the camera and perform movements. However, we have had issues with unintended control, which is inherent with this type of input. It

requires the user to focus only on controlling Stage. We have also found that the behavior of the Kinect is not consistent across different machines.

Another issue with using the Kinect is user fatigue. The average user will become tired after about one hour of use, whereas traditional input methods can be used for several hours before fatigue sets in. While this disqualifies the Kinect as a primary input method, it may be a valuable tool as a secondary input device.

8. Future Work

Stage is currently in a dismantled state. For future work, we would like to deploy the software and use it on the actual Stage platform instead of performing tests on standard display devices. This would allow us to test factors such as whether one Kinect sensor is sufficient for the whole environment.

In terms of using Stage for a mission operation, there is lots of work needed to be done. To begin, a stitching algorithm for constructing either spherical or cylindrical projected images from real-time camera data can be written. Another important function for Stage to have is the ability to issue RAPID commands, which is a communications protocol for controlling robots.

Future performance requirements will necessitate that Stage be ported over to a faster programming language, such as C/C++ or C#. We are currently having performance difficulties with processing and rendering Streetview images with the Java environment. It seems unlikely that Stage will be able to process and 30-60 frames per second of panoramic images without significant performance enhancements.

Acknowledgments

The work described in this paper was supported by the National Science Foundation under grant number 0852088 to California State University, Los Angeles. It was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

We would also like to acknowledge David Torosyan for collaborating with the work done on Stage, the CURE Program, and Section 317F.

References

Ardor3D. Retrieved 15 Aug 2011. <http://www.ardor3d.com>.

Thompson, Jamie. *Google Streetview Static API*. 15 May 2010. <http://jamiethompson.co.uk/web/2010/05/15/google-streetview-static-api/>.

UNITY: Game Development Tool. Retrieved 15 Aug 2011. <http://www.unity3d.com>.

Vrsinsky, Jan. *360Cities – Panoramic Photography Blog*. 10 June 2010. <http://blog.360cities.net/tag/google-earth/>.

Appendix A. Google Streetview Undocumented API

Given the latitude/longitude or panoId of a location, retrieve the metadata. The panoId is a unique identifier string for a location, which is used internally by Google. The metadata can be retrieved at the following URLs:

```
http://cbk0.google.com/cbk?output=json&ll=[latitude],[longitude]
http://cbk0.google.com/cbk?output=json&panoid=[panoId]
```

The output type can be changed to 'xml' if xml output is desired.

The metadata is used to retrieve the panoId of the location. With the panoId, it is possible to retrieve the image tiles and construct the spherical projected image. Several tiles put together form the entire image. The dimension of each tile is 512x512 pixels.

Tiles are accessed at the following URL:

```
http://cbk0.google.com/cbk?output=tile&panoid=[panoId]&zoom=[zoom]&x=[x]&y=[y]
```

Possible values for *zoom* vary from 1-5, but values of 4 and 5 may not be available at all areas. The x and y parameters are used to indicate the coordinates of the tile piece; these values are always positive and start at (0,0). The maximum values of these coordinates vary, even for identical zoom levels. In the case of an invalid *zoom*, x, or y parameter, it will return a single channel image which will be all black. This makes the detection of out of bound values simple. Retrieve the tiles by iterating over x and y until the boundaries are found.

Once the full image is retrieved, it may need to be cropped. To correct the image height, start from the bottom and eliminate all consecutive lines that contains all black pixels (within ε pixel values—the current implementation uses $\varepsilon = 10$). Correcting for width uses the property of spherical image projections, where width $= 2 \cdot$ height. Eliminate pixel columns from the right until this property is satisfied. Figure 3 is an example output produced by this method; Figure 4 is the same image as viewed from Stage. Figure 5 is a mockup image of the Stage environment.

Figure 3. Google Streetview Output

Figure 4. Google Streetview Output Viewed from Stage

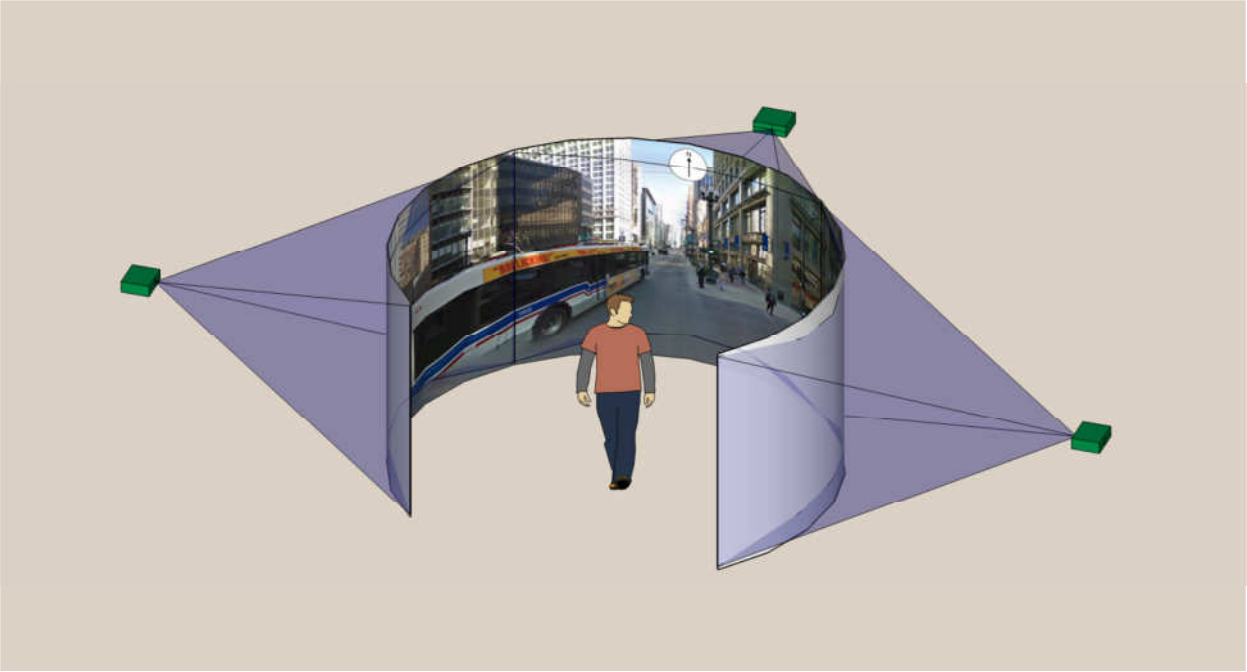

Figure 5. Stage Mockup

Appendix B. Architecture Diagrams

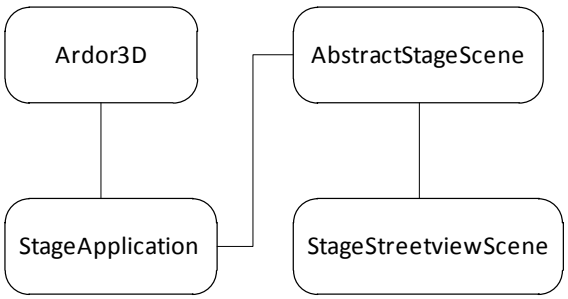

Figure 6. Stage Diagram

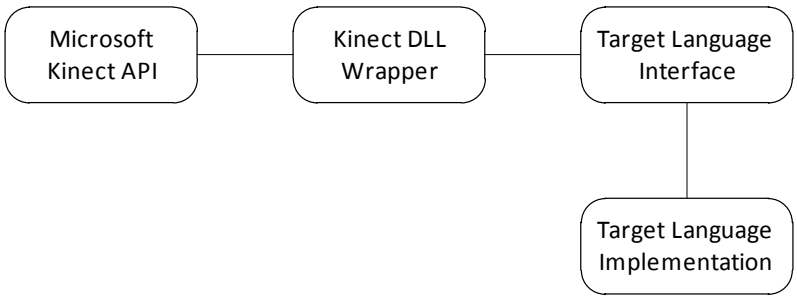

Figure 7. General Kinect Interface Diagram

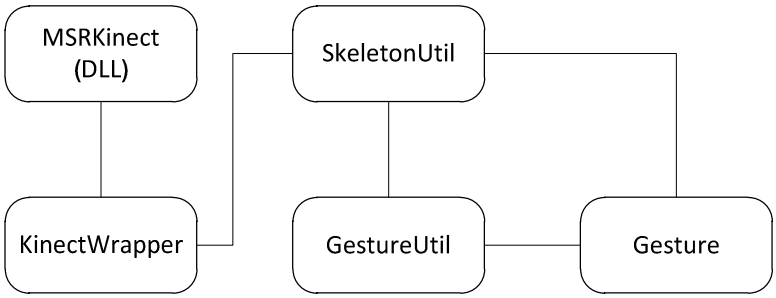

Figure 8. Java Kinect Interface and Implementation Diagram