Parallelism for Performance in AR Applications

Xuan Gu, Christiaan A. D’H. Gough and Mark Billinghamurst
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Abstract
Augmented Reality (AR) applications are highly performance critical applications. Because they insert themselves between the user’s senses and the real world, it is crucial that they run as fast as possible to provide maximum Realism. This requirement for high performance is offset by the fact that AR Applications are mathematically complex, making them inherently slow. Because AR applications generally form a pipeline of fairly distinct stages, they make good candidates for parallelisation. It should therefore be possible to maximise performance of AR applications by segregating these stages, forming a “waterfall” rather than a “pipeline” for the data to flow through. In this paper a multi-threaded AR application is implemented using the AR Toolkit, which allows each of the stages to attain their best possible performance and thus results in far greater overall performance and performance potential.

Keywords: Augmented Reality (AR), AR Toolkit, Performance, Parallelism, Multithreaded

1 Introduction
Most AR applications generally take a similar conceptual form a “pipeline” of several stages. The first stage captures the video to augment, usually from a live video feed from a camera, but sometimes from pre-recorded video files or other sources. The second stage analyses the captured video for features to augment. The third stage generates the augmentations and overlays them on the video, and the final stage displays this augmented video. A simplification of this pipeline can be seen from the Figure 1, and it is this general pipeline that is referred to throughout this paper.

![Image](image.png)

Figure 1: The general AR pipeline

This paper addresses the potential performance benefits of parallelising the general AR pipeline, describes an implementation of a sample parallel AR application and, finally, presents the performance of the sample AR application before and after parallelisation.

1.1 Classifying Latency
In Jacobs et al [1] paper, they address the problem of relative spatial and temporal registration, or the alignment of synthetic imagery with the real world, in AR systems. Correct relative registration is very important for conveying a convincing and realistic experience to the user, in much the same way that it is important that the sound and video in a movie are temporally registered or aligned correctly.

In order to identify the exact nature of misregistration, the authors classify 6 sources of latency: off-host delay, computational delay, rendering delay, display delay, synchronisation delay and frame-rate-induced delay.

Relative latency has its source in off-host delay, computational delay and synchronisation delay. The data from separate devices follow different paths in the system and each path has its own latency. The relative latency between these paths is what causes misregistration. For this reason, rendering delay and display delay do not cause misregistration, as they are common to all paths. Figure 2 shows where these delays are generated in the general AR pipeline of an AR toolkit application.
The methods of minimising misregistration are not as relevant to our paper as the classifications of latency sources and their classification in terms of relative latency are - although the techniques discussed for minimising relative latency could be implemented with the extra power gained by our implementation.

2 The Proposed Approach

2.1 Concept

As mentioned in the Section 1, AR applications are generally comprised of a pipeline of stages in sequence, and if one of these stages is significantly more expensive than the others it results in unnecessary performance penalties for the other stages. It is clear that decoupling the stages so that they form a sequence of "waterfalls" rather than a pipeline, each stage would be free to execute at its maximum potential.

Informal investigations show that applications are often hindered by the relatively long time it takes to capture a video frame, so that the performance of the entire application is bound to the refresh rate of the camera used. By changing the rate at which the camera captures images, it is possible to change the overall rate of the application, indicating a strong reliance of the pipeline on the capture stage. The time for the other tasks waiting for the camera to finish is effectively wasted, as the CPU is idle during the period between successive frame captures from the camera.

A consequence of having all the stages running in a sequence is that delays will be accumulated. Because of this, if it is possible to have these processes running concurrently, the total execution time will no longer be the sum of the execution time for each stage of the pipeline. This means that the other stages of the pipeline would not be left idle because of disproportionately high latency in one stage, eliminating an unnecessary performance cost.

Although these processes are not entirely distinct from each other due to their dependence on the output of the preceding stages, they can still be decoupled and run separately. Each stage can keep a copy of the last output generated by the previous stage and use that as input if new data has not been provided yet. In this way, each stage can operate at its full potential. While this may appear to be merely an intellectual performance gain, it has potential benefits if the bottleneck in the general AR pipeline really is the capture stage, as it means far more complex recognition, and display stages could be implemented without any performance penalty.

The use of parallelization would not only increase the application performance and minimize latency, but would also allow for the exploitation of multiprocessor systems or even multiple connected computers across a network, allowing even greater complexity in individual stages. For example, more accurate marker recognition or system registration algorithms, and more complex graphical scenes could be implemented if they did not have to "fit inside" the window of time created between updates of the camera.

This is an important benefit, as applications such as games [2] - which need highly detailed and compelling graphics, or medical applications [3] - which require far more accurate and therefore complex recognition stages - could be implemented. Other applications, such as the PDA approach in [4], would also benefit from this approach.

2.2 Parallelizing the AR Pipeline

Because, for most simple AR applications, the video capture is the slowest process, it makes an ideal candidate for the stage to initially attempt to segregate it from the others. Such a segregation would be possible if the data produced by the capture stage could be managed in a way that
allowed the other stages to access the latest output. Therefore, a “manager” is needed, to supervise the flow of data between the two “worker” stages. Figure 4 illustrates the relationship between the manager and the worker threads.

![Figure 4: Structure of the 2-threaded implementation](image)

# 3 Implementation

The AR Toolkit is a widely-used open source Application Programming Interface (API) library for the development of AR applications. The AR Toolkit implements the general AR pipeline; therefore, it inherits the performance issue mentioned in the previous sections. In order to demonstrate how the parallelization technique is applied to the AR applications, a case study is carried out.

In a standard AR Toolkit application, each stage of the general AR pipeline is carried out by a series of AR Toolkit build-in functions. “arVideoGetImage()” is used for capturing data from the camera; “arDetectMarker()” performs the marker recognition; “arVideoCapNext()” creates another thread which captures a new image and stores it in the background buffer; “argDispImage()” displays the captured image; “drawVirtualObject()” draws the augmented virtual scenes on top of the actual captured image; finally the “argSwapBuffers()” flips the background buffer with the current one so that the captured images are able to be displayed continuously and smoothly.

## 3.1 A Dual Threaded Implementation

Following the concept described in Section 2.2, the video capturing stage was segregated. To do this, both the “arVideoGetImage()” and “arVideoCapNext()” functions were removed from the main loop and put into another thread.

The reason why the capture stage was implemented as a separate thread rather than the other stages was because the other stages use the functions “argDispImage()” and “argSwapBuffers()”. Both of these functions are extensions of GLUT framework and therefore they must stay inside the OpenGL main loop in order to be executed successfully. This forms a good conceptual split, as shown below:

![Figure 5: Conceptual diagram of 2 threaded implementation](image)

The above diagram shows the two distinct conceptual threads that comprise an implementation of above concept using the AR Toolkit. The leftmost of the two loops represents the “capture” thread. It captures an image from the camera using the “argDispImage()”, and then advises the manager to “flip” the image buffer pointers. It then immediately begins another loop of the thread.

The right loop contains the recognition and display stages. The display stage is entirely contained within the GLUT framework. Each iteration, the recognition stage asks the manager for the pointer to the latest image buffer made available by the capture thread. This image will be displayed by calling function “argDispImage()” before the stages are executed.

Once the recognition stage has determined the location and the orientation of any feature to augment, it passes that information to the display stage by calling “drawVirtualObject()”. This draws the scene to display into the backbuffer, which is then flipped to the front using “argSwapBuffers()”. The repetition of this process is what makes an apparently animated view for the user.

The manager maintains two pointers, each of which points to a pre-allocated memory space containing a captured video frame. The duty of the manager is to provide the correct pointer to each worker thread on request. It also provides the concurrency control so that the pointers shared between these two threads will not be accessed at the same time. This prevents deadlocks and race conditions from occurring.

When the “capture” thread starts, it asks for an empty memory space for storage. Once a video frame is successfully captured and stored in the memory space provided, the manager switches of “flips” the pointers, so that the “capture” buffer now becomes the “recognition” buffer.
The recognition thread repeatedly asks the manager for the most up-to-date data captured. In this way, it is able to process and display data independently of the capturing process.

3.2 A Triple Threaded Implementation

Because the recognition stage has no reliance on the GLUT framework, it too could be implemented as a distinct thread. This would be a fully parallel implementation of the general AR pipeline defined earlier.

![Figure 6: Structure of the 3-threaded implementation](image)

The video capture thread captures the image data from the camera and passes it to the manager, which allows two other threads to access the image data immediately when they need. As the same idea, the information of the marker’s position and orientation calculated from the second thread is passed to the central manager and then accessed by the third thread which draws the overlay virtual scenes and display it on the screen. The information passed between any of these two threads is protected using mutexes.

3.3 Other Possible Implementations

An interesting implication of modifying the architecture of the AR Toolkit in such a way that the stages are “decoupled” is that it may be possible to implement each of the stages as distinct processes, rather than threads within a single process. This would allow for “scaling out” to other machines as well as “scaling up” within a single machine. This would allow for even greater scope in terms of possible complexity of recognition and rendering stages, and solutions such as augmentation of streaming video from webcams in remote locations would become seamlessly viable. It remains to be seen, however, if the bandwidth between machines on a network would be sufficient for this to be viable - although logic suggests that at least the video capture stage could be viable, as streaming webcams of reasonable quality are already a commercial reality.

4 Results

4.1 Testing on a single CPU machine

The results show a substantial improvement of performance. A simple sample application ran at just below the camera refresh rate - between 25 and 30 FPS, whereas for the same camera settings, the dual threaded implementation ran at almost 2 times that - around 60 FPS, despite the fact that a more complex recognition algorithm is used and more complex scene is rendered. The triple threaded implementation yielded even more impressive results. All measurements were taken on a Pentium Centrino 1.6 with 512 Megs of RAM and a Radeon Mobility 9200 GPU. Figure 7 and Figure 8 show the resulting data collected.

![Figure 7: Performance of various implementations of an AR application running on a single CPU with camera input of 15 FPS](image)

![Figure 8: Performance of various implementations of an AR application running on a single CPU with camera input of 30 FPS](image)

---

1 This technique is described in another paper prepared by the same authors: Improving the Robustness of Marker Detection in the AR Toolkit

2 A scene that contains five stacked rotating wireframe teapots. Wire frame scenes are far more costly than other scenes, especially in the case of highly tessellated geometry such as teapot primitives.
4.2 Testing on a dual CPU machine

The same procedure was repeated when performing the test on dual Xeon CPU machine. Each processor was clocked at 2.8 GHz and the machine was equipped with 1 Gig of RAM and a Radeon 9800 Pro graphics card. Figure 9 and Figure 10 show the resulting data collected.

Figure 9: Performance of various implementations of an AR application on a dual CPU machine, with camera input of 15 FPS

Figure 10: Performance of various implementations of an AR application on a dual CPU machine, with camera input of 30 FPS

4.3 Summary

The results demonstrate that our implementation allows for far more convincing AR applications, through much better utilisation of available hardware. The primary benefit of this is the ability to make use of far more complex display code, and implementation of much more robust recognition code, without affecting overall performance. In addition, it allows for development of other aspects. For instance, our implementations make use of an image smoothing technique to improve the quality of video from the camera.

This could have large benefits in terms of research, as it means that people can try out implementations of potentially expensive ideas without them affecting performance. A valuable improvement and area for future work would be to define interfaces or abstract classes for each stage. In this way, it would be trivial to simply define a concrete implementation of this abstract definition, making development of alternative recognition or display stages fast and easy. This would provide a system like modern graphics, whereby the developer uses the standard pipeline by default, but can easily override it and define their own for advanced purposes such as non-photorealistic rendering and researching new lighting models.

5 Conclusion and Future Work

This paper set out to explore the possibilities of parallelising AR applications in order to allow each stage of the overall process to operate to its best potential, and to prevent any one stage bottlenecks the whole pipeline. This paper describes the implementations of both 2 and 3 threaded approaches, and the results show the parallelism is effective in allowing AR applications to make the most of the hardware that they run on. This is an important achievement, because it means that far more complex recognition and display stages could be implemented without affecting the performance of the whole application.

Parallelism also means that multiprocessor systems could be fully exploited to further raise the possible complexity of AR applications, as each stage could run on a different CPU. This is an especially relevant improvement, because of the current market focus on "hyper threading" [5] and SMP. The segregation of the stages could also be easily implemented in such a way as to allow the stages to be executed on entirely separate machines on a network, thus allowing "scale out", rather than just "scale up".

It is important to note that parallelism does not accelerate the underlying algorithms; it merely allows the execution of the application in such a way that more complex algorithms can be employed without their performance costs impeding overall performance too much. Because of this, the key benefit of such an approach is that more complex algorithms can be used. It also allows for new possibilities, such as iterative recognition algorithms, which could perform several passes on a captured image before the next frame is captured.

The approach is not without some problems, however. Since the video frame may be updated much slower than the display stage updates, a small degree of "jerkiness" and image mismatch is perceptible. Using some of the newly available "headroom" in the recognition and display thread to implement movement prediction and image smoothing algorithms would overcome this problem, but still result in a net gain of performance - with far greater responsiveness and quality.
6 Acknowledgements

The work of this project has been shared evenly between us. For convention, our names are listed in the alphabetic order at the beginning of this paper. We would also like to thank Mark Billinghurst, who is our lecturer, for providing valuable suggestions.

References


