Real-Time Virtual Window Simulation

Christopher Giri
Connecticut College Department of Computer Science
Advisor S. James Lee

Abstract

This project involves real-time manipulation of live video feed with user head-tracking data to provide a viewing experience similar to that of looking out a window. While concepts related to head tracking to enhance viewer experience have been considered in the past, we seek a comprehensive approach with the added integration of live video. This paper will discuss various implementation techniques used to optimize this experience.

1. Introduction

With this project, we hope not only to push the technical boundaries of this problem to their logical ends, but also to engage with the ways this technology can impact and potentially heighten user experience. More broadly speaking, this project is concerned with considering the impact technology may have in ambient settings and the ways we can enhance and augment a user's experience of video. Though our research never fully answers these questions, they are a guiding force in the direction of our technical research and development.

By situating our problem within the confines of live video footage, we are forced to use more optimized solutions that minimize latency to allow for multiple levels of video manipulation in real time. For example, the use of general purpose GPU programming to correct for barrel distortion in a wide angle camera lens improved our theoretical frame rate by over one hundred times.

In our efforts to maximize realism, technologies used include a high definition network video camera and the second generation Microsoft Kinect.

In the simplest terms, our technology works as follows: a user's position is tracked by an overhead sensor. This data is used to zoom and pan feed from a networked camera to simulate the viewing angles and experience afforded by a real window.

2. Methods

From the outset, the foundations of our project were based on providing an experience that not only maintained a high degree of realism, but also one that was intuitively and aesthetically pleasant. This means that our project was driven not by the intention to explore a specific technology, but rather an ongoing experiment in comparing the efficacy of a wide variety of implementation methods.

2.1. Materials

Before providing a more in depth look at these methods, we will first give an overview of the various technologies and platforms used and why they were chosen for this project. From the outset, we knew Kinect would be our method of gathering user position data. In addition to being a relatively standard technology for applications like this, the Kinect provides three-dimensional data in an easily processed image format, with the x and y coordinates along the image's x and y axes, and the depth values cleverly integrated into the pixel colors. By mounting this device on the ceiling, we are able to forgo more intensive face and body tracking in favor of simply tracking the uppermost part of a user's body--the head. We further elected to use the most recent Kinect V2, a decision that significantly reduced noise in our depth images.

For our language and development platform, we opted to use Processing. Aside from having powerful native image processing capability and seamless integration with OpenGL, the language's base in Java opened up the possibility of integrating a variety of Java methods.

As for our video source, the Axis HD IP camera was chosen for two main reasons. From an installation perspective, a networked camera allows increased freedom; the camera can run without the need for a dedicated machine or a wired connection to the PC running the virtual window software. Beyond this, the camera's high resolution means that even after zooming and other adjustments, the onscreen image will still be of acceptable quality.

2.2. Implementation

At the outset of this project, there was one visible problem that needed to be addressed before moving forward. This was the issue of correcting for the barrel distortion provided by our wide field of view lens. Though the wide field of view was ideal as a video source that could simulate the viewing angles of a window, the image's distortion greatly reduced the believability of the program. In retrospect, our research into barrel correction was beneficial not only because of the interesting results produced, but also because it drove forward an ongoing philosophy of exploratory, problem-solving based research.

Initially, we attempted to remedy the barrel distortion with a simple iterative barrel correction algorithm based on pseudocode found on the web[1]. For each drawn frame, the algorithm calculated which pixel from the distorted image should be placed at each location onscreen. While this effectively removed the distortion, it lowered our frame rate to unusable levels, with the image only updating once perhaps a couple of times per second.

Seeking better results, we instead calculated the pixel displacement values in advance, using the same algorithm as above, and stored them in an array. The replacement pixel values are then called from the array without the need for calculation. While this seemed to improve the frame rate slightly, it was still more or less unusable. Moreover, this low of a frame rate would not bode well for the intensive head tracking and image manipulation that lay ahead.

From here, we knew that this would need to be handled with off-CPU graphics shaders. By mimicking the approaches used on the CPU, we sought to create good comparison points when analyzing frame rate data later on. The iterative approach was implemented similarly in OpenGL, with a further addition of value interpolation to create a smoother image. For the pre calculated approach, we stored the desired pixel positions into an image file and loaded them as a texture on top of the original video feed.

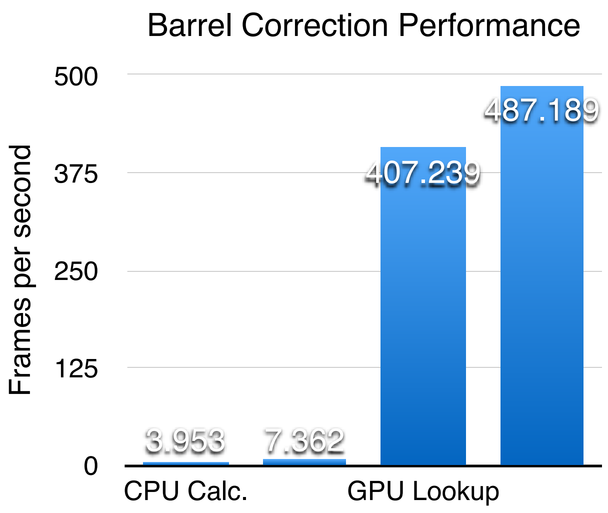
Our most early and naive approach at gathering head position data simply located the largest object in our depth image above an arbitrary threshold. The object detection was done by finding contours using the OpenCV computer vision interface. We calculated the average of these points to find an approximate head center point in the X and Y planes (that is, distance from the screen and location along the horizontal plane). These X and Y values, in turn, were used to pull head depth Z values from the Kinect's depth image.

An early and simple implementation method that greatly improved the usability of our program was the averaging of head position data. First, head X and Y centerpoints were calculated as an average of the past fifteen frames. Once this was established as the centerpoint, it was used to fetch the head

[1] See Helland, Tanner.

depth Z value. This depth position was averaged alongside all pixels touching its sides and diagonals (nine pixels in total). The average used for Z position drawing was also averaged over the fifteen most recent iterations.

While the concept of a dynamically changing thresholds is not new, we believe our approach to dynamic thresholding based on head positioning truly is novel.[2] The dynamic thresholding works as follows; before it calculates the head's center as described above, the program finds a rough centerpoint and uses it to get an approximate depth value for the top of the user's head. This is now used to set a new threshold a few inches below the user's head. By doing this, we are able to reduce camera noise from the surrounding environment and ensure that the OpenCV contour includes as little of the torso and shoulders as possible. When no user is present, the threshold returns to a much lower one to easily recognize a new user.



While the above methods made great strides in producing reliable and stable head position data, they took a major toll on performance. In the hopes of remedying this, we integrated multithreading into our program. Simply put, all head position calculation was offloaded to a separate thread. The main Processing draw loop could then call these values as needed without waiting for the calculations to be updated every time the loop ran.

With this data calculated, onscreen zoom was calculated as a factor of the user's Y position, and panning was adjusted in direct proportion to the user's X and Z position.

2.3. Evaluation

Our evaluation is, for the most part, based on the effects of various implementations on framerate. In our first set of data, we look at the effects of various barrel correction implementations on framerate. Though Processing provides a frameRate() function that outputs an approximate framerate, we found manually averaging iterations of the draw cycle over time to be more accurate and reliable. To this end, implementations of barrel correction were tested by iterating a counter for each iteration of Processing's draw loop. The program was run for approximately two minutes, and frame rate was calculated by dividing the counter over the amount of time the program ran (calculated based on timestamps taken at the program's start and close).

The approach for assessing multithreading performance took a very similar tack to the barrel correction mentioned above, with one minor difference. Though both our single and multi thread programs used the very same framerate calculation as mentioned above, the performance of our separately spawned calculation thread was quantified slightly differently. Instead of iterating for every iteration of the draw loop, the counter variable iterated for every iteration of the position calculation thread. Averages over time, though, were ultimately calculated in the same way. By calculating the framerate provided from both implementations' draw loop's, as well as calculating the iterations per second of our multithreading position calculation function, we hoped to discover the degree to which position calculation

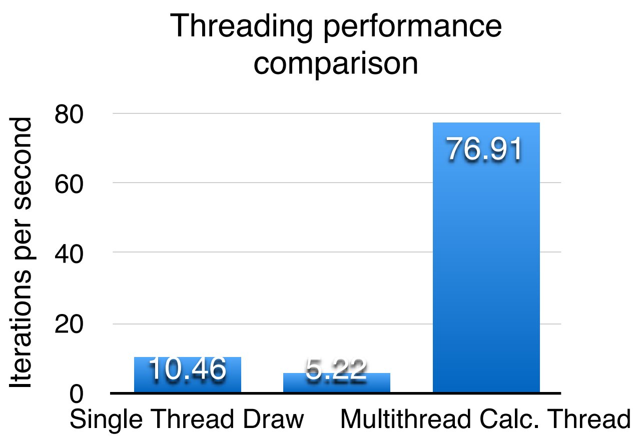
[2] See Lai for an application of dynamic thresholding that improves cell phone location tracking.

bottlenecked the performance of our program.

3. Results

3.1. Conclusion

The two data sets collected in our research, illustrated to the right and above, were completely polarized relative to our expectations. When looking at the barrel correction data, our expectations were almost exactly met. Offloading barrel correction to the GPU improved our performance by around one hundred times. Though it was slightly surprising that calculation allowed higher frames per second than lookup on the GPU side, these differences are negligible compared to the overall leap in performance between CPU and GPU calculations.



On the other hand, our data gathered on the effects of multithreading on program performance is completely at odds with our expectations. We anticipated a higher framerate in our multithreading implementation's draw loop, showing that position calculation was the major bottleneck in our single threaded implementation. Instead, the single thread implementation actually has a higher framerate than the multithreaded version. Further, the multithreaded calculation loop's iterations per second were around ten times faster than either version's draw loop.

3.2. Discussion and Future Work

Though we accomplished much of what we set out to in developing this project, there is still potential for future research. First, there is the problem of increased head tracking frame rate performance. Though we were able to get reliable head position data, it took a major toll on the program's frame rate. While this increased performance will likely include optimization in terms of added functionality, there are also elements of the head tracking as implemented whose effects on frame rate performance have not yet been quantified. In this paper we looked at the ways our position tracking was a bottleneck, and the degree to which multithreading was able to remedy it. Future research might dive deeper into this problem, seeking to better explain our current data and improve future results.

This problem of finding a multithreading implementation that does not reduce position data quality relates to a second point, quantifying the effects that our existing implementation have not just on performance but also on quality of data. Though it seems apparent that dynamic thresholding and averaging of frames improve our head positioning, we have not yet empirically tested them.

Beyond issues of performance, the overarching goals of this project call for a probe into the ways users can seamlessly interact with this technology. This means that calculation of viewing angles and user selection should be considered specifically from a user-centric perspective, and likely assessed with a user study. Beyond this, user studies may consider the software from an emotional and cognitive perspective, assessing how it affects users and how it compares to a real window or other ambient signage and displays.

References

1. Helland, Tanner. "A Simple Algorithm for Correcting Lens Distortion." Tanner Helland. N.p., 10 Feb. 2013. Web.

2. Ijsselsteijn, Wijnand, Oosting, Willem, Vogels, Ingrid, de Kort, Yvonne, and van Loenen, Evert. “A Room with a Cue: The Efficacy of Motion Parallax, Occlusion, and Blue in Creating a Virtual Window” Presence 17.3 (2008): 269-282. Web. 10 Nov 2014.

3. Lai, Yuan-Chen, Jian-Wei Lin, Yi-Hsuan Yeh, Ching-Neng Lai, and Hui-Chuan Weng. "A Tracking System Using Location Prediction and Dynamic Threshold for Minimizing SMS Delivery." Journal of Communications and Networks 15.1 (2013): 54-60. IEEE Explore. Web. 5 May 2015.

4. Merritt, John. “Virtual window viewing geometry” SPIE 1003 (1988): 386-392. Web. 10 Nov 2014.

5. Offenhuber, Dietmar. “The Invisible Display – Design Strategies for Ambient Media in the Urban Context”. Web. 10 Nov 2014.