GPU-Accelerated Tools for Medical Image Registration and Biomechanical Modeling

Grant Lloyd Schunemann Marchelli

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2014

Reading Committee:
Duane Storti, Co-Chair
Mark Ganter, Co-Chair
William Ledoux
David Haynor

Program Authorized to Offer Degree:
Department of Mechanical Engineering

University of Washington

Abstract

GPU-Accelerated Tools for Medical Image Registration and Biomechanical Modeling

Grant Lloyd Schunemann Marchelli

Co-Chairs of the Supervisory Committee:
Professor Duane W. Storti and Professor Mark A. Ganter
Department of Mechanical Engineering

This dissertation explores the implications of GPU-accelerated computing (i.e., harnessing the parallel computing power of modern graphics processing units) for applications in image registration and biomechanical modeling. The work described herein includes the development of two software toolkits: one that implements functions for anatomical modeling based on distance fields, and one for registration of 3D (volumetric) medical imaging with 2D imaging (including stereo pairs).

In both cases, the general aim of the research effort was to provide practical support for creating and simulating anatomical models with greater accuracy and efficiency; the specific goals involved applications of interest of a research group focused on understanding the biomechanics of the human foot. Thus, the distance-based tools were employed for purposes such as enhancing an existing finite-element model of the foot by identifying appropriate locations for cartilage

elements that cannot be reliably resolved from typical imaging studies. The primary application described for the 2D-3D registration toolkit involves high-accuracy markerless tracking of the kinematics of the bones in the foot during walking gait.

In both cases, GPU-based parallelism was found to have a significant impact on software execution time for data-intense applications and succeeded in realizing gains in computational efficiency of 10- to 150-fold over traditional CPU-based computing. The ramifications of such a radical increase in raw computing power are many, but possibly one of the most important outcomes for the group is the ability to rapidly and accurately quantify bone motion in the human foot during gait. Moreover, this research will deepen our understanding of foot bone motion as it relates to subjects exhibiting both normal and abnormal (i.e., deformities) foot conditions. Previous attempts at engaging in studies involving a large subject pool were stifled by prohibitively long software execution times; however, the GPU-based image processing tools developed in this dissertation will enable our group to revisit the once impractical investigation.

While the human foot is the anatomical region of choice for studies described in this dissertation, the tools presented can be adapted to other anatomical localities in the body, such as the knee, hip, hand or shoulder. Additionally, this toolbox need not be limited to the human anatomy, opening the door to efficient exploration of animal models.

Simulation of anatomically correct models can help investigators to more accurately predict physiological function, diagnose disease, and analyze environmental impact, which may lead to higher success rates during treatment and recovery. The GPU-based tools presented in this dissertation will provide the foundation for future work in the field of computational biomechanics by arming investigators with resources that encompass particularly relevant

parallel computing tools. The intent is to provide an intuitive and interactive environment for efficiently modeling and manipulating highly complex biological geometry.

Table of Contents

Table of Figures

List of Tables

This dissertation is dedicated to my brother Weston – may you walk in my shoes, see through my eyes and experience the many joys of a life that you did not have. Carpe Diem.

Chapter 1: Introduction

The research discussed in this dissertation bridges the gap between data-intense processing and computational efficiency by redefining the problem space. The working hypothesis for this dissertation was centered on the idea that real-world computational problems in biomechanics – including soft tissue modeling and, in particular, 2D-3D markerless registration – can be approached using a parallelized computational mindset to realize efficiency gains of several orders of magnitude without sacrificing the requisite sub-millimeter and sub-degree precision. Furthermore, it was postulated that software toolkits could be developed in such a way that enabled GPU-based parallel processing without having the user interact with any GPU-specific programming.

The findings presented in this dissertation were achieved by leveraging a multidisciplinary education regime that included the study of geometry, kinematics, biomechanics and programming, to overcome the technical difficulties of developing and implementing two GPU-based software toolkits. In broader terms, the seamless integration of bio/mechanical engineering, mathematics and computer science facilitated the realization of the toolkits discussed in subsequent chapters. During the course of this research, a strong appreciation for and understanding of the aforementioned disciplines was cultivated and applied to solve several challenging problems faced by the engineering community. The following excerpts were chosen to provide the conceptual framework for which this dissertation was built upon.

"The new subject is Computational Geometry, which is based on the premise...that classical mathematics needs to be augmented in order for us to be able to solve geometric problems efficiently on a computer."

-M. I. Shamos, *Computational Geometry* [1, 2]

In essence, the utilization of computer technology to generate, view, manipulate, analyze, and in some cases, produce, mathematical shape data based on algorithms, is a simple, yet ambiguous definition for computational geometry [1]. Computational geometry can be thought of as a multidisciplinary study that integrates mathematics, engineering, computer science, and physics into one robust system – the computer. By encapsulating these many sciences into a device that has the capacity on one end, to emulate complex naturally occurring phenomena, and on the other, to project, solve, and analyze the humanly inconceivable, we are granted with a great power. Harnessing this power will lead to new innovations in many fields of academia and industry, including biomechanical modeling.

"Biomechanics is the study of the structure and function of biological systems by means of the methods of mechanics."

-H. Hatze, *What is Biomechanics* [3, 4]

"It seems to be one of the fundamental features of nature that fundamental physical laws are described in terms of a mathematical theory of great beauty and power…"

-P. Dirac, *The Evolution of the Physicist's Picture of Nature* [5]

Biomechanical modeling, in a general sense, can be loosely classified as the generation, visualization, and manipulation of biological structures for studying mechanical behavior and biological phenomena. For example, the bones of the human ankle can be characterized as a biomechanical system akin to a common door hinge. Analyzing the static, dynamic, and kinematic behavior of the ankle joint would fall under the category of biomechanics. However, by generating geometrical data derived from the bones and various tissues that comprise the joint, we are in turn modeling a biological structure. Modeling a biological structure to analyze

the mechanical properties, e.g., material behavior, kinematics, contact stresses, etc., is a modest definition for biomechanical modeling.

The intersection of computational geometry, biomechanics and modeling provides an exciting and stimulating field of study with endless potential: computational biomechanical modeling (Figure 1.1).

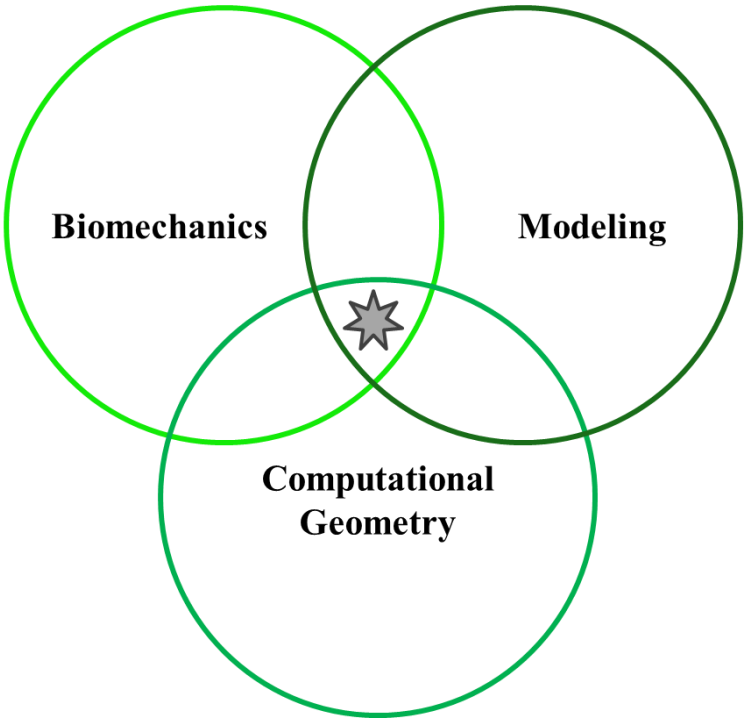


Figure 1.1. Computational biomechanical modeling. This simple Venn diagram illustrates the overlapping subejects of biomechanics, modeling and computational geometry. Their intersection, marked by the star, represents the extremely challenging and deeply fascinating study of computational biomechanical modeling.

1.1 Image registration

Medical image registration is progressing towards automated methods that enable efficient and accurate processing of volume data [6]. In 2D-3D registration, volumetric imaging data are digitally projected onto a 2D plane by sampling the density function of 3D CT scan data and approximating the integral of the densities along rays passing through the dataset. The resulting sum for each ray serves as the final corresponding pixel value or *intensity* in the projected 2D

image (see Figure 1.2). Once the projected 2D image – referred to as a Digitally Reconstructed Radiograph (DRR) – has been computed, it is compared to the fluoroscopic image via similarity measure using common image correlation techniques that are discussed in Chapter 5.

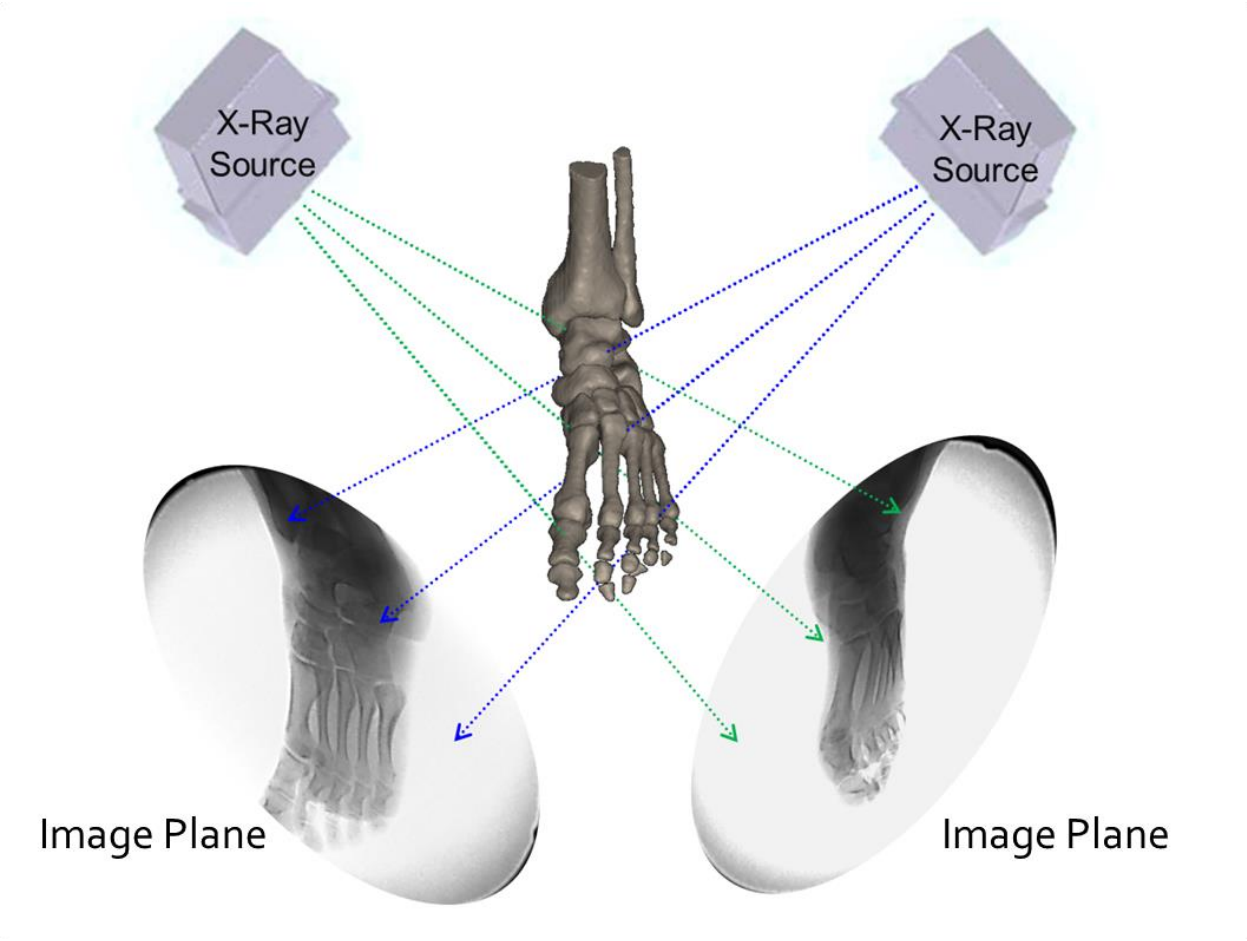


Figure 1.2. Stereoscopic fluoroscope system. An example of a stereoscopic fluoroscope system utilized in capturing still frames of a dynamic event, e.g., the foot as it moves through a gait cycle. The pair of image sequences are used to determine the kinematic motion of the bones in three-dimensions.

For 2D-3D registration using an X-ray system as the 2D imaging modality, 3D translations and rotations are found that optimally match the 2D X-ray images to the DRR [7]. These optimized transformation parameters provide the spatial information needed to match static imagery to dynamic observations – this allows for investigations ranging from the quantification of bone motion during complex anatomical movements [8] to image-guided therapy [9]. The two main

components in both 2D-3D intensity- and gradient-based registration are the construction of the projection image, i.e., the DRR, and the similarity measure [10, 11]. The latter enables quantitative comparison of the DRR with the true X-ray image. In general, the computational bottleneck in a 2D-3D registration algorithm is the calculation of the DRR, which typically requires integrating across millions of voxels to determine the image's individual pixel intensities [6, 7, 9, 10, 12-18].

There have been several attempts to decrease the time associated with DRR generation by using the graphics processing unit (GPU) [19] as the computational engine [9, 10, 12-14, 16-18]. (GPU computing will be discussed in Chapter 2.) LaRose was among the first to implement fast DRR computation on the GPU using 2D textures [14]. Kim et al. [16] and Ino et al. [18] built on the methods of LaRose [14], creating more accurate DRRs by extending his work to 3D textures. Spoerk et al.'s [13] earlier work utilized wobbled splat rendering on the GPU, but in later work, Spoerk et al. [9] highlight GPU-based ray casting as a more accurate technique. Ruijters et al. [12] took advantage of two intermediate z-buffers, which enable simultaneous input and output texture formatting. However, as discussed by Mori et al. [17], the previous works all require highly specialized training in computer graphics and programming, making it difficult to reproduce the results. More recently, Tornoi et al. discussed the use of general purpose computing on the GPU (GPGPU) to generate DRRs in parallel using 3D textures and ray casting. The methods presented by Tornoi et al. [20] were quite similar to those described in this dissertation; however Tornai et al. only considered DRR generation, not the complete registration process.

1.2 Biomechanical modeling

The search for an accurate and computationally inexpensive finite element (FE) model of anatomical regions in the human body for use in biomechanical evaluation is a topic of great interest in human-based physiological studies. Anatomically similar models can complement expensive cadaveric experiments, and offer invaluable insights into biomechanical applications including joint contact stress [21], bone-to-bone position [22-25], basic foot function [26], insole design [27, 28], surgical simulation [28], internal stress distribution [29] and patient-specific modeling [30].

In vivo, articular cartilage serves many functions, the most notable of which is its performance as a smooth, low-friction, wear-resistant surface that allows a pair of bones to effortlessly glide past one another during articulation [31]. The articular cartilage (or simply "cartilage" herein) also serves as a protective tissue that dampens vibrations during gait and other high-impact activities. For these many reasons, cartilage is an irreplaceable biological structure and should be considered when developing anatomically correct models.

In general, models of cartilage are created using labor-intensive manual segmentation of computed tomography scans [32] or magnetic resonance images of the desired biological structure [24, 25] (see Figure 1.3). Higher resolution cartilage segmentation is possible with micro computed tomography (μCT) scans [33] and some investigators have integrated laser scans with commercially available mesh generation software [34]. An approach based on filling the joint gap between the two bone models with a solid has been executed by our group; however, this method was far from anatomical [22, 23]. Extruded cartilage models based on surface projections of a rectilinear grid of points onto the bone [35] have led to promising results,

as have methods that identify joint contact surfaces by employing distance thresholding [36]. It is this last method that will be explored in this dissertation (Chapter 9).

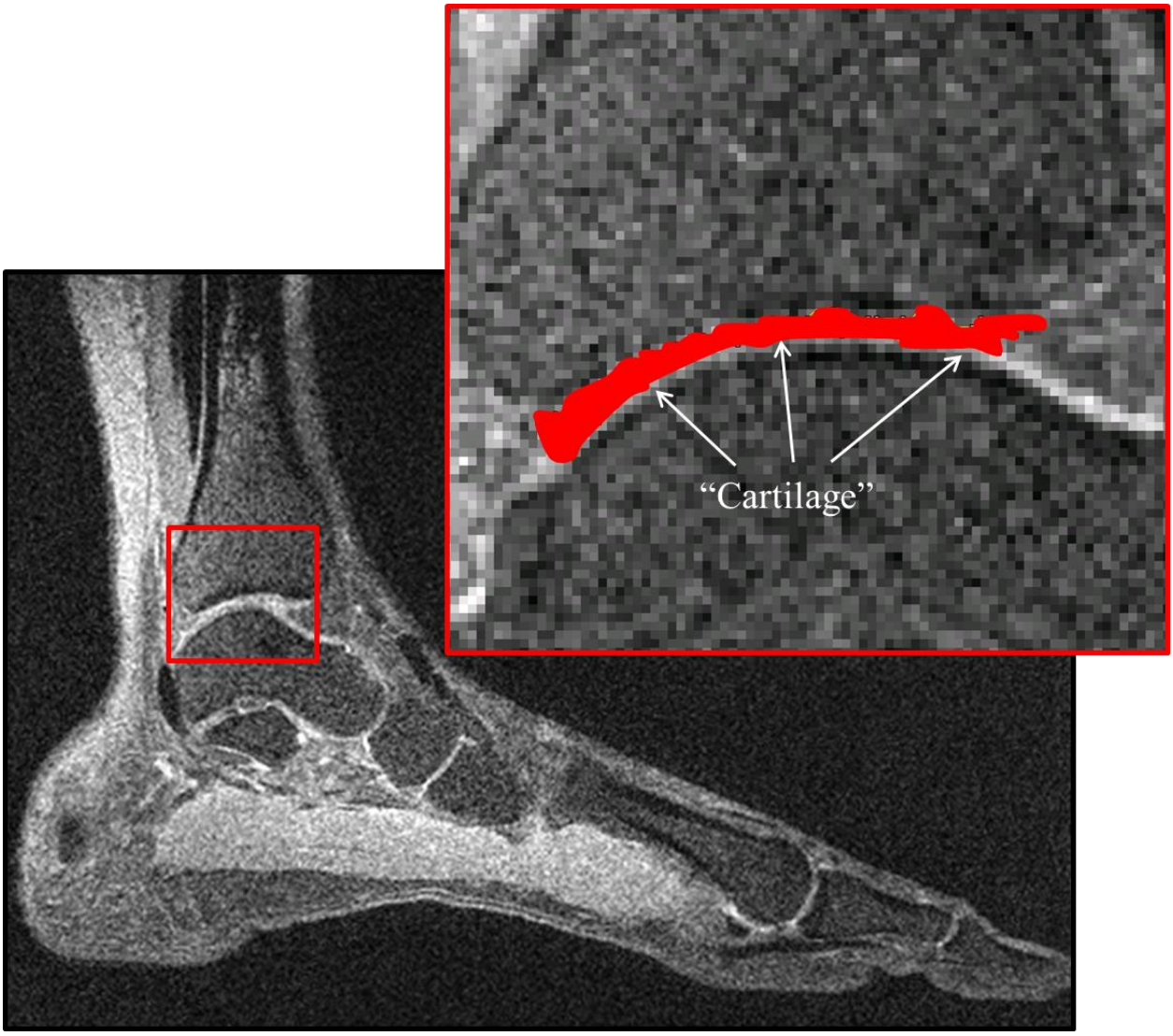


Figure 1.3. A typical image slice belonging to a CT scan of a human foot. The expanded view of the image contained within the red box quickly reveals the difficulties faced by researchers during manual segmentation of cartilage from bone and other soft tissues. The region in red being pointed to in the expanded view is the result of a true attempt at segmenting the cartilage from the bone, pixel-by-pixel, by a skilled researcher.

Automated approaches to modeling of cartilage (and other soft tissues) are needed to not only reduce user-bias, but also to increase overall computational efficiency for subsequent analyses. Nonetheless, many investigators in the field create models that focus more on joint function and

boney interactions than on soft tissue analysis [26-29, 37]. These methods may generate biomechanically realistic joint motion kinematics [38], but predicting accurate joint stresses would require anatomically-based soft tissue models. Furthermore, cartilage modeling generally relies on heavy user interaction, an inefficient and time consuming endeavor. Models are based on user distinction of bone and soft tissue and frequently are built on educated, but subjective, isolation of tissue in each slice of the scan. Misidentification of soft tissue versus bone results in rough and non-anatomical model estimations. Elimination of human bias can lead to models that more reliably represent true anatomical behavior.

1.3 Research Objectives

The primary objective for this research was to develop a deep understanding of GPU computing and to then study the feasibility of applying this programming paradigm to real-world problems in medical image registration and biomechanical modeling. Moreover, GPU computing was employed to determine whether such techniques could be used to drastically minimize image processing software duration of execution while maintaining or exceeding the precision realized by the CPU-based parent code. The applications revolved around 3D bone motion reconstruction and soft tissue modeling in the human foot.

An auxiliary objective was the development of a software toolkit that insulated the user from the details of the GPU programming language. This software architecture strategy provides the user with access to nearly 20 image processing functions, each of which taking advantage of the untapped computing power lying dormant on the GPU. The only requisite to successfully run the software is a cursory understanding of the C/C++ programming languages.

1.4 Dissertation roadmap

As alluded to in Sections 1.1 & 1.2 above, the work described in this dissertation is bifurcated into two distinct categories: medical image registration and biomechanical modeling. The former will comprise the bulk of this dissertation and works on this subject will be found in Chapters 3-8. Works relating to the latter, biomechanical modeling, will only appear in Chapter 9.

Chapter 2 will expose the reader to parallel computing on the graphics processing unit. This chapter is not considered a requisite for the remainder of the dissertation, but the reader should at the very least skim the chapter if not already familiar with GPU computing. Several terms and concepts such as GPU memory models will be introduced here and these ideas will be heavily referenced throughout.

Chapter 3 delves deeper into the challenges faced by researchers in the field of medical image registration. We describe the methods used for capturing data and introduce the reader to the two software programs used to process gait record data. A very high-level overview of the 2D-3D image registration problem is also touched upon in this chapter (i.e., Chapter 3).

Chapter 4 will serve to describe the approach used to achieve real-time image registration. This chapter will expose the inner workings of the software package DRRACC (**DRR**s **AC**celerated by **C**UDA) and its toolkit counterpart DRRACO (**DRR**s **A**ccelerated by **C**UDA **O**perations), the core of this research. We discuss programming language, toolkit development and data preprocessing, and finish strong with a thorough look at the methods employed for parallel DRR computation. Chapter 5 logically follows Chapter 4 by describing the approach for image correlation; DRRACC utilizes both image intensity and image gradients for correlation. The motivation for using a linear combination of the two correlation methods is also presented.

Chapter 6 tackles the challenge of visualization. The CUDA-OpenGL interoperability Application Programming Interface (API) is introduced and the real-time approach used to simultaneously display fluoroscopic comparison images with DRRs is visited. We also discuss the importance of implementing a Graphical User Interface (GUI) to facilitate accurate initial positioning and efficient debugging.

Chapter 7 is focused on the optimization suites available in DRRACC. Optimizers are used to efficiently test a variety of transformation parameters to quickly converge at the merit function optima. We briefly discuss two of the main types of optimizers: direct search and gradient-based. Two optimization suites that utilize the two differing approaches, CONDOR and SNLP, respectively, are introduced. Noise in the system is described as it relates to optimization complexity.

Chapter 8 applies the concepts in Chapter 2 through Chapter 7 to real imaging data in an effort to validate the efficacy of DRRACC. Image formation, image correctness, image linearity and image geometry are all validated through rigorous experimentation. Several data sets with varying degrees of intricacy are tested using DRRACC and the results are compared to those generated by a validated registration software suite. The chapter concludes with a performance review that pits the existing CPU software against DRRACC.

Chapter 9 diverges from the concepts discussed in Chapters Three through Chapter Eight to revisit biomechanical modeling of articular cartilage. The first few sections of the chapter are dedicated to introducing concepts such as point-to-triangle distance, distance fields, and geometric skeletal models. Next, a single-position study of the first metatarsophalangeal joint is presented, followed by a multi-position study. We show that joint range of motion (ROM) has a

significant effect on distance-based cartilage modeling. The chapter finishes with an investigation of joint capsule modeling.

Chapter 10 revisits the salient points of the research and provides recommendations for areas of future investigation.

The contributions of this dissertation include:

1. Development of parallelized computational tools for medical image registration.
2. Execution of a parallel visualization tool to assist with registration and troubleshooting.
3. The synthesis of two independent optimization suites with the parallelized image processing algorithms.
4. The integration of the algorithms (1-3 above) to construct the software toolkit DRRACC.
5. The development of a parallelized toolkit that automatically identifies patient-specific articular contact surfaces using joint-specific thresholds and distance fields.

Chapter 2: GPU Computing & CUDA

The work discussed in the subsequent chapters of this dissertation build upon the idea of GPU computing, that is, the programming paradigm shift away from sequential processing on the Central Processing Unit towards parallel processing on the Graphics Processing Unit. The computer industry is driven by consumers fervently demanding more computing power for each dollar they spend. The advent of multicore CPUs (e.g., dual-core, quad-core, etc.) in the early- to mid-2000s played an instrumental role in feeding this insatiable demand for faster, stronger and smaller personal computers [39]. However, the desire to garner even more power from computers beyond the parallel capabilities of multi-core CPUs sparked a revolutionary shift in the consumer computing industry: Enter GPU computing.

2.1 What is GPU Computing?

Before we can answer the question posed in the section title, we must first ask the question: *What is a GPU?* Traditionally, the Graphics Processing Unit (GPU) is the computational workhorse that rapidly generates and manipulates pixel memory for a computer's display. The GPU is comprised of many standalone calculators or *pixel shaders* that are extremely efficient at performing numerically-intense operations on floating point values and are designed execute in parallel. Programmers had limited access to the pixel shaders through Application Programming Interfaces (APIs) such as OpenGL [40, 41] and Microsoft's DirectX [42]. Development efforts on boosting GPU performance and efficiency was largely driven by the gaming industry in the mid-1990s, but the idea of using the GPU for general purpose computing did not coincide with industry goals for consumer products until 2001 when NVIDIA released the GeForce 3 [39, 43].

NVIDIA's GeForce 3 was the first consumer card that was in compliance with Microsoft's DirectX 8.0 standard, which opened up the doors to programmers looking to write scripts that

could take advantage of the GPU's pixel shader engines [43]. It was not long before programmers realized that the data going into pixel shaders was not restricted to the typical inputs that generated the pixel's color for output, however the convoluted programming language was not amenable to a broad audience of developers [39]. Unfortunately for researchers looking to reap the benefits of GPU computing *without* learning shader and computer graphics programming, it was not until late 2006 when NVIDIA finally responded with DirectX 10 GPU, the GeForce 8800 GTX and CUDA (formerly known as Compute Unified Device Architecture) [39, 43-46]. The motivation behind introducing CUDA C and the CUDA-compatible GeForce 8800 GTX was to address the complexities faced by researchers trying to harness the GPU for General Purpose Computing on the GPU or "GPGPU" for short [39].

2.2 Who Cares?

Almost immediately after NVIDIA's release of the CUDA architecture and corresponding CUDA C programming language, researchers across the globe began experimenting with the nearly untapped potential that GPU computing had to offer. Referring to the contents of this manuscript, it is clear that the medical imaging and biomechanical modeling industries are witnessing a massive shift towards efficient, data-intense processing. Moreover, applications for GPU computing have gravitated towards other markets and fields of study that also process large data including: computational fluid dynamics, finite element analysis, environmental science, gaming, solid modeling, dynamic system analysis [39, 43-45]…The list itself would comprise an entire chapter of this dissertation.

2.3 Introduction to CUDA C and Parallelism

The development of CUDA was driven by the demand for a more straightforward approach to accessing the GPU for general-purpose computing. NVIDIA answered this demand by releasing

the CUDA C software architecture, which is an extension of the well-established C programming language. CUDA C, however, separates itself from traditional C through the use of qualifiers and functions that execute either on the *host* (CPU) or on the *device* (GPU)[1]. It is worth noting that the concept of *host* and *device* will be heavily relied upon throughout this dissertation.

Now recall the primary benefits of enlisting the GPU to perform computations on data-intensive projects: The GPU is designed to efficiently perform arithmetic operations on data in parallel. *Why do we care about parallelization?* The answer is actually quite simple (and at this point, likely very obvious): Fast execution time. To solidify this notion, consider a shopping list that required you to visit a handful of different stores, with the assumption that all shopping tasks are independent of one another (e.g., purchasing item #2 at store #2 does not rely on purchasing item #1 at store #1). Your strategy for completing your shopping might involve planning the quickest route between sequential visits to storefronts. *What if you could visit all the stores on your list simultaneously? What if you could finish your shopping in the time it takes to visit a single store?* The ramifications of such ability would be astounding and would represent a paradigm shift in the way we as humans process information. Bringing the focus back to computers, parallel processing on the GPU embodies a paradigm shift from CPU programming.

The CUDA parallelism model relies on the strategic use of the GPU's *Streaming Multiprocessors* (SMs) and the architecture known as Single-Instruction, Multiple-Thread (SIMT) [39, 43-48]. SMs are the active portions of the GPU that create, manage, schedule and execute *warps*[2] during a CUDA *kernel* launch; the kernel provides a host-side link for executing

[1] For the remainder of this dissertation, *host* will refer to the CPU, and *device* will refer to the GPU (e.g., the host function was restructured to run in parallel on the device).
[2] A warp is comprised of 32 computational threads that execute in parallel.

code in parallel on the device [45]. Each SM is comprised of up to 192 *cores* (often referred to as *CUDA cores*), with the exact number of cores being specific to the GPU as defined by its *Compute Capability* [45]. The Compute Capability dictates which CUDA features are supported by a GPU and is used at runtime to check feature compatibility [45]. Table 2.1 provides the specifications – including the number of CUDA Cores – for the GPUs employed during the research discussed in this dissertation. A comprehensive table detailing the specific features associated with each Compute Capability can be found in Appendix A[3].

Table 2.1. The NVIDIA GPUs utilized in this research.

NVIDIA Card	*Architecture*	*Compute Capability*	*Number of CUDA Cores*	*Memory (GB)*	*Peak Performance (teraflops)***
GeForce GT 440	Fermi	2.1	96	1.5*	0.311
GeForce GT 640M	Kepler	3.0	384	2.0	0.480
GeForce GTX 460 v2	Fermi	2.1	336	1.0	1.045
Tesla C2050	Fermi	2.0	448	3.0	1.288
Tesla K20	Kepler	3.5	2496	5.0	3.520

* Depends on memory configuration (GDDR3 vs. GDDR5).

** Single-precision performance in teraflops (1×10^{12} floating point operations per second).

At this point it would be worthwhile to include a brief example demonstrating the concept of a kernel launch. Consider the following function prototype written in traditional C:

returnType exampleFunc(argType0 argument0, …, argTypeN argumentN);

Here, `returnType` is used to specify the data type that the function `exampleFunc()` will return upon completion. The *N* `argTypes` define the data type (e.g., `int`, `float`, `double`, etc.) for each of the *N* `arguments` being passed into `exampleFunc()`. Now consider the case where `exampleFunc()` is comprised of several nested loops:

[3] Additional details can be found here: www. http://en.wikipedia.org/wiki/CUDA.

```
1   exampleFunc(argument0, …, argumentN)
2   {
3         int index;
4         returnType output[iMax * jMax * kMax] = {};
5
6         for(int i = 0; i < iMax; i++)
7         {
8               for(int j = 0; j < jMax; j++)
9               {
10                    for(int k = 0; k < kMax; k++)
11                    {
12                          index = i + j * iMax + k * iMax * jMax;
13                          output[index] = i * j * k;
14                    }
15              }
16        }
17        return output;
18  }
```

In this example, the host must sequentially step through each of the `for()` loops (lines 6, 8 and 10), and for large values of `iMax, jMax` and `kMax`, this simple function becomes a computational bottleneck Each of the indices and return values (lines 12 & 13, respectively) are computed independent of one another, rendering this function quite amenable to parallel processing. Rather than incrementing indices sequentially like the CPU, the GPU has the capacity to launch a *grid* of *blocks* containing *threads*, each with unique CUDA IDs (`blockIdx` for blocks `and threadIdx` for threads), that can be assigned to compute each index and return value (lines 12 & 13) in parallel[4]. Converting this function into a kernel launch that will tell the compiler to execute it on the device would require the following modifications to the `exampleFunc()` prototype:

[4] The size of `iMax, jMax` and `kMax` would dictate whether this is actually possible. The limiting factors would include the number of Streaming Multiprocessors and the Compute Capability of the GPU enlisted for this computation (some of this information can be found in Appendix A, with all other being available here: www. http://en.wikipedia.org/wiki/CUDA).

```
__global__
void exampleFunc<<<grid, block>>>(argType0 argument0, …, argTypeN argumentN);
```

You will note three distinct changes to the `exampleFunc()` prototype: 1) prepending the prototype with the "`__global__`" qualifier, 2) the replacement of `returnType` with `void`, and 3) the addition of the triple-chevrons that encase the parameters ***`grid`*** and ***`block`***. (At this point it is also worth mentioning that `exampleFunc()` is now considered to be a kernel launch). The first change simply informs the compiler that `exampleFunc()` is to be executed on the device. The second change is where things start to get a little odd; the function return type is eliminated and replaced with `void`, meaning that `exampleFunc()` is no longer capable of returning a value. *If a device function cannot return a data type, what is the point of being able to compute in parallel?* This question will be answered shortly, but for now, we will focus on the third change to `exampleFunc()` – the addition of the triple-chevrons preceding the standard argument list and the two parameters within them.

The addition of the triple-chevrons and the two parameters – ***`grid`*** and ***`block`*** – are used at runtime by the SMs to specify how to launch the `exampleFunc()` kernel [39, 45]. Each CUDA core within an SM is capable of managing a single grid, which is comprised of *thread blocks* (the number of thread blocks is set using the *grid* parameter) [45]. The thread blocks (or "blocks") are each capable of employing *computational threads* – as specified by the *block* parameter – which are considered to be the computational workhorses of the GPU [45]. Recall the earlier discussion of pixel shaders; in the CUDA architecture, computational threads are synonymous with pixel shaders [49]. Figure 2.1 provides a simplified schematic of the CUDA grid-block-thread relation. Additional details on this topic can be found in the literature [39, 43-49].

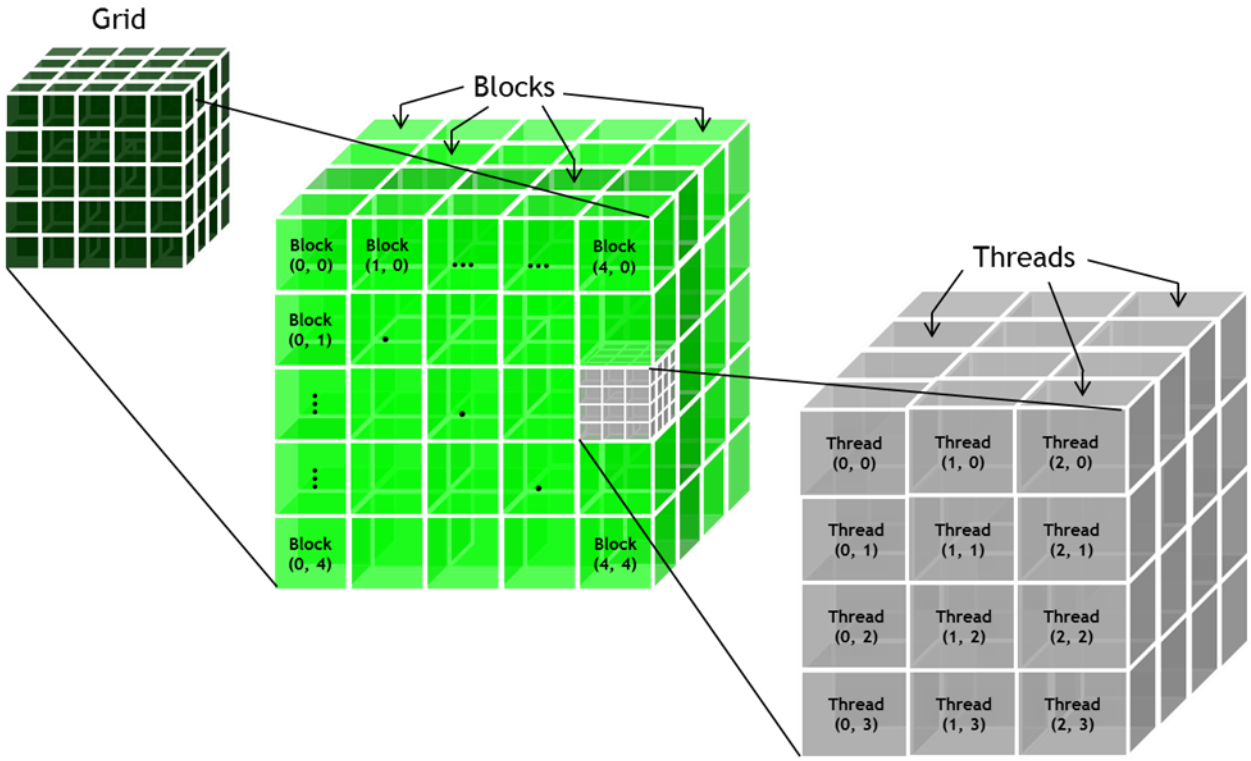


Figure 2.1. The CUDA grid-block-thread model.

One extremely utilitarian feature that NVIDIA imparted on the CUDA architecture is the ability to automatically scale parallelism (Figure 2.2) with the number of SMs on a given card [45]. This radical feature allows the user exploit parallel processing on any CUDA-capable device with minimal changes to the program. (Note that the specific number of blocks and threads for each kernel launch within a program should be considered for performance implications when switching between GPUs, especially if different Compute Capabilities are involved.) Now that we have outlined the basic CUDA parallelism model, it is time to move onto the challenging aspect of GPU-computing; memory.

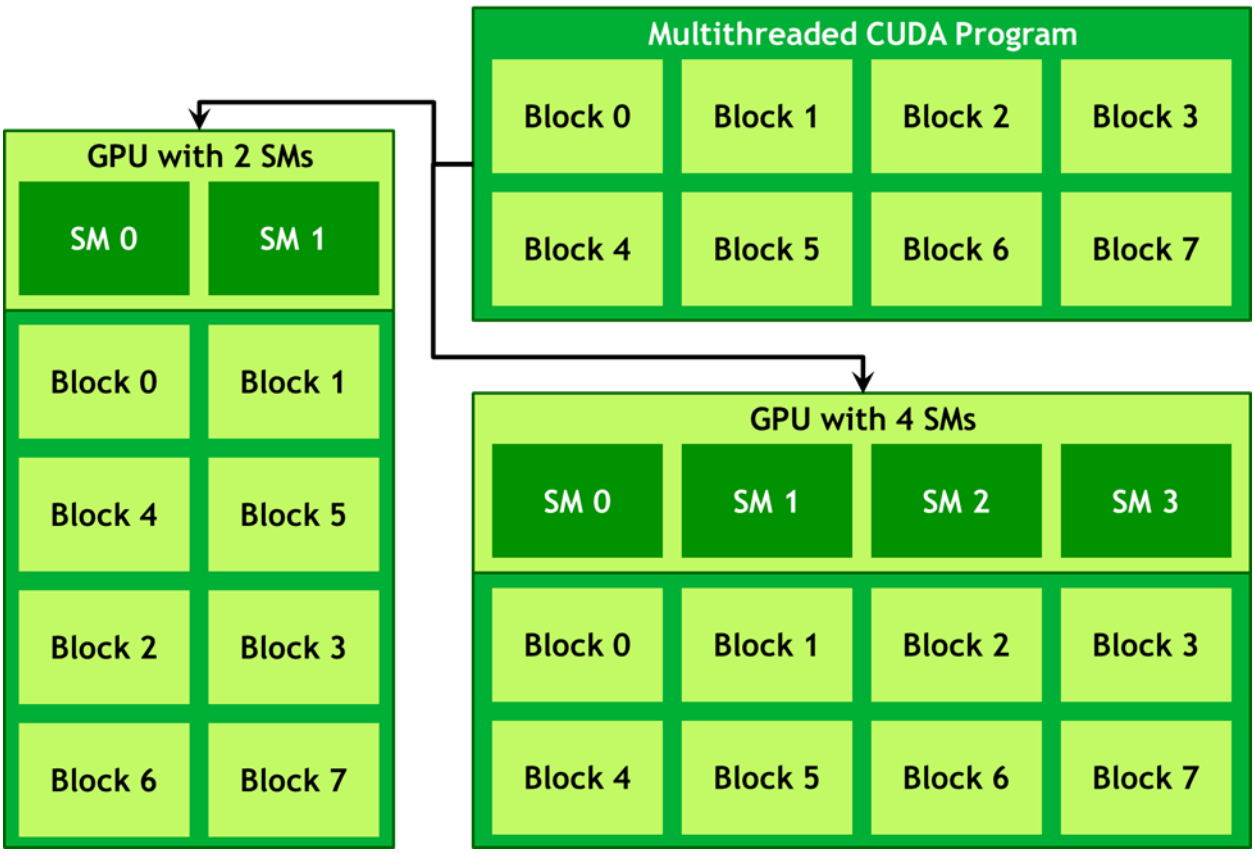


Figure 2.2. Automatic scalability of CUDA parallelism (recreated from [45]).

2.4 CUDA Memory Model

As we discussed in the preceding section, the CUDA architecture enables the user to access both host and device functions, but what we intentionally left out was the mechanism for actually accessing the data from within a device function. Device functions cannot explicitly operate on host-side memory, and likewise, host functions cannot operate on device-side memory. When possible, arguments can be copied into the *registers* at kernel launch as discussed in a subsequent section. If the data cannot rely on the *registers* for device access, a device function can only operate on given set of data that has been copied from host-side memory to device-side memory. The CUDA architecture provides a series of functions that support host-to-device, device-to-host, and device-to-device memory transfers.

Recall that kernels are incapable of returning **any** data type and let us revisit the question: *If a device function cannot return a data type, what is the point of being able to compute in parallel?* The answer lies with the CUDA memory model; memory for an output container is allocated on the device prior to launching a kernel. A pointer to that memory location is then passed via the registers, where it can be accessed and operated on by computational threads. Once the kernel has finished executing, the user can invoke a device-to-host memory transfer to retrieve the data stored in device-side memory, so that it can be accessed by the host (e.g., used by host functions, written to a file, etc.). The following sections will be used to detail the primary types of memory used in CUDA-based GPU computing.

2.4.1 Atomics

While not technically a memory type, the implications of *atomics* with respect to device-side memory is too great to not warrant a description in this section. Atomics possess the special ability to update memory locations that are being operated on by more than a single computational thread [46]. Because thread execution order is random, atomics are generally utilized for reduction operations such as addition or subtraction [46].

2.4.2 Registers

Data located on the host-side that is destined for use on the device-side must first be copied from host-side memory to device-side memory by invoking a built-in CUDA function. Once the data is located in device-side memory, a pointer to the memory location of the data can be passed to the device function via *registers* during the kernel launch (a single integer or floating point number can be passed without explicit memory transfer) [46]. The registers act as an express conduit for copying data to device-side memory at kernel launch, giving each of the computational threads unhindered access for the life of the kernel launch [47]. The registers are

the fastest, but not necessarily the most efficient, means for device functions to access host-side data [46].

2.4.3 Global Memory

On the other end of the data access efficiency spectrum is *global memory*, which is the least restrictive, but also the slowest of the memory types, with relative access times 100x slower than accessing data that is copied using the registers. Pointers to data stored in global memory are defined on the host using a built-in CUDA function and are accessible by device functions without the need for explicitly passing the pointers into the kernel by way of registers. Global memory is generally most beneficial when data need to persist throughout the program's life [48]

2.4.4 Local Memory

Kernel launches have the ability to allocate device-side global memory on an as-needed basis from within the function. This type of memory is referred to as *local memory* and behaves similarly to its host-side counterpart; memory allocated within the kernel only persists for the life of the kernel, or more precisely, for the life of the thread using it. The memory itself is located off-chip physically in device Dynamic Random-Access Memory (DRAM) and relative memory access times are identical to global memory [47].

2.4.5 Shared Memory

One of the faster memory access types is known as *shared memory*, which is located on-chip and behaves like a user-managed cache [48]. Data contained in shared memory can be efficiently accessed by all computational threads in a given **block**. The speed, however, carries with it a limitation in terms of memory capacity; a low ceiling of 64KB per SM[5] generally restricts the

[5] 64KB per SM is restricted to cards with a Compute Capability of 5.0 or higher. Appendix A provides a more complete list of features and limitations specific to each Compute Capability.

functionality of this memory type to less data-intense applications[6] [44]. Shared memory persists for the life of a thread. Relative access time to shared memory is 10x longer than passing data directly to the kernel by way of the registers.

2.4.6 Constant Memory

Another type of fast access memory, known as *constant memory*, can be utilized by all of the blocks (and their threads) within a given **grid**. Constant memory is read-only and restricted to 64KB regardless of the GPU's Compute Capability and is available to an entire grid of computational threads versus a single block of threads as with shared memory [45]. Relative access time to constant memory is also 10x longer than access to data passed through the registers.

2.4.7 Texture Memory

The final memory type – and arguably the most important – that will be discussed in this dissertation is *texture memory* (sometimes referred to as "texture" for convenience). Access to texture memory is on the same order as relative access times to global memory; however, as we will discover in the following paragraph, textures are capable of storing data in a fast access cache. In doing so, relative access time to texture memory stored in the cache is reduced by an order of magnitude, which leads to access times that match both shared and constant memory. Unlike shared and constant memory, texture memory does not carry their same burden of extremely limited capacity. Maximum texture memory array dimensions are based on a number of factors including the Compute Capability of the card and the dimensionality of the texture being used; textures are available in three forms including 1D, 2D, and 3D arrays [39].

---

[6] Strategic partitioning of larger datasets can be used to leverage shared memory.

Textures have two special features made their use invaluable for the success of the research presented in this dissertation. One such feature is related to sampling the texture at a single location and the effect it has on the neighboring memory locations. When the texture is tasked by a thread to retrieve the value stored at a particular location, the texture returns the value stored at that location as expected (provided that location is coincident with a grid point). For the applications considered in this dissertation, data that is stored in texture memory – 3D texture memory to be specific – is data that resides on a grid. That is, values stored within the 3D texture only exist at equally-spaced, finite locations along the principal *x*-, *y*- and *z*-axes; however, the probability that a sample point is coincident with a grid-point is low.

So what happens if the texture is sampled at a non-grid point? This question brings the discussion to the next special feature associated with texture memory access: Automatic device-based interpolation [44]. Sampling the texture at non-grid points returns a value that is determined via fast trilinear interpolation (for 3D textures). In the context of this dissertation, this powerful feature allows the DRR to more closely emulate the true fluoroscopic image by providing a semi-continuous function to sample rather than a piecewise function defined only at grid points. (This feature is heavily discussed and exemplified in Chapter 4.)

By now it should be very clear why texture memory warranted such an extended dialogue as compared to the other memory types; it was the computational epicenter of this research and was instrumental to the success of this project. Figure 2.3 provides a summary of the primary CUDA memory types, their relative access times, and their availability with respect to the SMs.

`

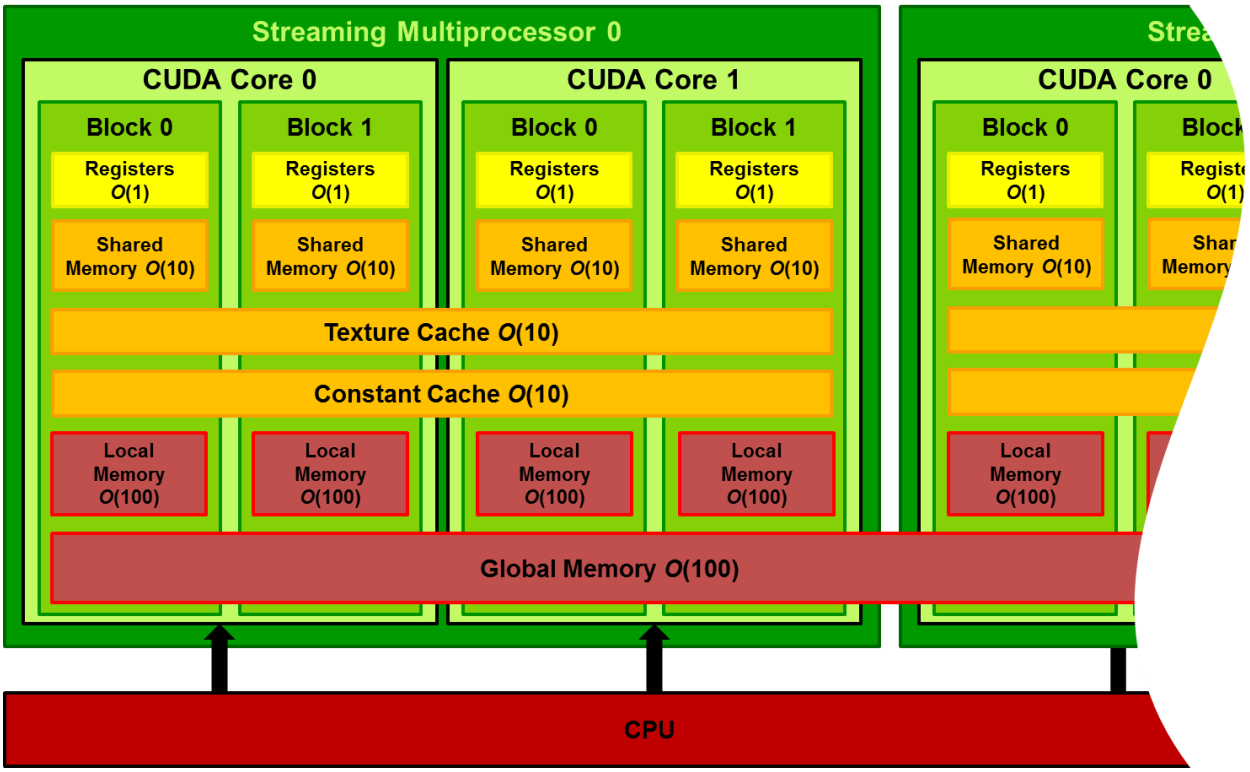


Figure 2.3. An examplified Streaming Multiprocessor . The primary memory types used in CUDA-based GPU computing and their relative memory access time (shown in *Big-O* notation) [47]. For example, if accessing data in constant memory took 10 µs, acessing data in global memory would take 100 µs.

Chapter 3: 2D-3D Image Registration

Motivated by an ongoing project aimed at non-invasive, marker-free measurement of the kinematics of the bones in the foot during gait, we consider a registration approach that involves the following:

1. Computing projections of CT imaging data (3D).
2. Calculating a quality measure to describe the agreement/discrepancy between the simulated projections and actual 2D images.
3. Optimization of the quality measure relative to the kinematic degrees of freedom to determine the kinematics corresponding to optimal registration.

A high-level overview of the registration process can be found in Figure 3.1. For our particular project, the 3D imaging modality is CT scan, the 2D modality is biplane fluoroscopy, the computed projection is a digitally reconstructed radiograph (DRR), the quality measure is normalized cross-correlation (NCC) between a pair of DRRs and a pair of corresponding fluoroscope images, and the 2D imaging includes a sequence of several hundred stereo image pairs.

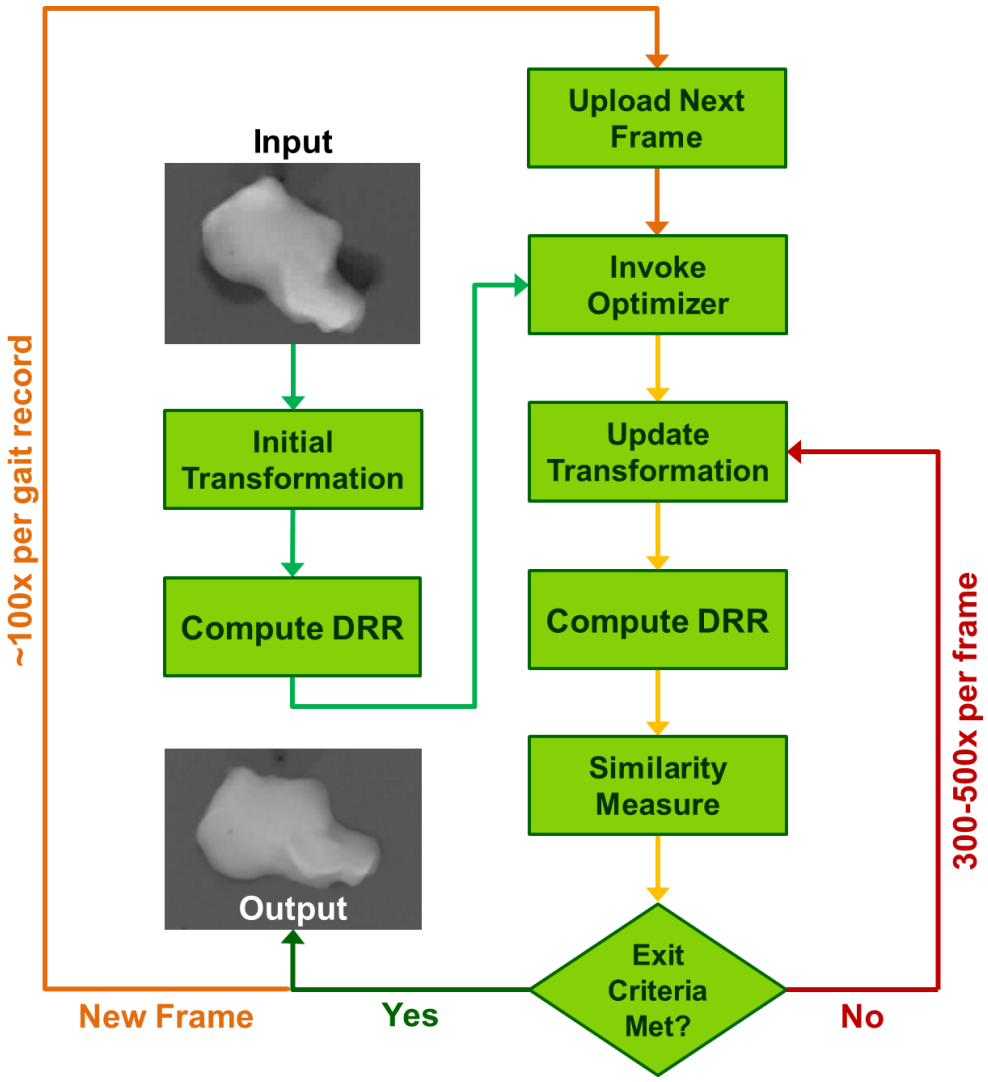
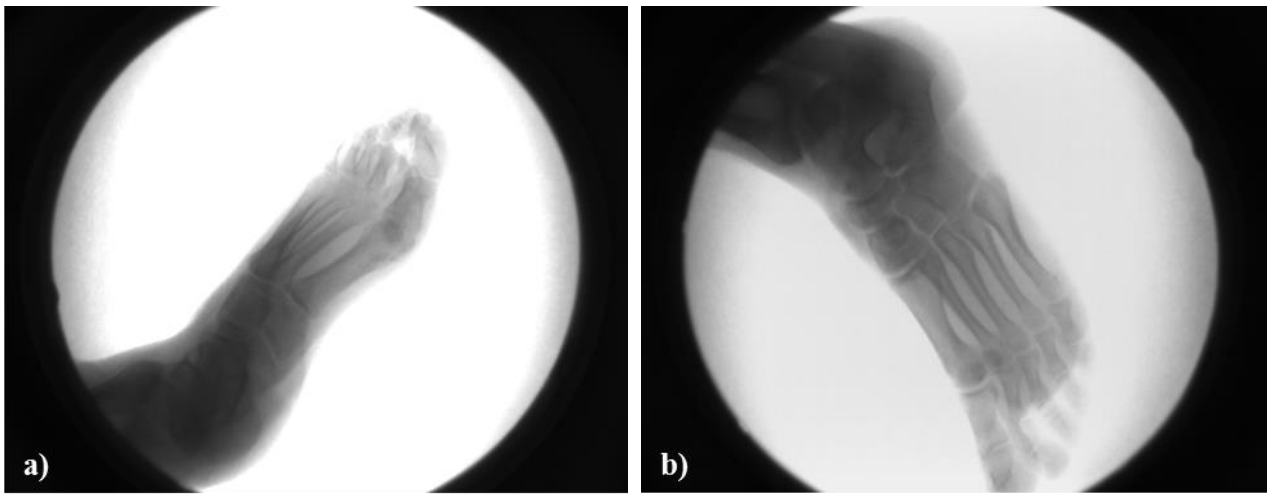


Figure 3.1. Process flow for image registration.

The research effort that led to the work in this chapter involves measuring the kinematics of the bones in the human foot during gait without the need to rely on physical markers, either superficial or surgically attached [50]. While early efforts to make such measurements required invasive methods involving surgical attachment of imaging markers [51, 52], current efforts were aimed at the realization of a non-invasive, markerless technique based on registration of 2D (Figure 3.2a & 3.2b) and 3D imaging (Figure 3.2c) [50, 53, 54].

Markerless approaches to track bone motion do not suffer from the problems encountered by superficial and surgically attached marker-based systems [50-52]. Superficially attached marker-based systems are susceptible to inaccurate motion recovery due to the skin – and subsequently the marker – moving relative to the bone. Surgically attached marker-based systems can also incorrectly represent bone motion due to interference of a surgically attached marker with regular anatomical motion (not to mention the extreme discomfort felt by the subject!). These shortcomings with marker-based systems prompted the investigation of a non-invasive, markerless technique based on registration of 2D (Figure 3.2a & 3.2b) and 3D imaging (Figure 3.2c) [50, 53, 54].

The basic idea behind the 2D-3D registration problem is to identify the actual 3D kinematic transformations for the set of bones based on a pair of stereoscopic 2D images (Figure 3.2a & 3.1b). Registration is considered complete once 3D transformations applied to the volumetric data (Figure 3.2c) are found that produce DRRs that provide the optimal match with the 2D stereoscopic image pairs captured at a particular time-step.



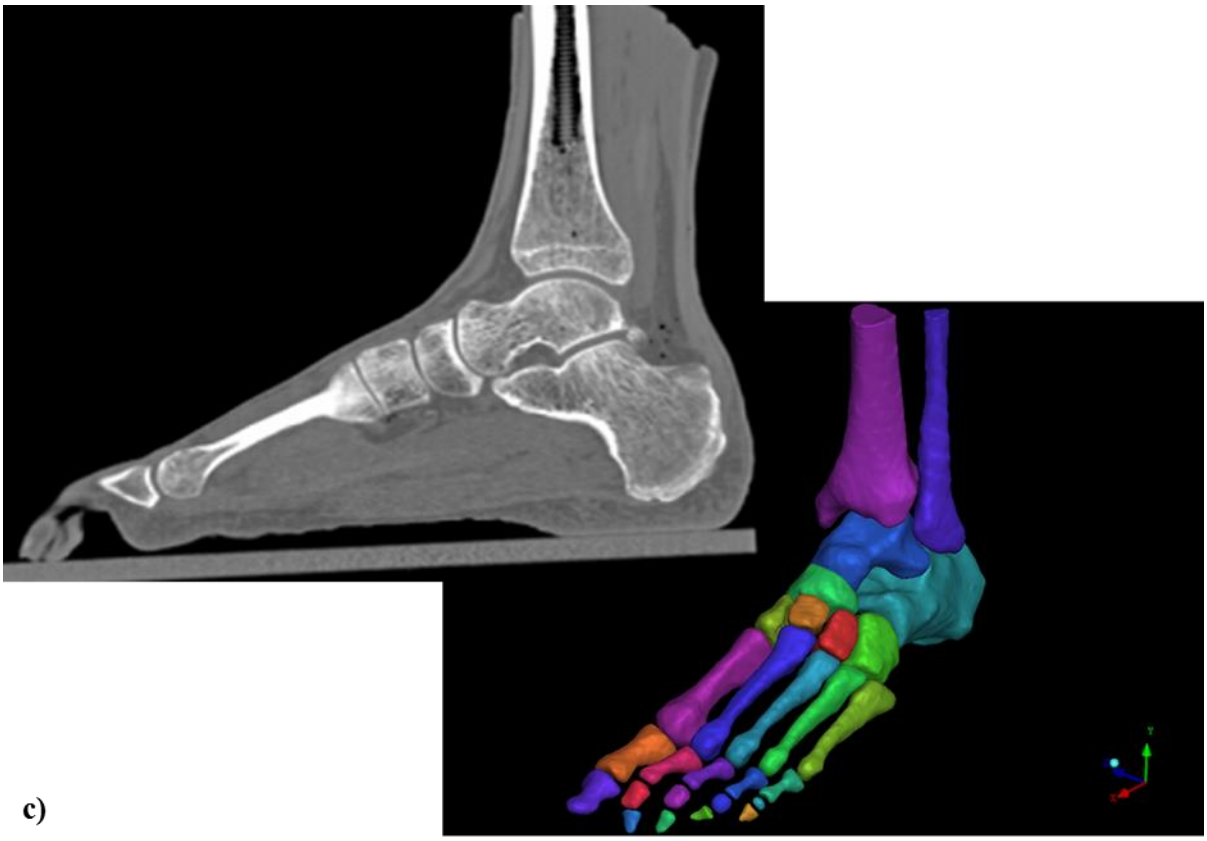


Figure 3.2. Visualization 2D-3D image registration data. (a-b) A stereoscopic pair of fluoroscopic images. (c) A representative slice of the CT scan data (upper left) and a volumetric (3D) representation of the CT scan data (bottom right).

To assist with registration, an optimizer[7] is employed to efficiently test hundreds of 3D configurations in an effort to maximize the similarity measure between the DRR and the stereo image pair (acquired from the biplane fluoroscope system mentioned in Chapter 1), and therefore the accuracy of the extracted 3D kinematic trajectory of the bones. During the optimization process that is invoked to determine bone kinematics by matching DRRs to fluoroscopic images, approximately 300-500 different 3D transformations are tested for each of the 20 bones of interest[8]. Each set of transformations requires that 2 DRRs be computed (one for each fluoroscope) and correlated with the corresponding fluoroscope image. This computation needs

[7] Two distinct optimizers were examined in this dissertation and are discussed in Chapter 7.
[8] Eight of the smallest phalanges are not tracked due to image resolution limitations.

to be performed for each of the ~60 frames of imaging data per gait record. Full processing of a single gait record thus involves computation of approximately 720,000 DRRs (20 bones x 300 transformations x 2 fluoroscopes x 60 frames) for fluoroscope image intensifiers with an 1152 x 896 pixel array ($>10^{11}$ total pixel values). Each pixel value in the DRR corresponds to an approximation of the line integral of the density[9] data along a ray from fluoroscope focus to image intensifier pixel location (Figure 3.3). If a DRR computation requires even 1s, full processing of a single gait sequence would take multiple days. For studies that aim to include multiple measurements on dozens of subjects, there is significant motivation to speed up the DRR computation.

[9] Density is proportional to the attenuation along a particular ray.

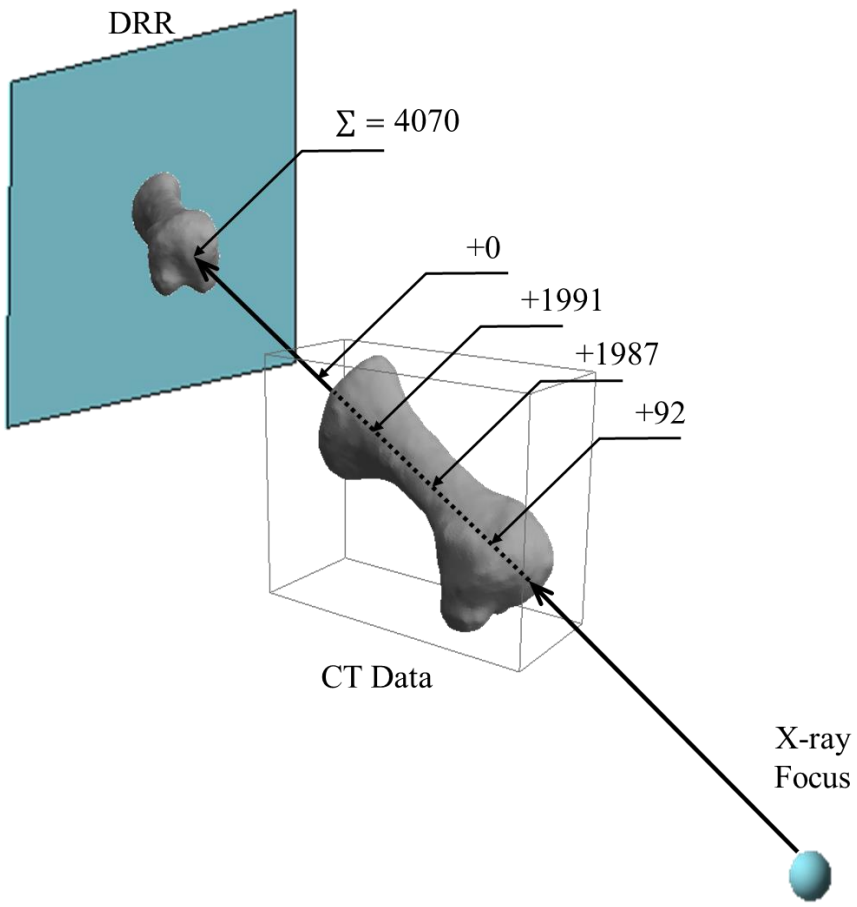


Figure 3.3. Volume rendering approach. A simplified illustration of the volume rendering approach to computing digitally reconstructed radiographs. For each pixel in the DRR, the line integral along the ray from X-ray focus to that pixel is approximated within the bounding volume of the bone by summing CT densities at each sampled point on the ray. Prior to being added to the current pixel sum, densities are weighted by a line segment, the length of which is determined by subdividing the portion of the ray that initializes at the bounding box intersection point and terminates at the bounding box exit point.

3.1 Imaging

For this study, volumetric imaging data in the form of a computed tomography (CT) image sequence (Philips CT MX8000IDT scanner – Philips Healthcare, Best, the Netherlands – at the Veterans Affairs Puget Sound Health Care System) was acquired for projection onto the simulated fluoroscope screens for 2D-3D registration. The 2D imaging system used in this study consists of a pair of Philips BV Pulsera C-arms with field of views that coincide with a radiolucent imaging area that rests atop a custom-built railed walkway (Figure 3.4). The C-arms have been modified by (a) removing the C-arm linkage (Figure 3.4a), so that the X-ray source

and image intensifier could be positioned independently (Figure 3.4b) and (b) by replacing the built-in cameras with high speed CMOS cameras that capture X-ray images at 1000 frames per second (fps), with a resolution of 1152 x 896 pixels (Vision Research, Wayne NJ). The system generates a massive amount of critical imaging data for subsequent processing and analysis; a mere 1.5s image sequence during a typical gait trial can produce up to 1.5 GB of data [55, 56]. For discussion purposes, the two biplane cameras and their corresponding image intensifiers are referred to as the "blue" and "green" subsystems.

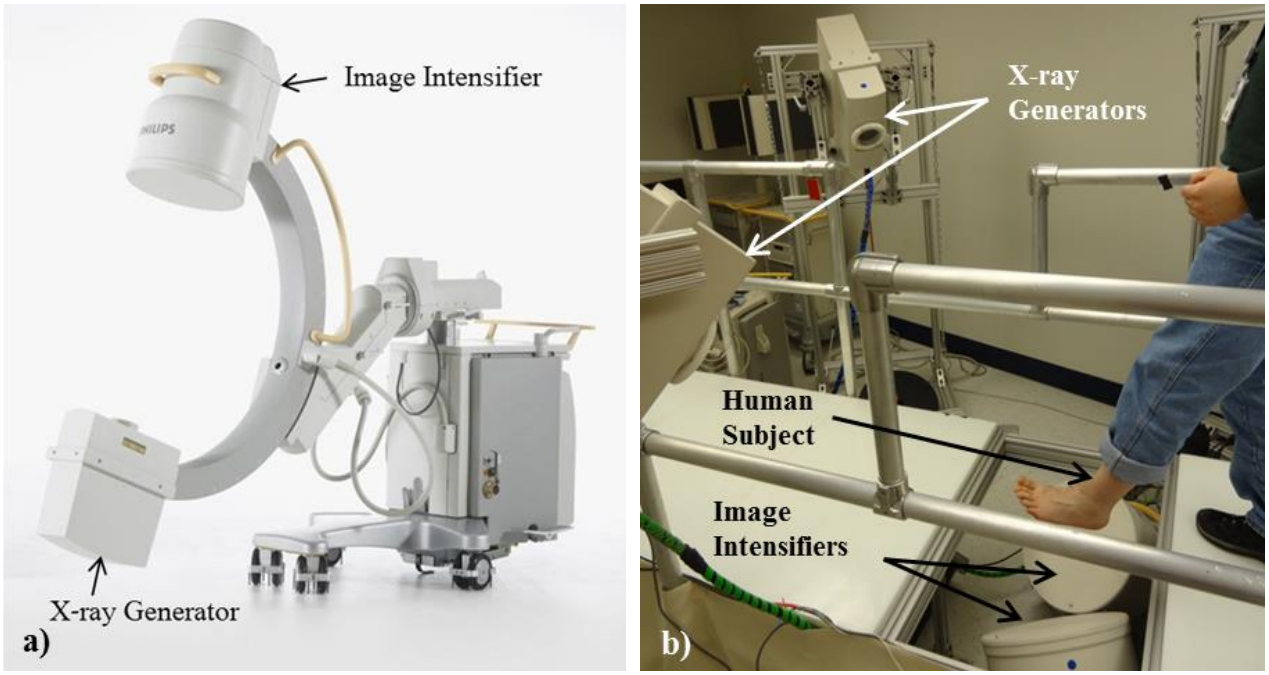


Figure 3.4 Biplane fluoroscope system. (a) A Philips Pulsera fluoroscope C-arm that was used to construct the novel biplane fluoroscopy system in (b). The biplane system in (b) consists of a ramp for the subject to walk on, handrails, a pair of disarticulate C-arms and a radiolucent composite panel that covers the imaging system (not shown) [56].

3.2 Data Processing

After the 3D imaging data are obtained, they are processed using a custom in-house segmentation algorithm – via the Multi-Rigid software package [57] – to identify the voxel set that belongs to each of the 20 bones of interest. Once processing is complete, the 3D data are separated into two distinct files: an integer label file and a floating point density file. The integer

label file serves as a voxel roadmap, identifying voxels belonging to a segmented object on a 0 to 28 range[10], which is used during DRR generation to determine the floating point density value of a voxel at a specific location within the CT volume. Figure 3.5 illustrates how the label file is used to determine which voxels in the density file contribute to a particular segmented object. The example is shown in 2D for simplicity, but the approach extends directly to 3D.

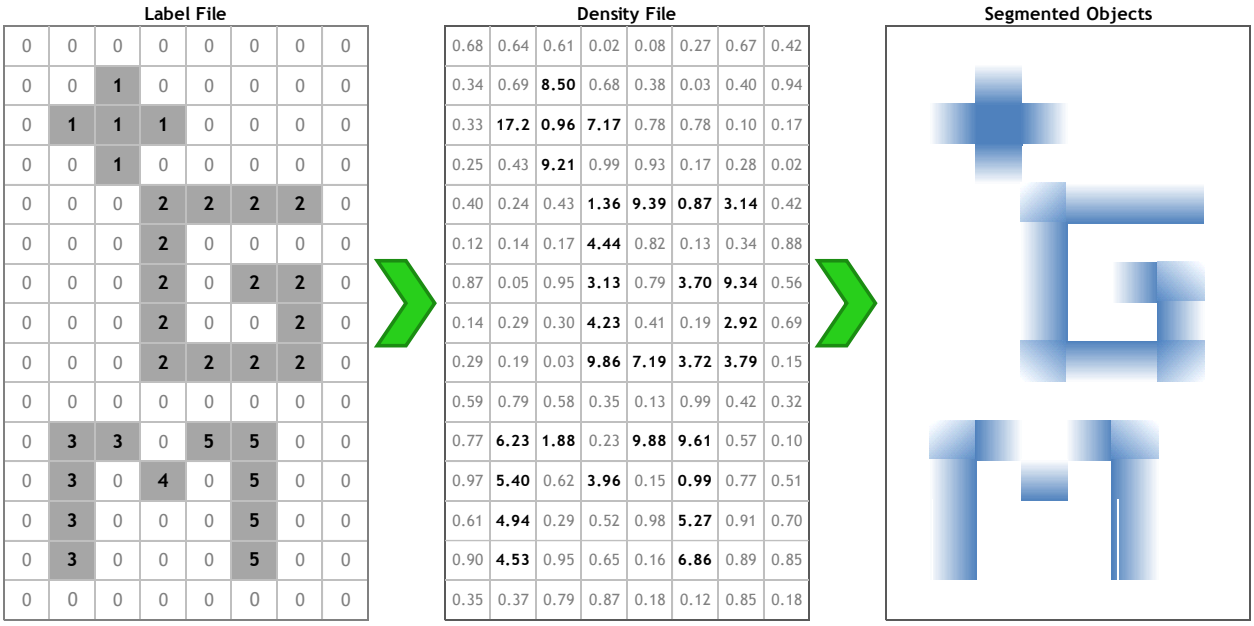


Figure 3.5. Visualization of label and density files. A slice through a synthetic label file (left), with voxels corresponding to segmented objects depicted by grid locations with label values of '1', '2', '3', '4', and '5'. Non-segmented objects (e.g., soft tissues, CT artifacts, etc.) are labeled '0'. A synthetic density file (middle) with voxel values corresponding to the measured Hounsfield units acquired during scanning. The segmented objects after the label file has been applied to the density file (shaded, right).

The floating point density file contains the actual Hounsfield Units acquired during the CT scan, which provide a mechanism for quantifying the radiodensity of a particular substance, i.e., the measure of a substance's ability to attenuate electromagnetic radiation [58]. The scale is normalized to water, which exhibits a Hounsfield Unit of '0'. A table of common substances and their corresponding Hounsfield Units are shown in Table 3.1.

[10] For example, a bone designated as "bone 3" is defined in the label file by voxels with a value of '3'.

Table 3.1. Common substances imaged via CT and their Hounsfield Unit [58].

Substance	*Hounsfield Unit (HU)*
Air	-1000
Lung	-400 to -600
Fat	-60 to -100
Water	0
Soft tissue	40 to 80
Bone	400 to 1000

Next, the imaging data acquired using the biplane fluoroscope system discussed in Section 3.1 is processed using custom in-house software to correct pincushioning and magnetic lens distortion [56]. Following preprocessing, the data is ready for 2D-3D image registration using one of two custom software packages. The first, referred to as the "CPUDRR" herein, was built using MATLAB and the CONDOR optimization suite [59]. This software package has been used to successfully process several sample data sets, including bead-based validation data[11], bone-based validation data[12] and multiple gait trials from two living subjects yielding 3D kinematic information for the tibia, calcaneus, first metatarsal and hallux during gait (heel-strike through toe-off). The second package, DRRACC – vaguely mentioned in Chapter 1 and discussed in Chapter 4 – is the culmination of this dissertation and was built using NVIDIA's CUDA. For the remainder of this dissertation, "CUDADRR" will refer to DRRACC's method for computing DRRs. Validation of DRRACC's image formation approach and optimization strategy against the previously proven and independently developed CPUDRR system is the main focus of Chapter 8.

[11] Bead-based validation data consists of a radiolucent foam calibration block with radiodense tantalum beads embedded at known locations. The calibration block is then imaged using the biplane fluoroscope system and DRRs of the calibration block are compared to the fluoroscope images. Errors are computed by finding the in-plane distance between the bead centroids in the DRRs and corresponding fluoroscope image pair. A sub-millimeter root mean square error is required to validate the software. Additional details can be found in Chapter 8.

[12] Bone-based validation data consists of a bone and tantalum beads embedded in foam. The bone is then subjected to pure translation or pure rotation using a highly precise stepper motor. The software is used to register the bone as it moves through a prescribed set of translations or rotations. 3D kinematic information about the bones in the form of a transformation matrix is compared to the known positions as recorded by the stepper motor. A sub-millimeter and sub-degree root mean square error between the position determined by the software and the measured position is required to validate the software. Additional details can be found in Chapter 8.

Chapter 4: DRRACC and DRRACCULA

Digitally Reconstructed Radiographs Accelerated by CUDA (DRR Accelerator or DRRACC for short – pronounced like the theoretical physicist Paul Dirac) was developed as a flexible, general purpose, distributable[13] toolkit for fast, parallel DRR computing and 2D-3D image registration on the GPU. The development of DRRACC was motivated by the desire to reduce the computational choke points of 2D-3D image registration: DRR generation and image correlation. To achieve this goal, several image processing tools including DRR computation, masking, correlation, composition, registration and visualization, were converted into parallel functions that run exclusively on the GPU.

4.1 Programming environment

Development of the DRRACC toolkit took place in a C/C++ programming environment using Microsoft Visual Studio 2010 [60] in conjunction with NVIDIA's CUDA v5.5 [44]. This permits a relatively simple approach to writing C/C++ code that takes advantage of the GPU's tremendous processing capabilities without the need to dabble with traditional GPU shader programming [39]. All development took place on a midlevel workstation with an Intel Core 2 Quad CPU @ 2.4 GHz. DRRACC was tested for compatibility on several NVIDIA GPUs – with varying Compute Capabilities – including a Tesla K20, Tesla C2070, Tesla C2050, GeForce GTX460 v2, and GeForce GT440. Comparative timings for the various graphics cards can be found in Chapter 8.

[13] At the time of dissertation submission, the distributable version of the software was not yet available.

4.2 Toolkit interface

The most important goal during toolkit development, aside from rapid DRR computation, was the creation of a main program that would allow a broad audience to harness the power of GPU computing without months of formalized training. Development of the DRRACC toolkit interface took advantage of CUDA C in conjunction with VS2010 C/C++ to achieve this goal. The flexibility of the CUDA C programming architecture promoted a toolkit interface that would allow the user to simply input the necessary data and hit "go". Figure 4.1 provides a high-level overview of toolkit flow in terms of data input and handling. The blanket statement of "Copy to device structure" in Figure 4.1 is used for convenience; the "copy" functions perform all necessary actions to setup and fill device-side structures with host-side input from the user.

The toolkit reads the data and automatically assigns it to internally defined structures. This approach shields the user from needing to explicitly deal with the details of DRRACC's device-based data structures. The structures are then copied into CUDA memory types – global, shared and texture – that are readily accessible by the various kernels that comprise the fast DRR algorithm. Parallelized functions requiring kernel launches include: masking, dilation/erosion, cropping, composition, summation, gradient approximation, convolution, DRR generation, and similarity measure via NCC. DRRACC also provides support for device functions that can be accessed from within a kernel.

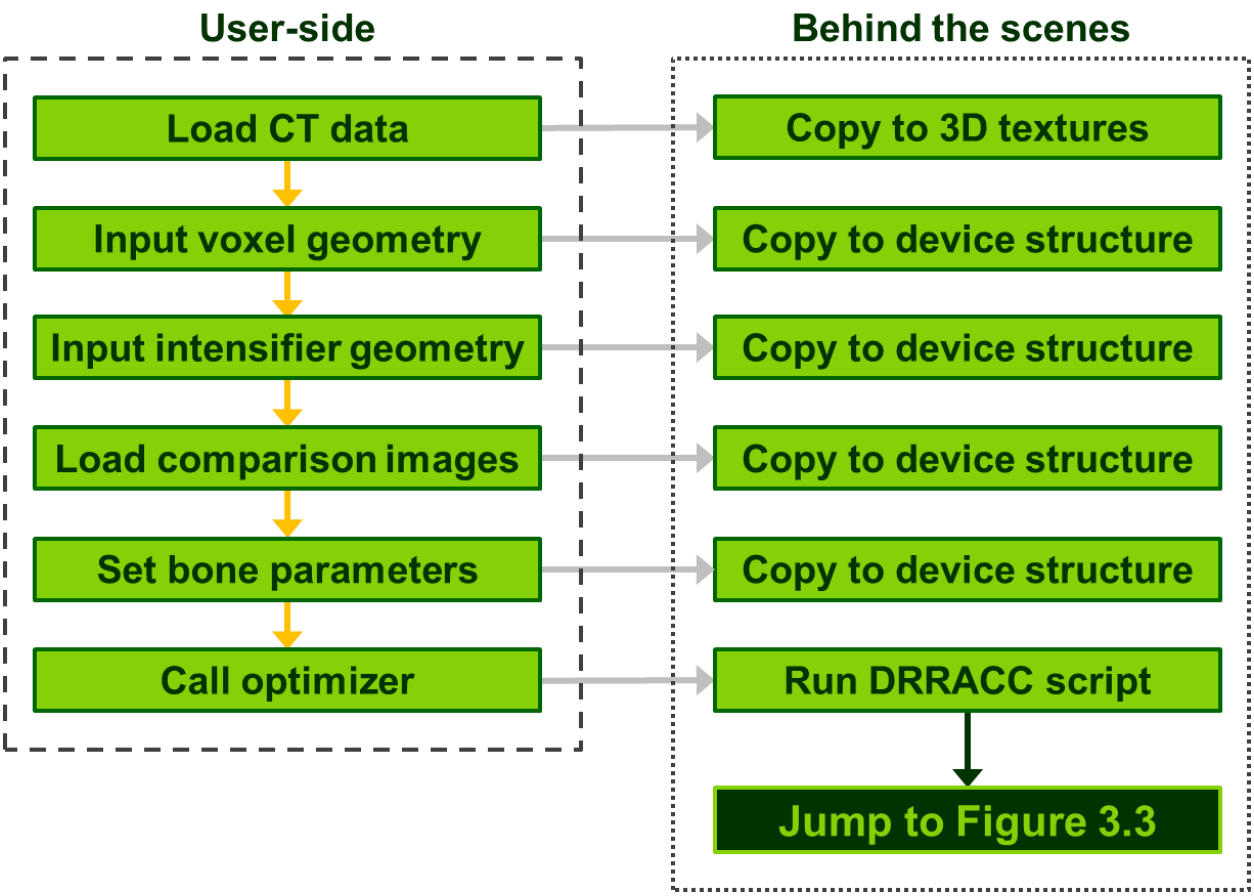


Figure 4.1. DRRACC data flow. A high-level overview of DRRACC's memory transfers to shield the user from GPU computing.

To accommodate users with different programming backgrounds, we developed the DRRACC interface using traditional C with less than 85 lines of code: The user simply must follow the instructions and pass input data into DRRACC. The input data for the DRRACC toolkit is detailed in the following bullet-list:

- Voxel information:
    - Segmented volumetric imaging datasets
        - Integer label data[14] (0 to 28) – identifies which bone each voxel belongs to

[14] Label '0' is reserved for non-segmented voxels, e.g., soft tissue, ID tags, miscellaneous artifacts, etc.

        - Floating point density data – contains voxel densities in Hounsfield units
    - Physical voxel spacing – scanning slice dimensions in *x*-, *y*-, and *z*-directions
    - Voxel count – number of voxels in *x*-, *y*-, and *z*-directions
- Intensifier geometry:
    - X-ray focal point(s) – coordinates of X-ray focus
    - Intensifier screen corners – coordinates of (3) screen corners (prevents reflection)
    - Intensifier screen dimensions – width and height in terms of number of pixels
- Rigid body transformation matrices – one per segmented object to define 3D position
    - The inverse is applied to the ray start (x-ray focus) to the ray end (pixel on intensifier screen) to describe the ray in the frame of the volumetric data
- Fluoroscope images – used to quantify registration quality

In summary, the toolkit performs all of the necessary steps, including file reads/writes, memory allocations, memory transfers and computations, for DRR generation, image comparison and registration via optimization. Many of the preprocessing steps are device functions that efficiently store the data in a format which enhances performance. After the DRR and similarity measure have been computed, the user has the option to request data in the form of a binary output file or a pointer that points to the memory location of the output.

4.3 Preprocessing

Several preprocessing steps are performed to provide DRRACC's numerous parallel kernels with fast access to frequently used data. During initial setup, axis-aligned bone bounding boxes are computed by performing a single-pass through the label data and storing the minimum and maximum coordinates for the voxels that contribute to a particular bone. In doing so, we define

independent bounding boxes that are aligned with the principal axes of the data set. Corresponding geometric centers are then calculated for each bounding box, which are utilized during optimization (bones are rotated about their centroid). It is worth noting that preprocessing is performed only once per gait record and completed in ~5s, which includes reading in CT label and density data (generally larger than 300 MB) and storing it in 3D textures.

4.4 Fast DRR computation

Recall that CPUDRR computes a DRR in ~750 ms, which is prohibitively slow for processing even small pools of subjects. To achieve significant reductions in the DRR calculation time, we employ general purpose computing on the graphics card [44] to parallelize the computation of the DRR corresponding to a given set of transformations of the bone configurations. (The toolkit can be applied to any sort of segmented objects, but we will refer to the objects as bones in the context of the gait analysis examples.) The toolkit supports independent kinematics for each segmented bone up to a maximum of 28 bones, and individual bone DRRs can be computed and then composited by simply summing corresponding pixel intensities for image resolutions with up to a total of 1,048,576 pixels (equivalent to a 1024 x 1024 image).

A DRR is comprised of a 2D array of pixel intensity values, and each intensity value corresponds to a numerical estimate of the integral along a ray – from the X-ray focus to the pixel location on the intensifier screen – of the density function obtained by interpolating the CT data (Figure 4.2). The coordinates at each step along the ray are used to sample the 3D texture storing the CT data and the voxel located at the specified coordinates provide the algorithm with a density value[15]. The sum of all voxel values along a single ray corresponds to the final intensity value for a single

[15] If the point on the ray being sampled does not coincide with the center of the voxel, trilinear interpolation is automatically invoked by the 3D texture memory and an interpolated density value is returned.

pixel in the DRR. The computation employs one ray per pixel (terminating at the center of the pixel), and we parallelize the computation by launching a computational thread corresponding to each ray. For a single bone DRR, the computational load for each thread involves computing the intersection with a bounding box (requiring a few ray-plane intersections) and summing on the order of 300 interpolated density values.

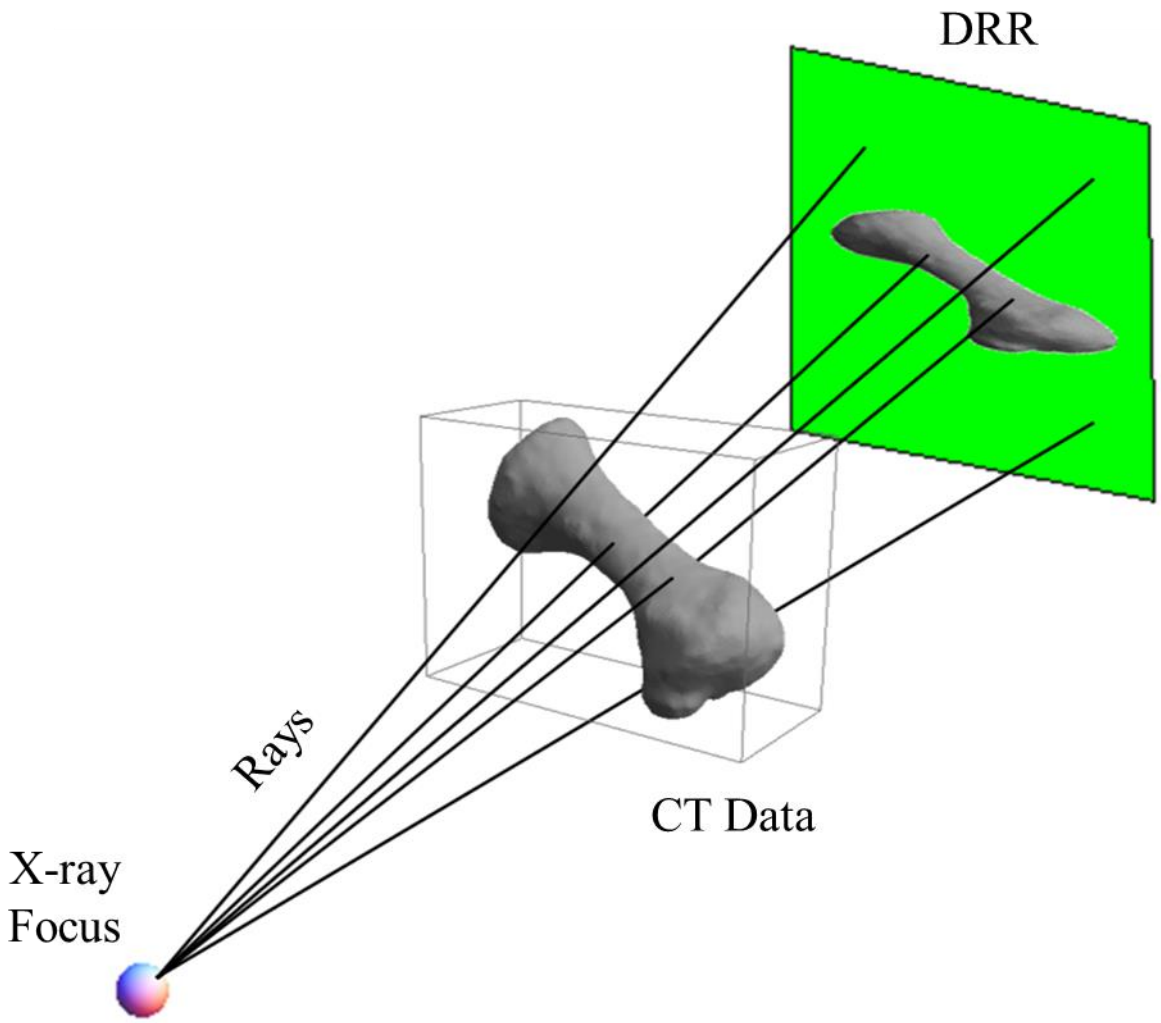
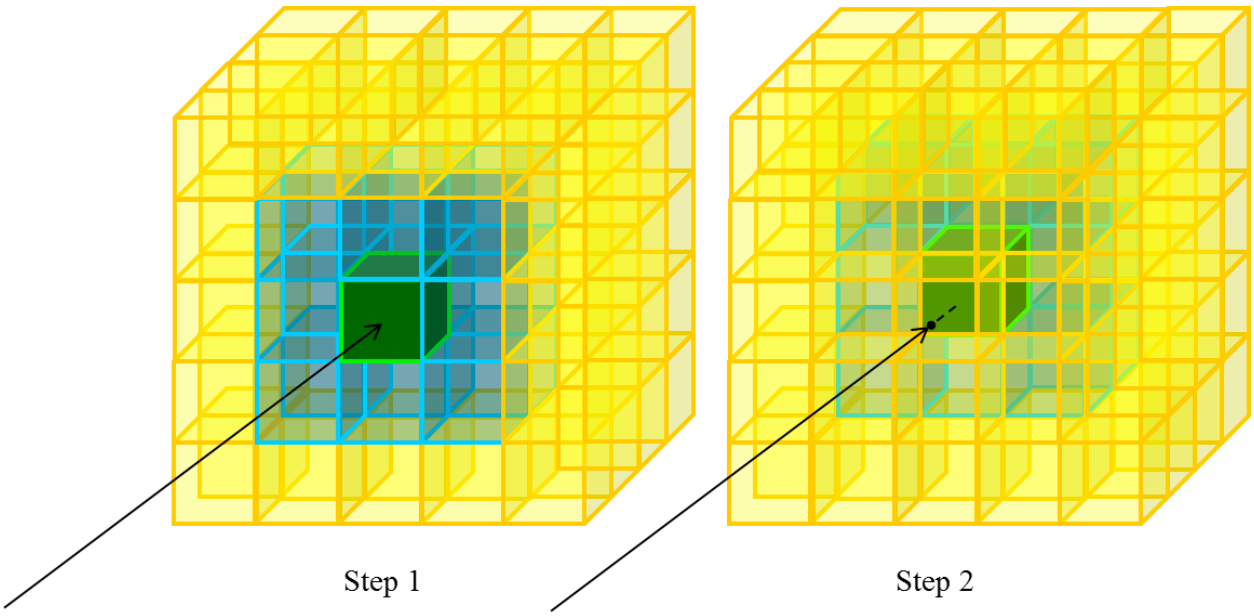


Figure 4.2. DRRACC's parallelization model for computing DRRs. Rays originating at the X-ray focus (bottom left) pass through the CT voxel set (middle) accumulating densities, with the final sum being projected as a DRR (upper right). Only voxels corresponding to the specified label value contribute to the computation. Note that in this example, zero-valued pixels correspond to the ray intersecting voxels assigned to a label value of '0' and are depicted in green.

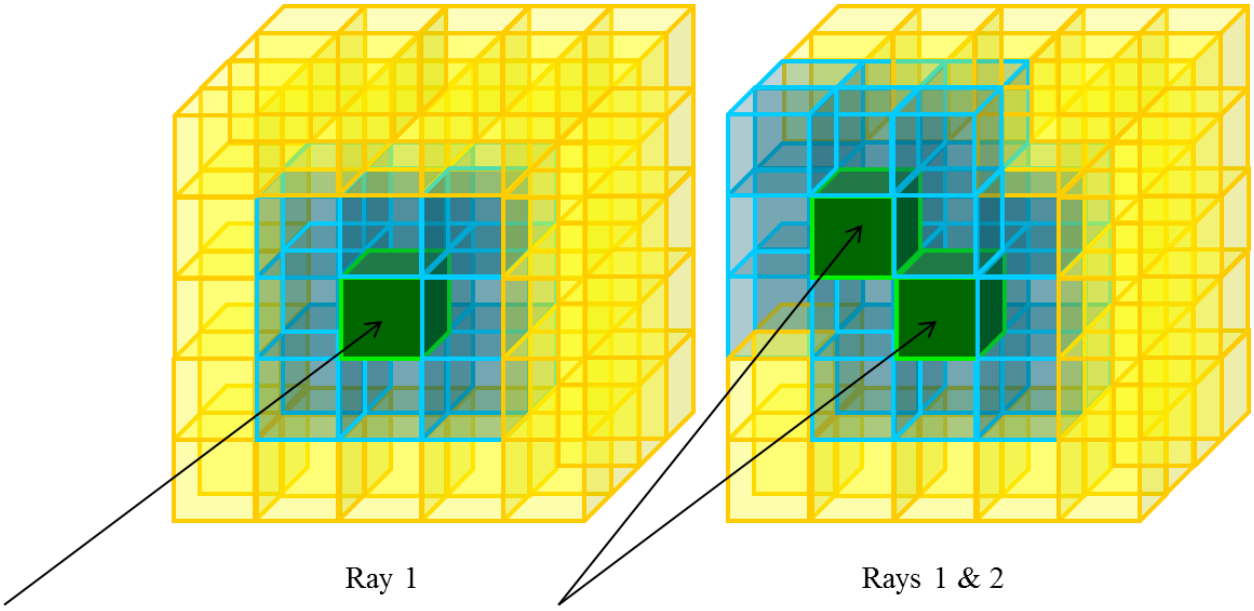
The critical issues that determine computational performance are fast access to and interpolation of CT data, and fast access to intensifier geometry[16]. Intensifier geometry involves a small amount of data that applies for all pixels, so our approach is to store the intensifier information in device-side structures and pass a pointer through the register at kernel launch. Given that many of our CT sets contain over 23 million voxels, the same strategy could not be applied to the voxel data.

Fast access to and interpolation of the > 95 MB CT data – which is too large to be stored in CUDA's 64 KB constant or 48 KB shared memory – is achieved using CUDA's 3D texture memory. One 3D texture stores the integer label data and is sampled in "nearest neighbor" mode[17]. A second 3D texture stores the floating point density data and tri-linear interpolation is rapidly computed in hardware on the texture module. For both 3D textures, the sampling is spatially coherent along the current ray and along neighboring rays comprising a given warp; during sampling, CUDA 3D textures automatically read and store values in the vicinity of the current sample location in a fast access cache (Figure 4.3). The combination of passing intensifier information through the register and CT data in a pair of 3D textures effectively supports efficient computation of each single bone DRR. A multi-bone DRR is achieved by summing intensity values from single bone DRRs. All summations in DRRACC are performed using kernel launches, shared memory, and atomics, to prevent computational bottlenecks associated with host-side summing.

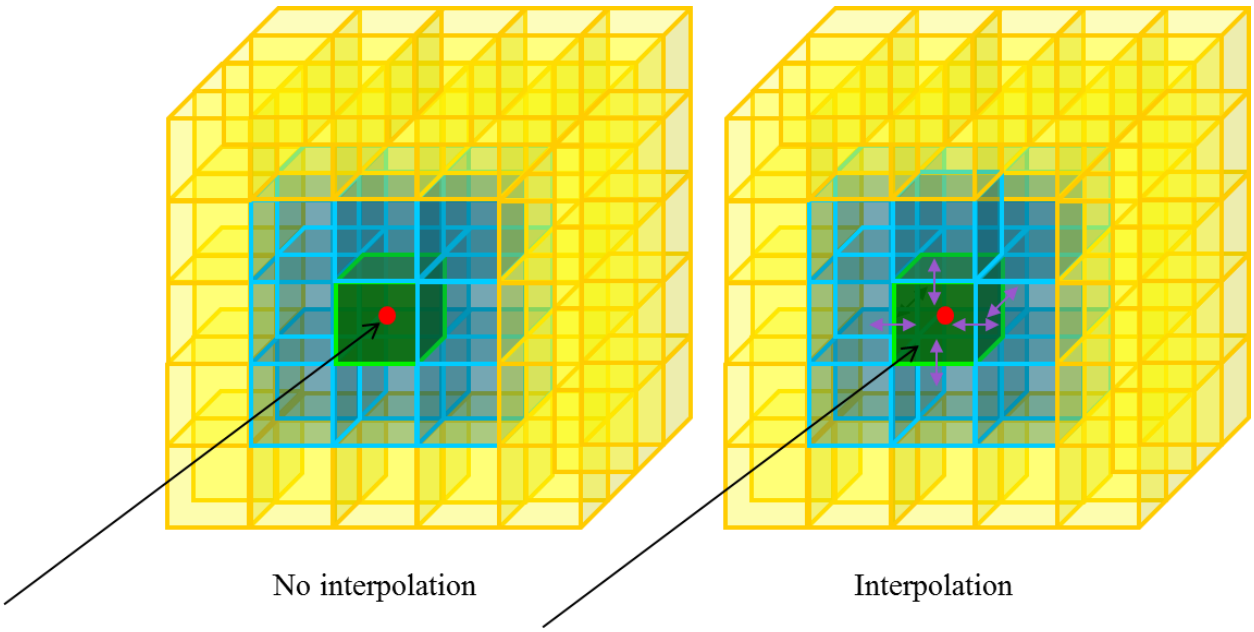
[16] X-ray focus, intensifier size, intensifier location (defined by the center of pixels located at three screen corners).
[17] This sampling mode returns the value of the nearest voxel.



a)



b)



c)

Figure 4.3. 3D Texture sampling. (a) A simple illustration to visualize texture memory and fast access caching. The green block represents the current voxel being sampled as we step through the CT volume along the ray from focus to intensifier. The blue blocks represent texture locations that are stored in a fast access cache when the green block is being sampled. The yellow blocks are outside of the storage region and are not copied into the fast access cache. (b) All rays being "fired" share commonly accessed memory (e.g., intensifier geometry, fast access cache, etc.), aside from which the rays are entirely independent and so a unique computational thread can be launched for each pixel in the DRR. CUDA launches threads in warps (32 threads), but to help with visualization, only two rays are drawn. The image on the left shows ray 1, with the intersected voxel (center, green), cached voxels (center, blue) and missed voxels (yellow). The image on the right shows rays 1 and 2. Note that ray 2 intersects a voxel that is within first ray's fast cache region. Now imagine the image with several additional rays that were all launched in parallel (per warp); it is clear that CUDA greatly increases the computational efficacy of mass data access. (c) The image on the left depicts the case when a ray intersects the exact center of a voxel. In this scenario, sampling the texture yields the exact value stored at that particular location. The image on the right reveals the more prevalent case where the ray does not strike the center of the voxel. In this scenario, CUDA samples the texture using a built-in parallel trilinear interpolation function.

When the parallel kernel is called to compute a DRR, a 2D array of thread blocks are launched; the number of blocks per grid is determined by a function of the image dimensions and the number of threads per block is set by the user with a `#define`. (This parallelization strategy is particularly beneficial when running the software on machines with different GPU specifications.) CUDA provides each thread with the components of its block and thread indices (introduced in Chapter 2 as `blockIdx and threadIdx`). A simple computation then determines

the pixel location corresponding to the thread. We then calculate and apply the inverse of the current rotation matrix, for a chosen bone, to the initial and terminal positions of the ray (the focus and pixel location), which places the ray in the frame of the CT data. Next, we test the ray for intersection with the CT bounding box as given by Owen [61]. If the ray misses the bounding box, e.g., the pixel corresponding to the ray does not lie on the projection of the bounding box onto the digital intensifier screen, the current ray does not contribute to the DRR and DRRACC returns an intensity value of '0.0'. If the ray hits the bounding box, the entry and exit locations are computed and the intensity is initialized to 0. We subdivide the ray from entry to exit into a constant number of steps ($n = 300$; discussed in Chapter 7), which defines the length of the steps along the ray. At each step, if the label texture matches the specified bone label, we increment the intensity by the value returned from the density texture.

After summing the densities along a ray passing through the CT data, the pixel value (intensity) corresponding to the accumulated densities (attenuation) is written to the corresponding position in a device array that stores the DRR data for the specified bone. Since each array stores a flattened (1D) image for a single bone, values are indexed into the array using an offset calculation ($position = thread_x + thread_y * image_{width}$)[18].

An excessively large bounding box will return more "hits" with the ray-box intersection test than a tightly defined bounding box (e.g., axis-aligned corresponding to minimum/maximum label file locations for a given label index). Moreover, the excessive bounding box would contain more empty voxels than the tightly defined bounding box; ergo an increase execution time would be

[18] Note that evaluation of threads corresponding to a ray that did not intersect the bounding box is extremely inexpensive as only the ray-box intersection test is required. However, the thread associated with the now inactive ray is also inactive until all other rays in the warp are finished executing. The evaluation is particularly efficient if all the threads in a given warp miss the bounding box.

expected as a direct consequence of stepping along the ray accumulating zero-value densities. Thus, implementation of individual bounding boxes that tightly contain each bone, as opposed to using the full voxel extent as the bounding box, can significantly reduce the number of rays that intersect the bounding box, which in turn minimizes computation time. Further discussion on the effects of bounding box size on computational performance can be found in Chapter 8.

Multibone DRRs are comprised of the collection of single bone DRRs that are generated independent of one another and composited using a parallelized image composition function. User-defined input flags for bone visibility (e.g., if visible, it will appear in the DRR) determine which bones are included in the composition function and the function sums pixel values for a given index from each of the single bone DRRs to generate the multibone DRR. This approach facilitates fast DRR updates when transformations are applied to an active bone by eliminating the need to re-compute DRRs for the inactive bones (illustrated in Figure 4.4):

1. DRRACC removes the active bone from the composite DRR via parallel decomposition
2. A new transformation is applied to the active bone and an updated DRR is computed
3. DRRACC adds the updated DRR back to the composite DRR.
4. Repeat steps 1-3 for all active bones.

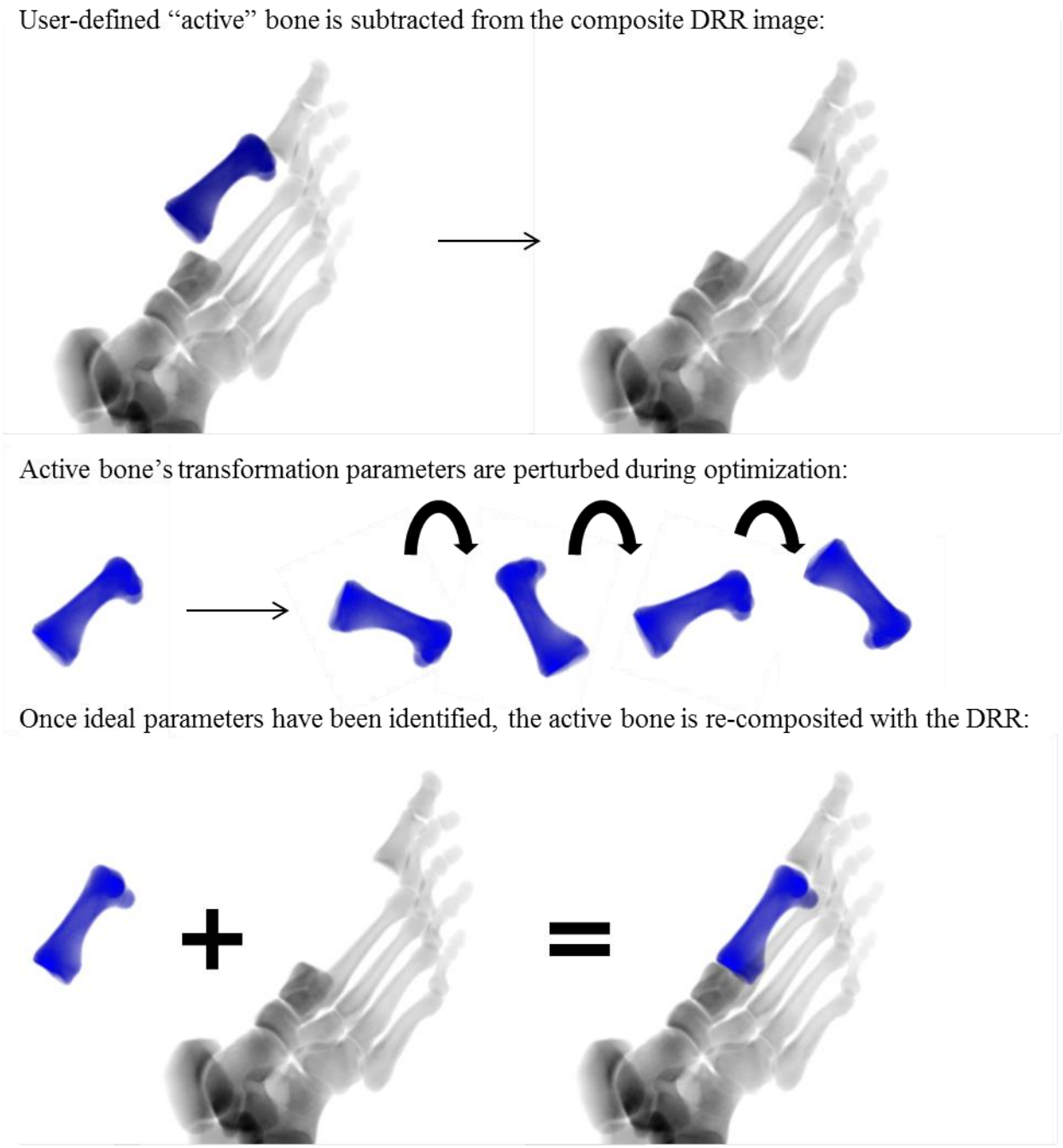


Figure 4.4. Multibone optimization scheme.

4.5 DRRACO[19]

The toolkit DRRACO – DRRs Accelerated by CUDA Operations – was created through an effort to improve the overall ease, flexibility and usability of DRRACC, while maintaining its full

[19] Pronunciation of DRRACO is identical to the constellation *Draco*.

functionality and computational efficiency. DRRACC itself was essentially a monolithic block of code that was intended to perform a primary task: accelerate the generation of DRRs. The interface described in Section 4.2 was simple and intuitive, but the user was limited to thinking about the problem in one way without the capacity to explore alternative solutions.

A lot of information was learned about data and process flow with the development and testing of DRRACC, and it was decided that DRRACC would undergo a complete overhaul that would transform the software into a collection of library algorithms that could be accessed independent of one another. Development was centered on the user; ease of use, flexibility, efficiency and access to the heart of the parallel computing power, while including the ever important shroud that concealed the details needed to tap into that power. The result of these efforts was DRRACO.

Each of DRRACO's parallel functions was designed to execute entirely independent of one another, which will enable DRRACO's use in applications beyond 2D-3D image registration. Beyond function independency, features not present in DRRACC were implemented to streamline DRRACO's process flow. One of the primary features of DRRACO is the explicit use of host and device image structures shown in Table 4.1.

Table 4.1. DRRACO's hostImage and deviceImage structures.

```
// Host Image
struct hostImage
{
	int width;
	int height;
	float *ptr;
	size_t size;
};
// Device Image
struct deviceImage
{
	int width;
	int height;
	float *ptr;
	size_t size;
};
```

These simple structures make it possible for the user to take full advantage of DRRACO's GPU-based operations without any significant experience with GPU computing[20]. Both structures are comprised of four elements: width, height, *ptr and size. The width and height are defined by the user. The size is the `width x height x sizeof(data type)` and is automatically stored when the user invokes the `makeHostImage()` or `makeDeviceImage()` functions. The *ptr is a pointer to the image memory location and is required on the CUDA side of DRRACO.

Each of DRRACO's parallel functions was designed to execute entirely independent of one another, which will enable DRRACO's use in applications beyond 2D-3D image registration. A list of the accessible functions is located in Table 4.2 (a complete description of functions and their arguments can be found in Appendix B). The "make" functions in Table 4.1 are used to allocate and initialize memory for various host-side and device-side functions. The "load" and "extract" functions are used for performing host-to-device, device-to-host and device-to-device memory copies.

[20] The user will need to recognize and practice the limitations of *host* images and *device* images.

Table 4.2. A list of DRRACO's user-accessible functions. Full descriptions of the functions can be found in Appendix B, with an example main function in Appendix C.

```
/************************MISCELLANEOUS Functions************************/
// Print active device
int checkGPU(int deviceID);
// Choose active device (ONLY for machines with multiple GPUs)
int setGPU(int deviceID);
// Find axis-aligned bounding boxes and compute corresponding centroids
boneData*   findCentroids(int   *label,   hostVoxelGeometry   hVox,   boneData
boneInfo[NROWS]);
// Free memory (device pointers)
int Free(deviceImage d);
// Free memory (device pointers)
int Free(deviceImage *d);
// Free memory (host pointers)
int Free(hostImage h);
// Reset device
int reset();
/*****************************MAKE Functions*****************************/
// Allocate and initialize host image
hostImage makeHostImage(int width, int height, float initVal);
hostImage makeHostImage(deviceImage dImg, float initVal);
hostImage makeHostImage(deviceImage dImg, float initVal, int copy);
// Allocate and initialize device image
deviceImage makeDeviceImage(int width, int height, float initVal);
deviceImage makeDeviceImage(hostImage hImg, float initVal);
deviceImage makeDeviceImage(hostImage hImg, float initVal, int copy);
// Create host structure to store voxel geometry
hostVoxelGeometry makeHostVoxelGeo();
// Create device structure to store voxel geometry
deviceVoxelGeometry* makeDeviceVoxelGeo(hostVoxelGeometry hVoxGeo);
// Create host structure to store fluoro geometry
hostFluoroGeometry makeHostFluoroGeo();
// Load fluoroscope geometry and store in device structure
deviceFluoroGeometry* makeDeviceFluoroGeo(hostFluoroGeometry hostStruct);
// Create host structure to store bone data
boneData makeBoneData();
// Create device structure to store bone data
deviceBoneData* makeDeviceBoneData(boneData *hBoneInfo);
// Create Region of Interest (ROI)
int* makeROI(deviceImage dImg);
/*************************TRANSFER Functions*************************/
// Load image and store in device array
int loadImage(hostImage hImg, deviceImage dImg);
// Copy image from device to host memory
int extractImage(deviceImage dImg, hostImage hImg);
// Transfer device image to device
int deviceImageTransfer(deviceImage dImg, deviceImage dImg2);
// Load CT data and store in texture memory
deviceVoxelGeometry* loadCTData(float *density, int *label, hostVoxelGeometry
```

```
hVoxGeo);
/**********************IMAGE PROCESSING Functions**********************/
// Calculate a dilated mask
int dilateMask(deviceImage dImg, deviceImage dMask, int dilationSz);
// Calculate an eroded mask
int erodeMask(deviceImage dImg, deviceImage dMask, int erosionSz);
// Apply a mask to a device image
int applyMask(deviceImage dImg, deviceImage dMask, deviceImage dMaskedImage);
// Convolve an image with a kernel
int imageConvolve(deviceImage dImg, deviceImage dConvImage, float *kernel,
int halfW);
// Use finite difference approximation to compute a gradient image
int finiteDiffGradient(deviceImage dImg, deviceImage dGradImage, int type,
int dir);
// Composite images
int compositeImages(    deviceImage dDRRs[NROWS],
deviceImage dCompositeImage,
int compositeList[NROWS],
int type);
// Compute DRRs
outputData computeDRR(  deviceImage bDRRs[NROWS],
deviceImage gDRRs[NROWS],
int visibleBones[MAXOBJECTS + 1],
deviceVoxelGeometry *dVox,
deviceFluoroGeometry *dFluoro,
boneData boneInfo[MAXOBJECTS + 1],
outputData dataOut);
// Compute NCC between two images
float calculateNCC(deviceImage drrPTR, deviceImage fluoroPTR);
float calculateNCC(deviceImage drrPTR, deviceImage fluoroPTR, int ROI[4]);
// Perform all necessary tasks for a single optimization call
outputData exampleMeritFunction(    deviceImage bDRRs[NROWS],
                                    deviceImage gDRRs[NROWS],
                                    deviceImage bFluoro,
                                    deviceImage gFluoro,
                                    deviceImage bEdgeFluoroi,
                                    deviceImage bEdgeFluoroj,
                                    deviceImage gEdgeFluoroi,
                                    deviceImage gEdgeFluoroj,
                                    int gradType,
                                    deviceVoxelGeometry *dVox,
                                    deviceFluoroGeometry *dFluoro,
                                    boneData boneInfo[NROWS],
                                    CONDORparams cParams,
                                    int optBones[NROWS],
                                    int visibility[NROWS],
                                    outputData dataOut);
/**************************OPTIMIZER Functions**************************/
// Perform all necessary tasks for invoking optimizer
outputData runOPT(      meritFunc mc,
```

```
int optBones[NROWS],
int visibility[NROWS],
deviceVoxelGeometry *dVox,
deviceFluoroGeometry *dFluoro,
CONDORparams cParams,
int gradType,
deviceImage bDRRs[NROWS],
deviceImage gDRRs[NROWS],
deviceImage bFluoro,
deviceImage gFluoro,
deviceImage bFluoroEdgei,
deviceImage bFluoroEdgej,
deviceImage gFluoroEdgei,
deviceImage gFluoroEdgej,
boneData boneInfo[NROWS],
float lowerBounds[6],
float upperBounds[6],
outputData dataOut);
```

Chapter 5: Image Correlation

A variety of techniques, both intensity- and spatial-based, for computing the correlation between two images can be found in the literature [6, 10]. DRRACC's image correlation is performed using normalized cross-correlation (NCC) [62], which is computed with the following equation:

$$NCC = \frac{1}{n}\sum_{x,y}\frac{(f(x,y)-\bar{f})(d(x,y)-\bar{d})}{\sigma_f \sigma_d} \tag{5.1}$$

where n is the number of pixels in the image, $f(x,y)$ and $d(x,y)$ are the fluoroscopic and DRR images, $\bar{f}$ and $\bar{d}$ are the mean pixel values, and σ_f and σ_d are the standard deviations, respectively [62]. DRRACC's merit function relies on a user-specified weighted, linear combination of intensity- and gradient-based NCC calculations. The following list outlines the tasks performed by DRRACC's merit function:

1. Call DRRACC's merit function with appropriate arguments
2. Compute DRRs for blue- and green-cameras using the input transformation parameters
3. Iterate through a list of active bones
4. Add the background image (e.g., composite image of all visible bones)
5. Calculate a binary image mask based on the current bone's DRR
6. Apply the mask to the current intensity-based DRR and fluoroscope image
7. Compute and store the NCC between the two images
8. Approximate the *x*- and *y*-components of the image gradient
9. Apply the mask to both the DRR and fluoroscope gradient images

10. Compute and store the NCC for both gradient components between DRR and fluoroscope
11. Apply the user-defined weighting factors to each the intensity- and gradient-based NCCs
12. Compute the final NCC as a weighted sum of the number of bones being optimized
13. Subtract the active bones from the background image
14. Return the final NCC value

In general, the intensity-based NCC outperforms gradient-based NCC – computed using finite difference methods – when large transformations are needed to successfully correlate the images. The relatively simple approach of comparing pixel intensities that are weighted by statistical parameters (e.g., mean, standard deviation, etc.) provides a robust solution when grossly aligning images [63]; that is, when the nonzero pixels in one image is compared to the nonzero pixels of another image. Figures 5.1a & 5.1b provide intensity-based DRRs (colored blue/green) superimposed over the corresponding fluoroscope images (grayscale). However, its simplicity also prevents it from arriving at more precise solutions. To perform finer adjustments, we employ the gradient-based NCC. Figures 5.1c & 5.1d illustrate the result of applying gradient-based techniques to the DRRs (colored blue/green) and fluoroscope images (grayscale). The gradient-based technique produces images such as those shown in Figures 5.1c & 5.1d. The technique identifies changes in pixel intensity and is essentially highlighting the object's boundary (edges); large pixel-to-pixel disparity is reflected by large gradient values and these regions appear as bright, thin lines within the images. The gradient-based NCC is quite sensitive to misalignment of edges between two images with large gradient, which is related to the narrow band of pixels being evaluated; NCC function gradient's on either side of the ideal registration

position are steep compared to the intensity-based NCC, enabling the gradient-based NCC to be used for fine-tuning registration.

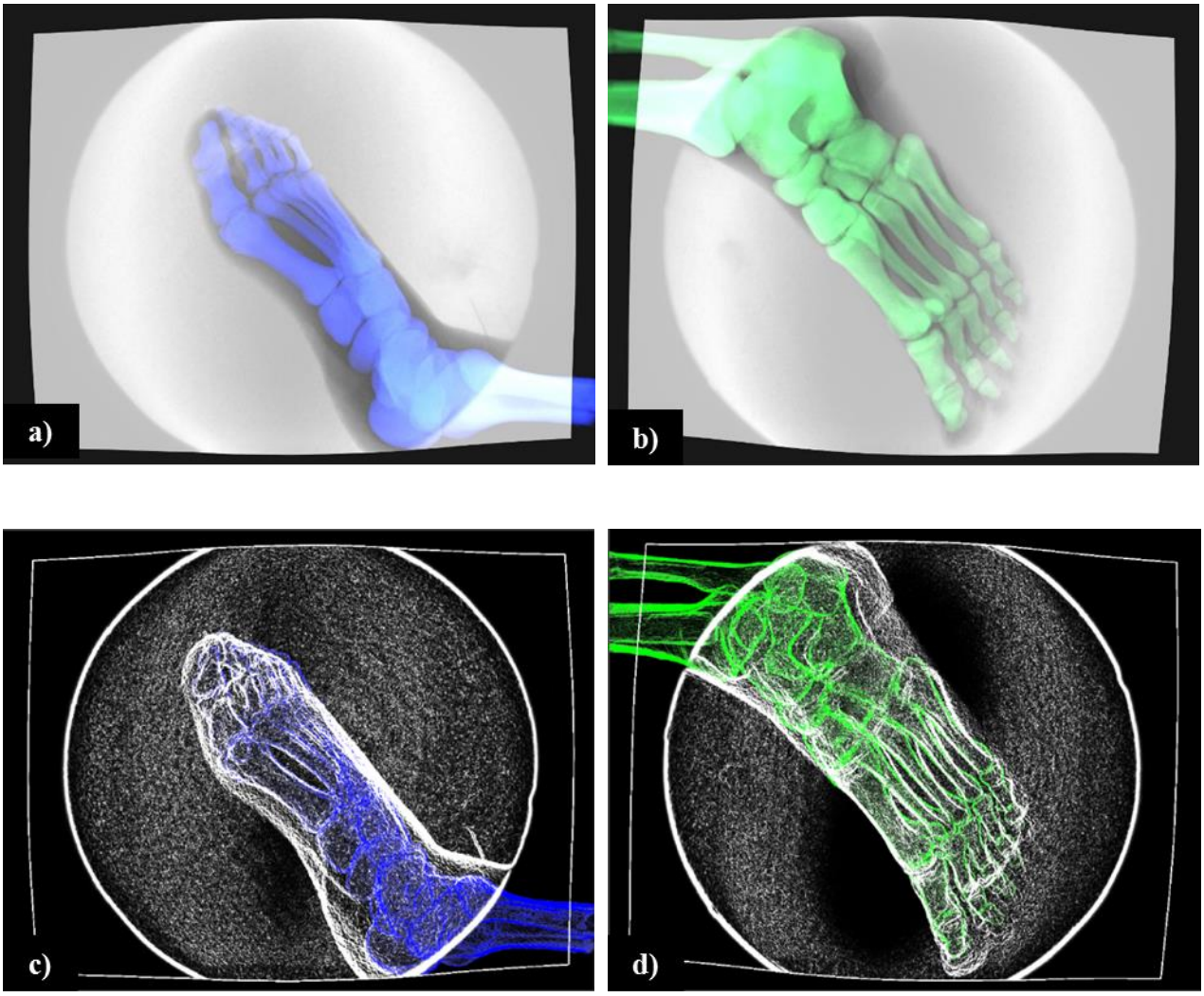


Figure 5.1. Comparison of intensity- and gradient-based images. (a & b) DRRACC generated intensity-based DRRs in blue and green superimposed atop their corresponding fluoroscope images. (c & d) Edge-based versions of (a) and (b). Recall that "blue" and "green" are used to describe the pair of imaging hardware – the use of corresponding colors in the image sequence above (and throughout this dissertation) simplifies discussion, but does not have a physical meaning.

Calculating the merit function occupies only a small fraction (~3-6%) of the total DRRACC computation time and, in fact, the merit function can be easily modified (see, e.g., the work by Haber & Modersitzki [64]). As outlined at the beginning of this chapter, the merit function is comprised of a series of subroutines used to prepare DRRs and corresponding fluoroscope

images for use in the similarity measure. Table 5.1 provides the details of DRRACC's parallelization model for evaluating the merit function:

Table 5.1. Parallelization strategy for DRRACC's merit function.

#	Merit Function Subroutine	Parallelization Strategy	Summary of Subroutine
1	Compute new DRRs using updated parameters	Kernel launch	Approximate the integral along a ray for each pixel
2	Create background image	Kernel launch	Sum DRR pixel intensities if visibility flag is 'on' and optimization flag is 'off'
3	Calculate binary mask using the DRR	Kernel launch	Set nonzero pixel values to '1' and all others to '0', and dilate or erode mask based on user-input
4	Apply mask to intensity-based DRRs and fluoroscope images	Kernel launch	Multiply image pixel values by mask pixel values
5	Compute intensity-based NCC		
5.1	Find region of interest (ROI)	Host	Single-pass over image to find min and max nonzero pixels
5.2	Crop images	Kernel launch	Write pixels contained within the ROI into a global device array
5.3	Find mean pixel intensities	Atomic summation	Sum the intensities in the global device image array
5.4	Compute intensity standard deviation	Host	Divide mean pixel intensity by the number of pixels in the ROI
5.5	Pixel-by-pixel multiplication	Kernel launch	Multiply corresponding pixels in DRR & fluoroscope
5.6	Sum inner product	Atomic summation	Sum the entries of the global device array containing the pixel-by-pixel products
5.7	Calculate correlation value	Host	Evaluate Equation 5.1 and return its output
6	Find DRR and fluoroscope image gradients	Kernel launch	Apply finite difference method to global device arrays
7	Apply mask to gradient-based DRRs and fluoroscope images	Kernel launch	Multiply image pixel values by mask pixel values
8	Repeat subroutine #4 using gradient-based images	Various	Evaluate Equation 5.1 and return its output

9	Find weighted NCC	Host	Apply user-defined weighting to intensity and gradient NCCs
10	Subtract updated DRR from background image	Kernel launch	Subtract DRR pixel intensities from background pixel intensities at each entry in the two images
11	Return weighted NCC and wait for new parameters	Host	Return weighted NCC

5.1 Intensity-based correlation

Intensity-based DRRs are generated using CUDADRR; voxel values are sampled from the 3D texture containing the CT density data and accumulated as a ray passes through the volume. The approximate line integral along a given ray provides the intensity value for a corresponding pixel in the final DRR image. The DRRs are used directly in comparison with the fluoroscope images for the intensity-based correlation.

5.2 Gradient-based correlation

Image gradients are computed in DRRACC using finite difference methods. The user is given the option to use forward difference or central difference (two- or four-term) approximations. The following equations define the specific difference operators implemented in DRRACC:

Forward Difference

$$f_x = \frac{f(i+1,j) - f(i,j)}{h} \tag{5.2}$$

Central Difference (two-term)

$$f_x = \frac{f(i+1,j) - f(i-1,j)}{2h} \tag{5.3}$$

Central Difference (four-term)

$$f_x = \frac{8 * f(i+1,j) + f(i-2,j) - f(i+2,j) - 8 * f(i-1,j)}{2h} \tag{5.4}$$

Magnitude

$$f^* = \sqrt{{f_x}^2 + {f_y}^2} \tag{5.5}$$

where f_x & f_y are the image gradients (x- and y-directions), i and j are the pixel indices, h is the step size, and f^* is the magnitude of the gradients. DRRACC's gradient computations were validated for correctness through comparison with numerically calculated gradients in each of the x- and y-directions of the input image. Equation 5.3 was used in the image sequence of Figure 5.2. Figure 5.2a reveals a simple input image (a binary annulus), with gradients shown in Figures 5.2b & 5.2c, and comparison of DRRACC-based gradients to numerically calculated gradients (Equations 5.2 through 5.5 were tested) in Figures 5.2d & 5.2e. The two images on the far right represent profile plots of the DRRACC- and numerically-calculated in-plane image gradients. Profile plots were generated by plotting the magnitude of the gradient image along a vertical line (green) produced by plotting the gradient function values along a single vertical line that intersected the gradient image.

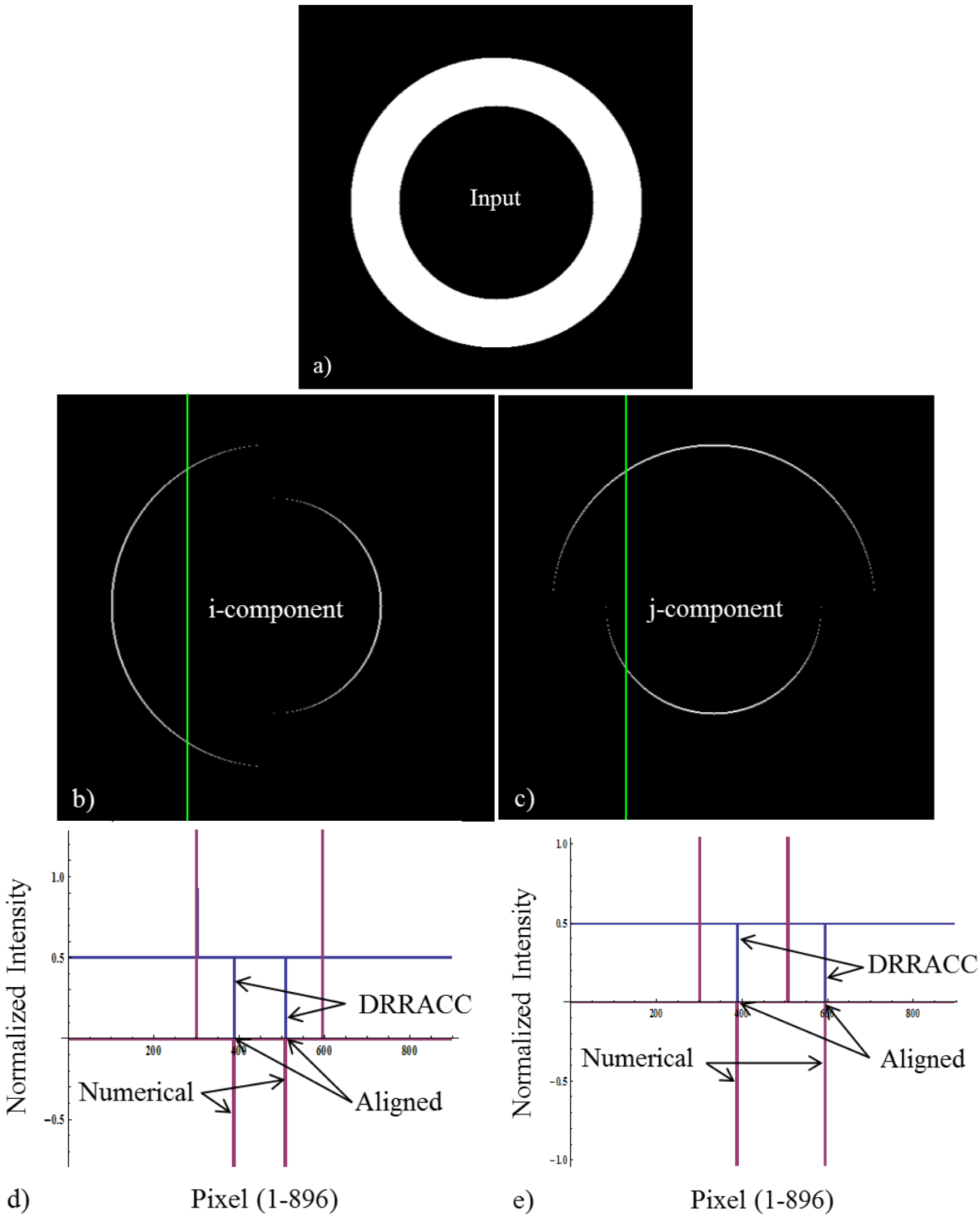


Figure 5.2. Gradient-based comparison. (a) An example input geometry. (b-c) i- and j-components of the gradient computed by DRRACC. (d-e) Profile plots of the numerically calculated input gradients versus the DRRACC generated gradients. Note that the DRRACC gradients (blue) have been offset from the x-axis to improve clarity. The sets of parallel lines in (d-e) represent profile plots along a line intersecting the gradient images. The numerically determined edges are aligned with DRRACC's edges, indicating that DRRACC's finite difference method is being implemented correctly.

The image array in Figure 5.3 demonstrates the variance between gradient components and the gradient magnitude for the calcaneus used in validating DRRACC's precision (Chapter 8). In this image suite, bone edges highlighted in bright white are locations where the gradient exhibits positive, high rates of change. Scaling applied to the images for viewing purposes increases the brightness of peaks (e.g., locations in the image with positive, high rates of change in pixel intensity) and decreases the brightness of troughs (e.g., negative, high rates of change), which appear as "shadows" in Figure 5.3. Conversely, the magnitude of the gradient is strictly positive by construction (see Equation 5.5), so applying the same scaling technique as before produces the images in the far right of Figure 5.3 that display clearly defined object edges.

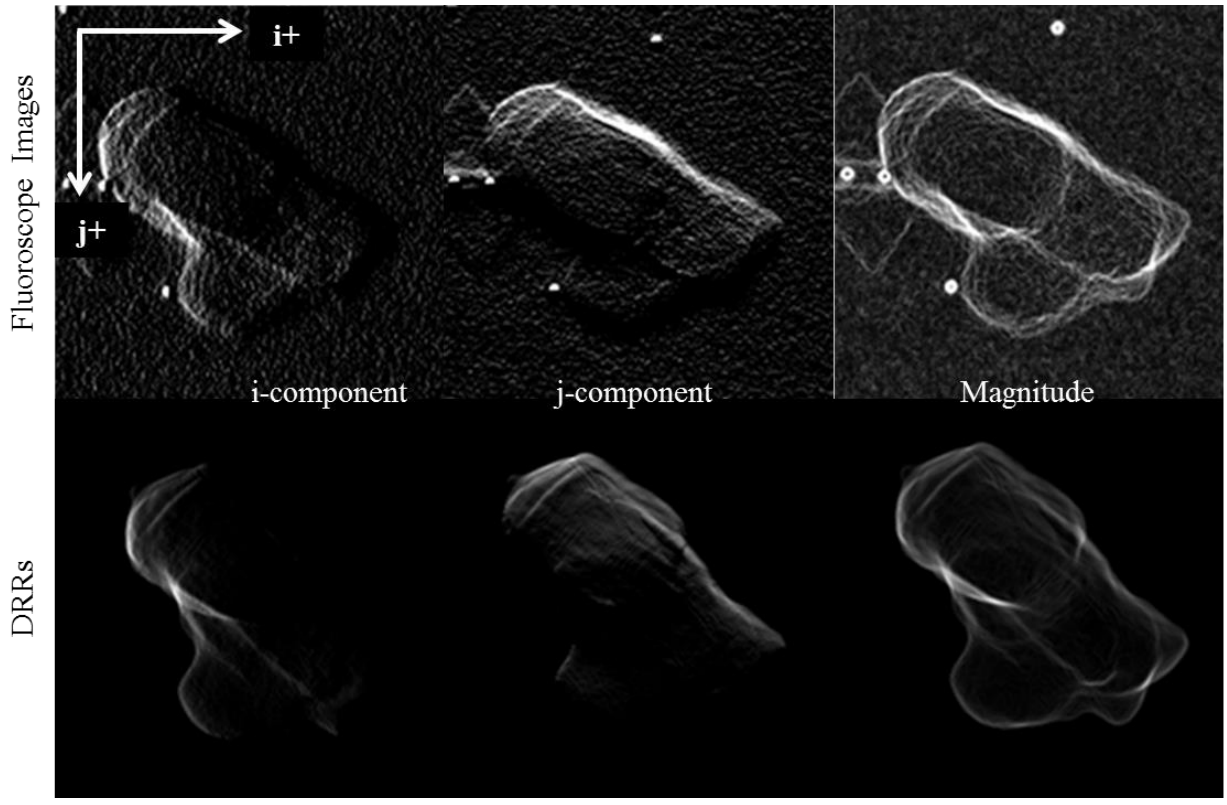


Figure 5.3. Gradient image examples. Comparison of fluoroscope image gradients (top row) to DRR image gradients (bottom row). The *i*-component (left), *j*-component (middle), and magnitude (right). Edges with positive, high rates of change in pixel intensity appear in white. Edges with negative, high rates of change in pixel intensity appear as black, shadow like regions due to scaling for viewing purposes. The magnitude is strictly positive, resulting in bright, nicely defined object edges in white.

Chapter 6: Graphical User Interface

Image-guided medical therapies and image-guided biomechanical measurement systems often combine 2D and 3D imaging modalities. Determination of relations between the 2D and 3D imaging data is known as 2D-3D registration. While fully automated 2D-3D registration is desirable, there are situations (such as creating a reasonable starting configuration for optimization, re-starting after the optimizer fails to converge, and visual verification of registration relations) when it is desirable/necessary to have a human in the loop. In this chapter, we present an OpenGL-based graphical user interface developed to integrate with the DRRACC toolkit – described in Chapter 4 – to allow the user to manipulate the kinematics of individual objects (bones) segmented from the 3D imaging, and to view the corresponding DRR and the associated correlation with a reference image in real time.

The goal of the work presented in this chapter was the implementation of a user-friendly GUI for 2D-3D registration. One of our primary objectives was to build a GUI that could display results at a rate that matches current screen refresh rates (60 Hz). This objective was largely driven by the DRRACC toolkit's ability to generate updated DRRs in <1.75 ms. A secondary objective was the development of an interface that would enable intuitive selection and 3D transformation of bones in the displayed DRR. This interface will enable human intervention during DRR optimization by simplifying the process of selecting initial set of transformation parameters for the optimizer. Without this type of visualization tool, investigators must "blindly" select a starting set of parameters for the bones, which can prevent the optimizer from converging. This type of occurrence forces the investigator to restart the optimization loop using a new set of initial parameters, potentially wasting hours of computations. If the optimizer fails to converge,

the proposed GUI would allow the investigator to reconfigure the bones on the fly without having to restart the experiment.

6.1 GUI Development Environment

DRRACC's GUI was developed in Visual Studio 2010 by employing C/C++, CUDA, and OpenGL together to create a real-time display. We used the combination of a pixel buffer object and an OpenGL 2D texture to facilitate communications between the CUDA kernel and the display. Since a kernel cannot directly operate on an OpenGL object, the pixel buffer object is *mapped* onto a pointer to a `cudaArray` that is accessible by the GPU and *bound* to a 2D texture for subsequent display. In this context, mapping is defined by the transformation of the pixel buffer object into a `cudaArray`, while binding is defined by the transfer of data in the pixel buffer object to a 2D texture [46]. (The former is more computationally efficient than the latter.)

6.2 Methods

The GUI in Figure 6.1 was built using OpenGL and interactions with DRRACC were made possible through the CUDA-OpenGL Interop API [41]. However, we were faced with the challenge that OpenGL textures cannot be written to directly by CUDA kernels. Conveniently, the CUDA-OpenGL Interop API provides the programmer with a conduit for mapping OpenGL pixel buffer objects (PBOs) to CUDA memory. Once in CUDA memory, a parallel kernel can swiftly generate an image and write it to a pointer to the mapped PBO. After the image has been generated, the PBO is unmapped from CUDA memory, allowing it to be mapped to an OpenGL 2D texture. Finally, the 2D texture containing the CUDA-modified image, in this case the DRR, can be displayed.

Historically, OpenGL display windows have used dimensions that are powers of 2 [40], which prevented us from directly rendering the 1152 x 896 pixel DRR used in this study. To overcome

this limitation, we implemented a GPU-based function that crops and pads the native image so that its dimensions are forced to be powers of 2. (It is worth noting that the function only applies to the display. The DRR itself is not affected by this crop/pad function.) Additionally, the function converts the floating point DRR pixel intensity values into 8-bit RGBA channels for display.

The OpenGL API provides tools that allow the user to interact with the display using the mouse and keyboard. The user has access via popup menu to several convenient functions that are detailed in Table 6.1. The menu can also be linked directly to the keyboard to avoid on-screen click operations. The process of reconfiguring a DRR based on an actual X-ray image begins when the user selects a bone using the menu feature. When a bone is selected, its color changes for simple on-screen identification and troubleshooting. Utilizing keyboard input, the user can then translate the bone along and rotate it about the *x*-, *y*- and *z*-axes with the bone's bounding box centroid as the center of rotation. The GUI stores the updated transformation matrix for the selected bone which is then used in subsequent DRR computations. Each update only requires that DRRACC compute a single DRR and composites the updated DRR with the full DRR (i.e., all bones in the foot), an operation that takes <1.75 ms on average[21].

[21] Depends on bone size, geometry and camera angles.

Table 6.1. Menu entries and keyboard commands available to DRRACC users.

Menu Entry / Keyboard Command	Entry Type	Function
Select bone	Submenu with list of bones	Select active bone
Deselect bone	Submenu with list of bones	Make selected bone inactive
Toggle color	Keyboard command	Toggle color/grayscale
Scale DRR intensity	Keyboard command	Increase/decrease DRR viewing intensity
Scale fluoroscope intensity	Keyboard command	Increase/decrease fluoroscope viewing intensity
Toggle fluoroscope	Menu entry/keyboard command	Toggle fluoroscope image on/off
Select translation axis	Menu entry/keyboard command	Select active translation axis
Select axis of rotation	Menu entry	Select active axis to rotate about
Return NCC	Menu entry/keyboard command	Print NCC values to command window
Run SNLP	Keyboard command	Run SNLP on active bones
Run CONDOR	Keyboard command	Run CONDOR on active bones
Run batch processing	Keyboard command	Run batch processing on gait record
Show ROI	Keyboard command	Toggle region of interest on/off
Save configuration	Menu entry/keyboard command	Save current configuration
View configuration	Menu entry/keyboard command	View saved configuration
Reset configuration	Menu entry/keyboard command	Reset configuration to initial positions
Write output	Menu entry/keyboard command	Write DRRs to file

6.3 Display

The GUI offers real-time viewing, coupled with quantitative feedback that facilitates registration of DRRs with X-ray images. The former is the ability to visualize the DRRs superimposed atop the X-ray images; this strategy allows a skilled researcher to analyze the quality of registration using visual inspection. The DRR and X-ray images are each displayed using a different RGB channel and successful alignment effectively mixes the two color channels (i.e., a blue DRR aligned with a red X-ray image will produce a purple image). The latter relies on the result of the NCC similarity measure. The user can select a region of interest (ROI) using a click-drag operation. Once the ROI has been selected, the GUI's title bar provides a real-time streaming NCC value that is updates automatically after each kinematic transformation. In theory, the user could manipulate the bones into a reasonable starting configuration with only the current NCC as feedback (this approach is somewhat impractical for live subject data, but was successful for bones scanned in space or some other medium). However, the integration of the NCC feedback with the continuous visual feedback of superimposed images renders DRRACC's high-speed GUI a powerful tool for 2D-3D registration initialization and troubleshooting. Figure 6.1

provides three different color schemes that can be switched on the fly to accommodate different data (some datasets are can be viewed more easily in white/blue/yellow, etc.). The following features can be found in Figure 6.1:

1. Simultaneous viewing of DRRs (top/bottom – colored) superimposed atop corresponding stereo image pair (top/bottom – grayscale).
2. Camera-specific regions of interest (boxes surrounding parts of the image).
3. Streaming NCC updates in the window title bar.
4. Current frame rate (fps) in the window title bar.

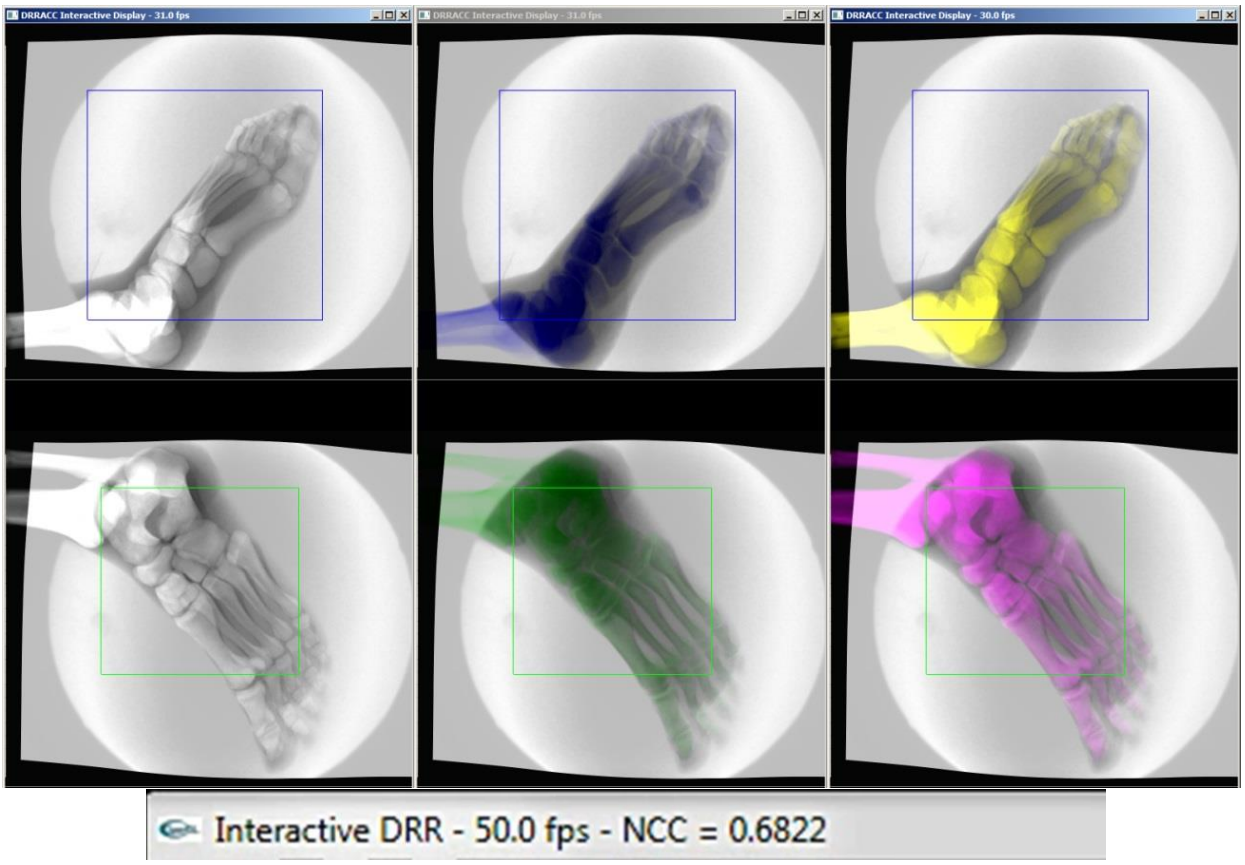


Figure 6.1. DRRACC's graphical user interface. Three examples of DRRACC's GUI are shown. The GUI I relied on to determine initial alignment of bones for subsequent registration via optimization. Different color schemes have been found to increase visibility and can help minimize eyestrain for the user during manual alignment. Note that the fluoroscope image is shown in grayscale for all three display options. DRRs are shown in white, blue/green, and yellow/purple. The image below the three GUI snapshots is an example of the real-time metrics available to the user (frame rate and NCC).

6.4 GUI Performance

Since the user interface builds upon the DRRACC toolkit, frame rates comparable to the typical monitor refresh rate of 60 Hz are possible. During static viewing, the high resolution 1024 x 1024 pixel window is displayed at ~60 frames per second, which is printed in the window's title bar. Because the toolkit can generate an updated DRR and composite it with the full DRR <1.75 ms on average, real-time kinematic transformations for individual bones are possible while maintaining ~60 fps on a high resolution display. Real-time feedback and transformation is not possible using CPUDRR, which is a consequence of the time it takes to update a DRR (~750 ms).

Chapter 7: Optimization

Now that we have detailed how DRRACC is used to generate and visualize DRRs, it is time to turn our attention to the 2D-3D registration process. Recall that a DRR is the projection of 3D CT imaging data – in a given orientation described by a transformation matrix – onto a 2D digitally simulated screen. The goal of registration is to align a pair of DRRs to the true stereo pair of radiographs by applying 3D kinematic transformations to the CT data. The metric used to quantify the quality of the registration is determined by calculating the NCC discussed in Chapter 5. If the fit between the DRRs and fluoroscope images is not sufficient to meet the registration criteria, we must derive a new transformation matrix and repeat the process. As you can imagine, testing an infinite number of transformations to yield the best registration is simply not realistic, which is why image registration in DRRACC is carried out using one of two integrated optimizer suites: SNLP [65] and CONDOR [59].

The role of the optimizer is to characterize the parameter space – in this case, defined by a translation and a rotation along and about each of the three principal axes, respectively, for a total of six parameters per bone – and find the optimum value of a merit function described by the quality of registration between the DRRs and fluoroscope images [66]. Characterization of the parameter space is accomplished by modeling the space through simultaneous perturbation of the optimization parameters.

There are many different types of optimization routines, but most can be loosely classified as direct search or gradient-based [66]. Direct search methods approach the optimization problem by characterizing merit function values across the parameter space directly through user-controlled perturbation of the parameters. Conversely, gradient-based methods characterize the parameter space by building a model using the partial derivatives of the parameters [66]. Both

techniques are widely used and the choice often depends on the type and complexity of the optimization problem.

The complexity of optimization during 2D-3D image registration is strongly related to the types and severity of noise found in the inputs. Noise can take on many forms, but the primary types that will be discussed in this section include the following:

- **System noise:** This type of noise can manifest as X-rays from other sources, incorrectly calibrated X-ray systems, loss of fidelity during data processing, thermal effects on sensors and circuitry, heterogeneity of emitters and/or sensors, vibrations during image capture, etc.
- **Background noise:** Soft tissues (e.g., skin, fat, ligaments, tendons, muscle and cartilage) and overlapping bones found in the fluoroscope images are considered to be background noise and form what is commonly referred to as a *noise floor* as depicted in Figure 7.1c/d.
- **Round-off error:** The loss of data correctness as a result of approximate rather than exact representation of a number (e.g., in single-precision, the value of π is represented by 3.1415927, but the exact value is 3.14159**65358979323846…**).

By the nature of their creation, digitally reconstructed radiographs are relatively noiseless. The primary cause of noise in the DRR is the numerical estimate of an integral along a ray through an interpolated function that is used to compute a pixel's intensity. Round-off error is also a source of noise in the DRR, but its effects are generally considered minimal. In contrast, fluoroscope images are littered with noise that manifests itself as system noise, background noise and round-off error introduced during bias and distortion correction. The extreme differences in image quality tends produce a challenging merit function for the optimizer, which can cause sporadic convergence to non-optimal solutions. Figures 7.1a & 7.1b represent a typical stereo pair of

fluoroscope images with a DRR of the hallux superimposed shown in black. The profile plots in Figures 7.1c & 7.1d correspond to pixel intensities extracted from both the DRRs and fluoroscope images along the white lines intersecting Figures 7.1a & 7.1b.

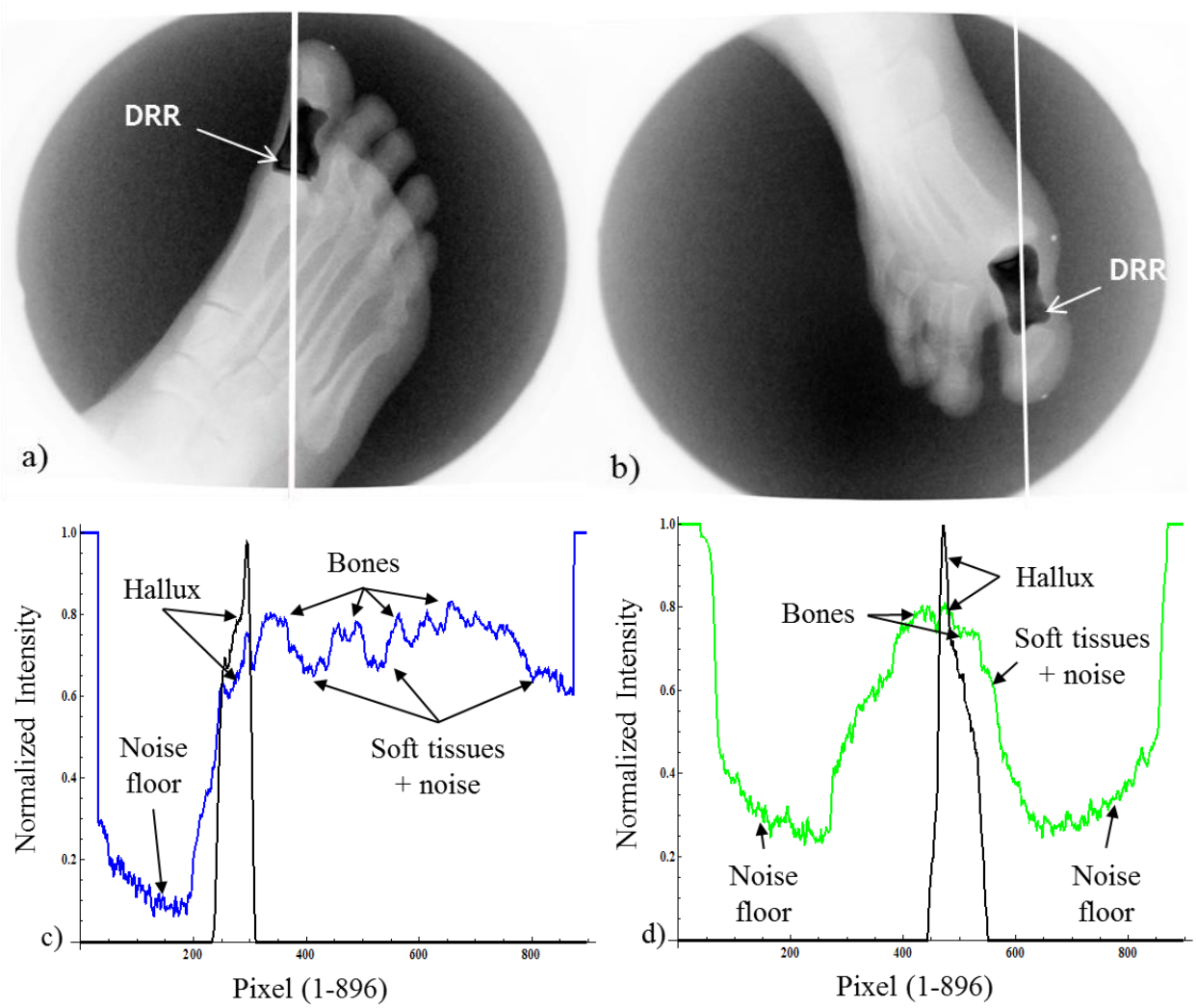


Figure 7.1. Live subject profile plots. (a-b) Blue- and green-camera CUDADRRs of the hallux (in black) superimposed blue and green fluoroscope images, respectively. (c-d) Profile plots taken along the red line in (a-b). The black curves in (c-d) represent the pixel intensities along the red lines in (a-b) for the DRRs. Similarly, the blue and green curves are pixel intensities for the fluoroscope images. Note the drastic difference in overall shape between the black curves and the blue/green curves. The differences are mainly due to the presence of soft tissues, other bones and noise in the fluoroscope images, and in part to finite integration and interpolation in the DRRs[22].

[22] Note while Figure 7.1 presents a seemingly impossible problem to solve, the CPUDRR code has achieved successful registration with the presented data.

The stark contrast between the DRR (represented by a single, black, impulse-like spike in 7.1c & 7.1d) and the corresponding fluoroscopes (represented by the curve that starts and ends with a normalized pixel intensity of 1.0) is dominated by background noise from soft tissues and other bones. It was postulated that the presence of background noise in the fluoroscope image would raise challenges for the optimizer, which was driven by the fact that inclusion of the background noise in NCC computations for the fluoroscope would misrepresent the true measure of registration; the DRR has zero-value pixels surrounding the bone, so ROI mean pixel value and standard deviation for each will vary depending on noise levels.

The optimization process within DRRACC is detailed in Figure 7.2. A bone or set of bones are selected using DRRACC's OpenGL GUI (see Chapter 6). The user can opt to define a region of interest or let DRRACC automatically detect a region of interest in each DRR – for each bone – that will act as a bound for image correlation; only pixels located within the ROI bounding region are considered during the comparison (all pixels located outside of the ROI are neglected). Once selected, the user-specified optimizer (SNLP or CONDOR) is called and during optimizer initialization, a background image is generated that is comprised of all bones not chosen for optimization. The background image plays an important role in reducing the computation time associated with evaluation of the merit function by eliminating the need to composite each of the 28 single bone DRRs at each of the 300-500 evaluations of the merit function per frame. Relative optimization timings with and without the inclusion of this composition strategy are shown in Table 7.1. The values surrounded by red boxes in Table 7.1 reveal the importance of employing this background composition strategy; a 47% reduction in computational execution time can be realized.

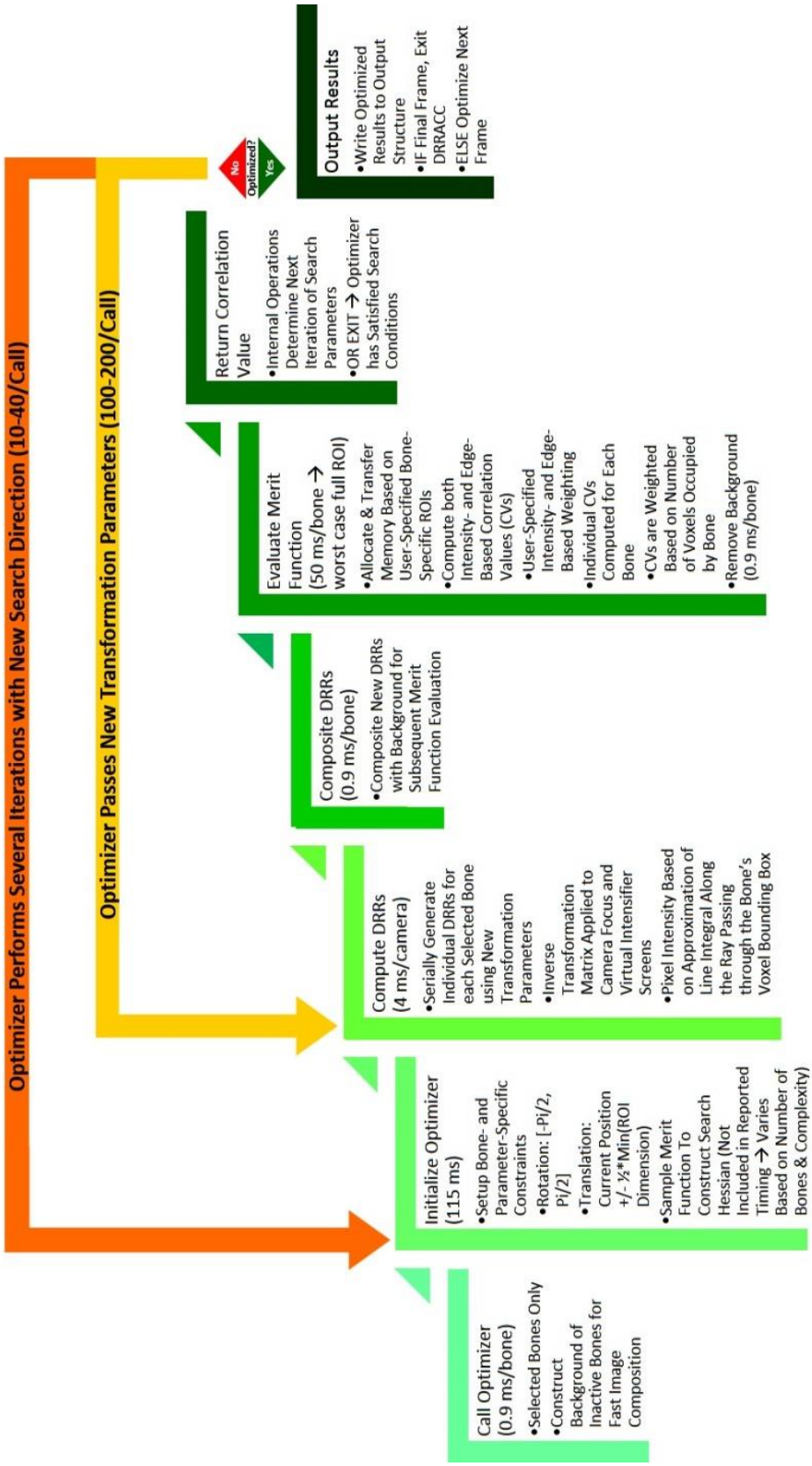


Figure 7.2. An optimization loop using SNLP, with a description of operations and associated timings.

Table 7.1. Comparison of relative merit function evaluation time (averaged over 25 runs) for optimization with and without the background image strategy. Note the large improvement in execution time when using the background image strategy as a result of a reduced number of image compositions.

	Number of Merit f'n Evaluations	Total Time (ms)	Time per evaluation (ms)
	Without Background Image		
Min	34	1590.8	32.73
Max	167	7126.2	51.86
Mean	97.5	4108.5	42.62
Standard Deviation	35.4	1499.6	5.190
	With Background Image*		
Min	25	784.7	16.18
Max	205	5152.5	37.59
Mean	96.8	2155.1	22.63
Standard Deviation	47.7	1117.1	5.437
	Difference		
Min	-26%	-51%	-51%
Max	23%	-28%	-28%
Mean	-1%	-48%	-47%
Standard Deviation	35%	-26%	5%

* This strategy requires that a background image be computed upon entering the optimizer provided the bone to be optimized is not the same as the bone that was just previously optimized. If so, the previous background image is preserved and the process continues.

7.1 SNLP

The SNLP suite is a sequential quadratic programming algorithm that minimizes the L_1 (or L_∞ depending on the problem) exact penalty function [65]. The algorithm determines a search direction by constructing the gradient of the merit function, takes a step, tests conditions and backward tracks until the conditions are satisfied. SNLP also has a pattern search option that

provides a much higher resolution search, but at the expense of an order of magnitude increase in overall execution time. The following text is adapted from [67]:

"SNLP consists of two sequential quadratic programming (SQP) algorithms for the solution of nonlinear programming problems. The nonlinear programming (NLP) problem:

$$\min_{x \in R^{n_x}} \{ f(x) | h_j(x) = 0, j \in \varepsilon, g_i(x) \leq 0, i \in I \} \tag{7.1}$$

where $x \in R^{n_x}$ is the set of parameters, $f(x)$ is a scalar cost function, $h_j(x)$, $j \in \varepsilon$, $\varepsilon = \{1, 2, \ldots, n_h\}$, $n_h \leq n_x$, is a set of equality constraints, and $g_i(x)$, $i \in I$, $I = \{1, 2, \ldots, n_g\}$ is a set of inequality constraints. Our solution strategy is based on replacing (7.1) with a problem that minimizes an unconstrained exact penalty function. Here we will employ the L_∞ exact penalty function, and the L_1 exact penalty function to solve the nonlinear programming problem.

Specifically, the NLP problem is solved via (7.1) the L_∞ exact penalty function minimization problem:

$$\min_{x \in R^{n_x}} \theta_\infty(x, \bar{\rho}) = f(x) + \bar{\rho} \max\{0, |h_j(x)|, g_i(x)\}, j \in \varepsilon, i \in I \tag{7.2}$$

where $\bar{\rho} > 0$ is a suitably chosen scalar penalty function weight; or (II) the L_1 exact penalty function minimization problem:

$$\min_{x \in R^{n_x}} \theta_1(x, \rho) = f(x) + \sum_{j \in \varepsilon} \rho_j |h_j(x)| \sum_{j \in I} \rho_{j+nk} \max\{0, g_j(x)\} \tag{7.3}$$

where $\rho_j > 0$, $j = 1, 2, \ldots, n_h + n_g$, are a suitably chosen scalar penalty function weights. The motivation for using (7.2) or (7.3) to solve (7.1) is based on the fact that, under certain conditions, a local minimum of $L_\infty P$ or $L_1 P$ is also a local minimum of (7.1). The idea of solving NLP via $L_\infty P$ or $L_1 P$ has a rich history, and some very useful techniques have appeared in the literature. A feature that distinguishes these algorithms from similar methods that have appeared in the literature is that we use relaxed strictly convex quadratic programming (QP) sub-problems to determine search directions in the minimization procedure. The relaxed QP problems used here are feasible even in cases where the constraints are inconsistent.

It is well known that, for some problems, the use of exact penalty functions prevents the SQP method from achieving a superlinear rate of convergence. More precisely, these algorithms are unable to accept full step sizes in the vicinity of the solution. This phenomenon is known as the 'Maratos effect'. In the algorithm developed here, the Maratos effect is avoided by using second-order corrections."

-B. Fabien, *linfsqp and l1sqp: The implementation of two algorithms for the solution of nonlinear programming problems* [67]

7.2 CONDOR

In contrast, the CONDOR suite is a direct optimization tool, based on Powell's UOBYQA [68] that builds a quadratic model based solely on the merit function, not its derivatives [59]. CONDOR first constructs a local quadratic model around the initial input, finds the local minimum, rebuilds the model and repeats this process until the conditions are satisfied. The following text is adapted from [69]:

*"CONDOR, short for **CO**nstrained, **N**on-linear, **D**irect, parallel, multi-objective **O**ptimization using trust **R**egion method for high-computing load, noisy objective functions. The aim of the CONDOR optimizer is to find the minimum* $x^* \in \mathbb{R}^n$ *of an objective function* $f(x) \in \mathbb{R}$ *using the least number of function evaluations. It is assumed that the dominant computing cost of the optimization process is the time needed to evaluate the objective function* $f(x)$*. The algorithm will try to minimize the number of evaluations of* $f(x)$*, at the cost of a huge amount of routine work.*

The algorithms implemented are Dennis-Moré Trust Region steps calculation (it's a restricted Newton's Step), Sequential Quadratic Programming (SQP), Quadratic Programming (QP), Second Order Corrections steps (SOC), Constrained Step length computation using L_1 *merit function and Wolf condition, Active Set method for active constraints identification, BFGS update, Multivariate Lagrange Polynomial Interpolation, Cholesky factorization, QR factorization and more!*

Many ideas implemented inside CONDOR are from Powell's UOBYQA (Unconstrained Optimization BY quadratic approximation) [70] for unconstrained, direct optimization. The main contribution of Powell is equation 7.4 that allows us to construct a full quadratic model of the objective function in very few function evaluations (at a low price).

Powell's heuristic

$$\frac{M}{6}\left\|x_{(j)}-x_{(k)}\right\|^3 \max_d\left\{\left|P_j\left(x_{(k)}+d\right)\right|:\|d\|\le\rho\right\}\le\epsilon \qquad j=1,\ldots,N \tag{7.4}$$

Like most optimization algorithms, CONDOR uses, as local model, a polynomial of degree two. There are several techniques to build this quadratic. CONDOR uses multivariate Lagrange interpolation technique to build its model. This technique is particularly well-suited when the dimension of the search space is low (n < 100).”

-F. Vanden Berghen, CONDOR User’s Guide [69]

7.3 Methods

7.3.1 System & Image Noise

Neither SNLP nor CONDOR is well suited to handle noisy input to the merit function. An initial look at the fluoroscope imagery such as that shown in Figure 7.3 did not appear to be a concern. However, the intensity in Figure 7.3 (upper left) was scaled to prevent oversaturation of the image. If the fluoroscope image’s pixel intensity is scaled to allow for color saturation (Figure 7.3 – middle), it becomes apparent that there is in fact nontrivial system noise in the image (system noise is defined at the beginning of the chapter). We anticipated that fluoroscope images would have additional pixel contributions in the form of soft tissues (skin, tendons, ligaments, muscle, etc.) that were not present in the DRRs due to the nature of their formation, but the high level of biplane system noise was unexpected.

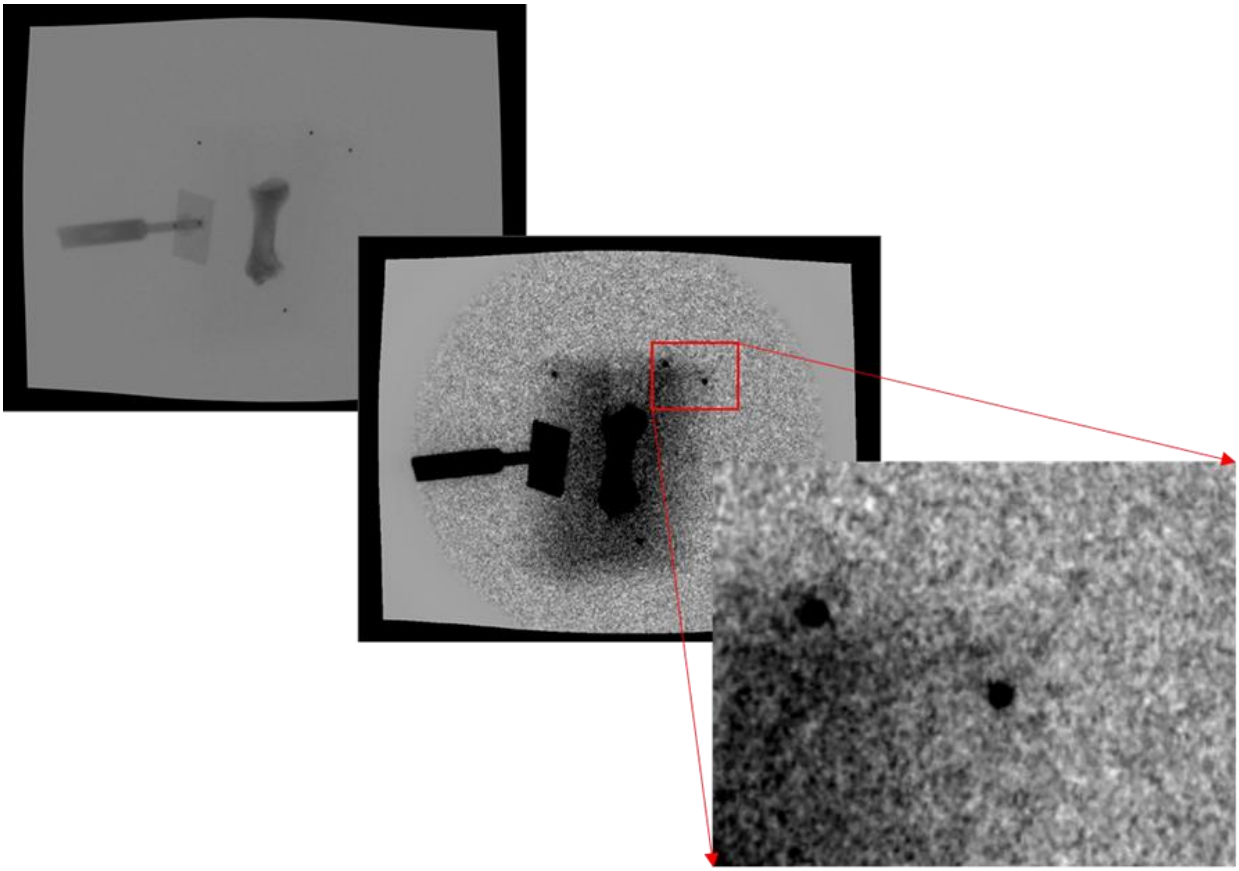


Figure 7.3. Fluoroscope image noise. (Upper left) A typical fluoroscope image used to validate the system and processing software. The image appears to be relatively noiseless and uniform, however simple scaling reveals the true level of image noise shown in the middle and lower right figures. This type of noise can severely affect performance by complicating registration for the optimizer due to the presence of numerous local optima.

CUDADRR computes the intensity value for each pixel in the DRR by digital approximation of an integral along a ray through the 3D data. The approximation introduces errors at each sampled point that does not lie on a grid point, because interpolation must be used to determine the corresponding value; however, the interpolation could be considered as an averaging procedure that is reducing system noise. Another potential source of noise is the number of integration steps taken through a bone volume's bounding box (denoted by *'n'* in this section). Upon entering the bounding box, the ray accumulates voxel densities at each of its *n* steps through the bounding box and the final pixel value – for a particular ray – is the sum of the interpolated voxel densities

(Figure 7.4). Equal steps are taken along the ray which is achieved by subdividing the ray into equal parts from bounding box entry to exit.

The parameter *n* is responsible for the number of steps taken through the bounding box and therefore controls the integration resolution, which directly affects the resolution of the DRR. Low values of *n* could potentially amplify system noise by increasing the amount of interpolation required during sampling. Single-parameter transformation studies were performed by sampling the merit function while perturbing single transformation parameters as *n* varied (50, 100, 200, and 300). The perturbation of a translation parameter in the DRR transformation was compared to a synthetic fluoroscope image for correlation to eliminate image noise from the system. (By construction, the synthetic fluoroscope image exactly matches the DRR to within machine precision.)

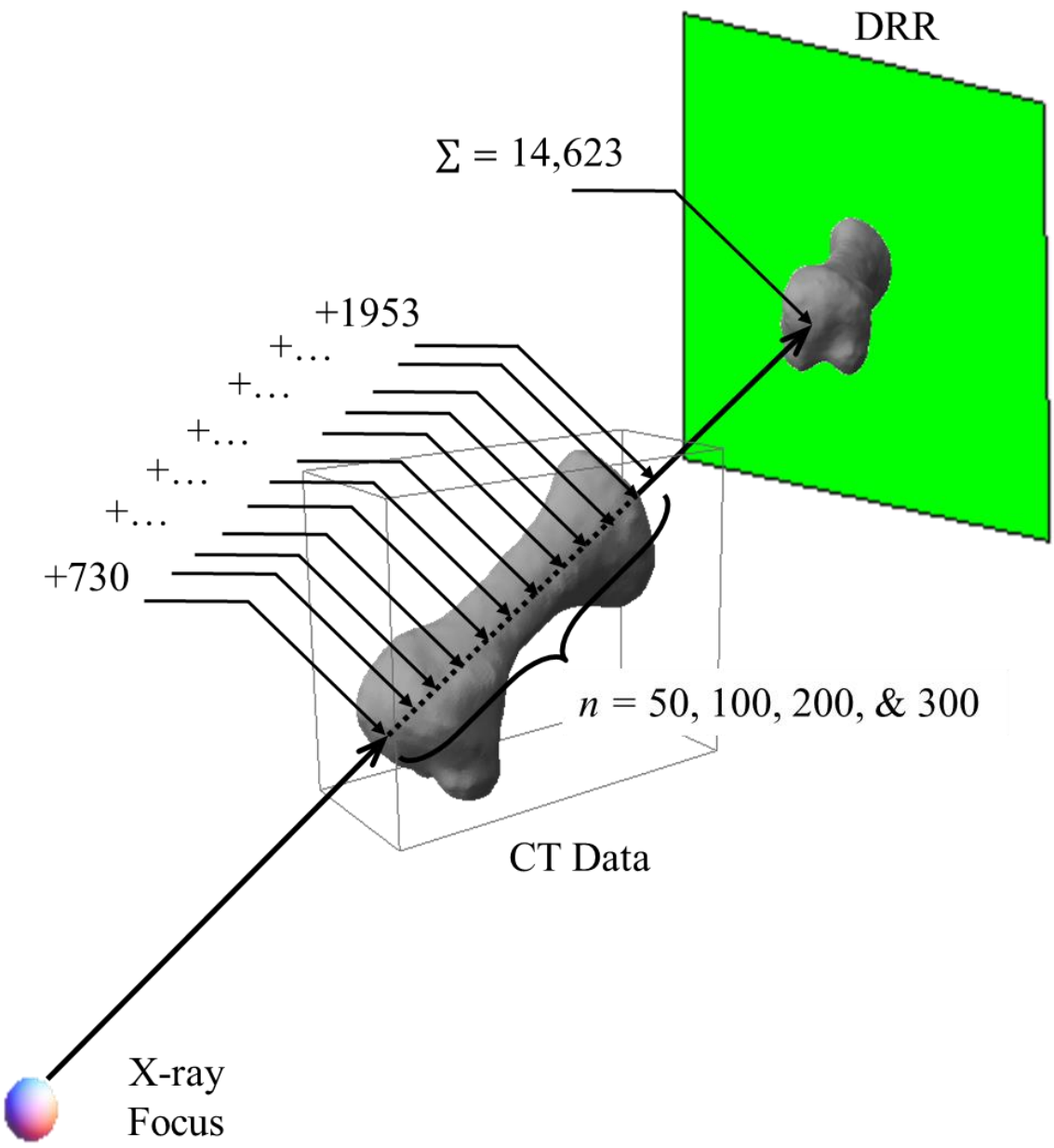


Figure 7.4. Visualization of integration steps through a bone's bounding box.

7.3.2 Parameter Sensitivity

To test parameter sensitivity, a synthetic reference image was created for use as the comparison image, allowing for NCC values to reach unity[23]. We performed a series of single bone

[23] In practice, X-ray images contain both system and background noise that prevents the NCC from reaching unity.

transformations to profile the NCC as a function of translation along two axes and as a function of rotation about two axes. The bone was transformed into 400 configurations and the NCC was recorded at each step. Sample locations were achieved by incrementing parameter values by 0.5 units (mm or degrees) over the range from [-5, 4], with '0' being the starting configuration. DRRs were generated for each sampled location and the NCC was calculated.

7.4 Results

7.4.1 System & Image Noise

To assess the impact of fluoroscopic noise on the NCC, five of the six transformation parameters were held constant while the sixth, a translation parameter, was perturbed from its initial position on both sides. The bone was translated by ±8 mm and the NCC was computed at each location using a step size (sampling resolution) of 0.01 mm. At first glance, NCC as a function of a single transformation parameter appeared to be a relatively smooth and well-behaved function (Figure 7.5a), however further inspection revealed nontrivial occurrences of noise across the function (Figure 7.5b). The particular merit function used in this example was chosen because it provides a diverse collection of features, including a global maxima and dozens of local maxima and minima, resulting in an extremely challenging single degree-of-freedom optimization problem.

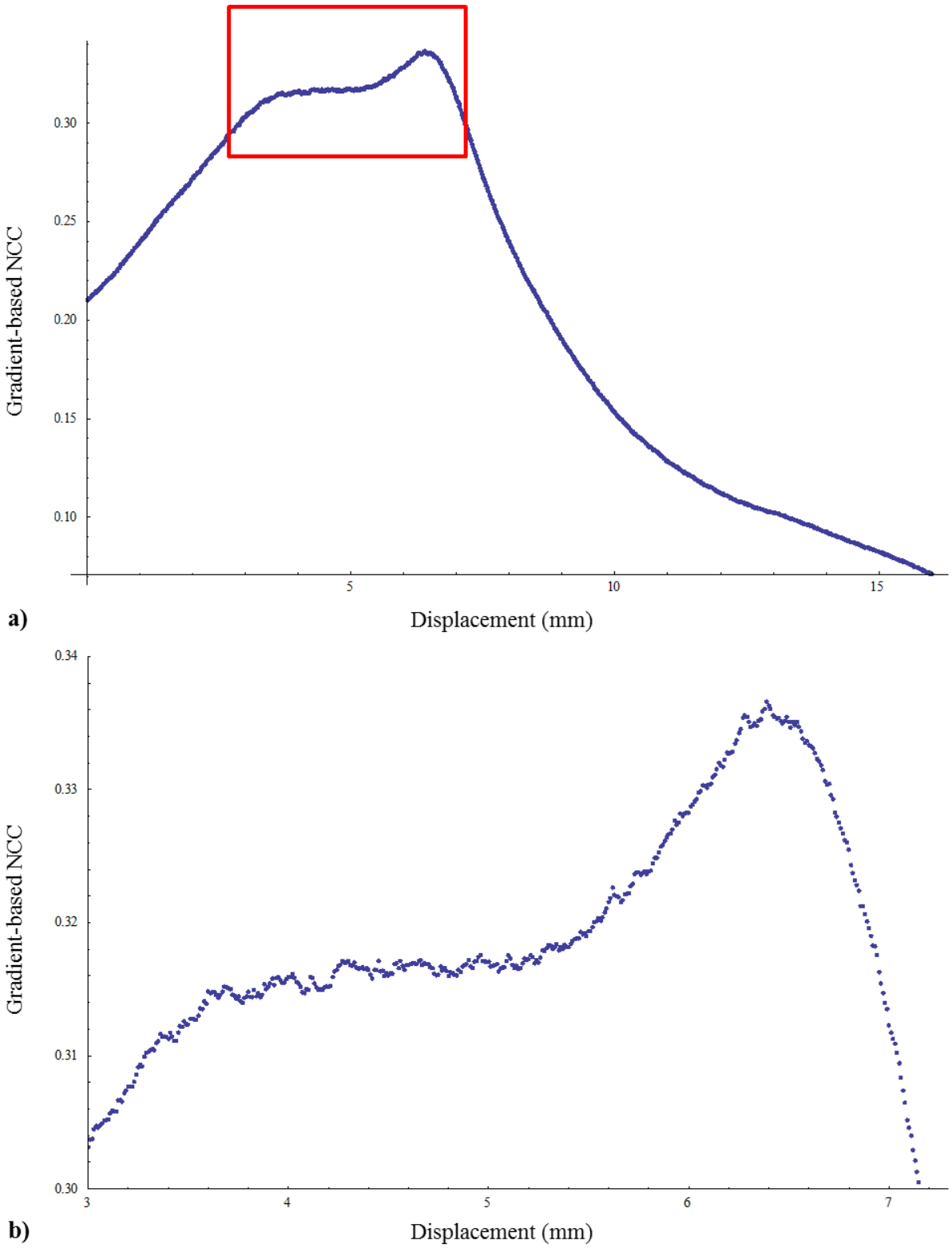
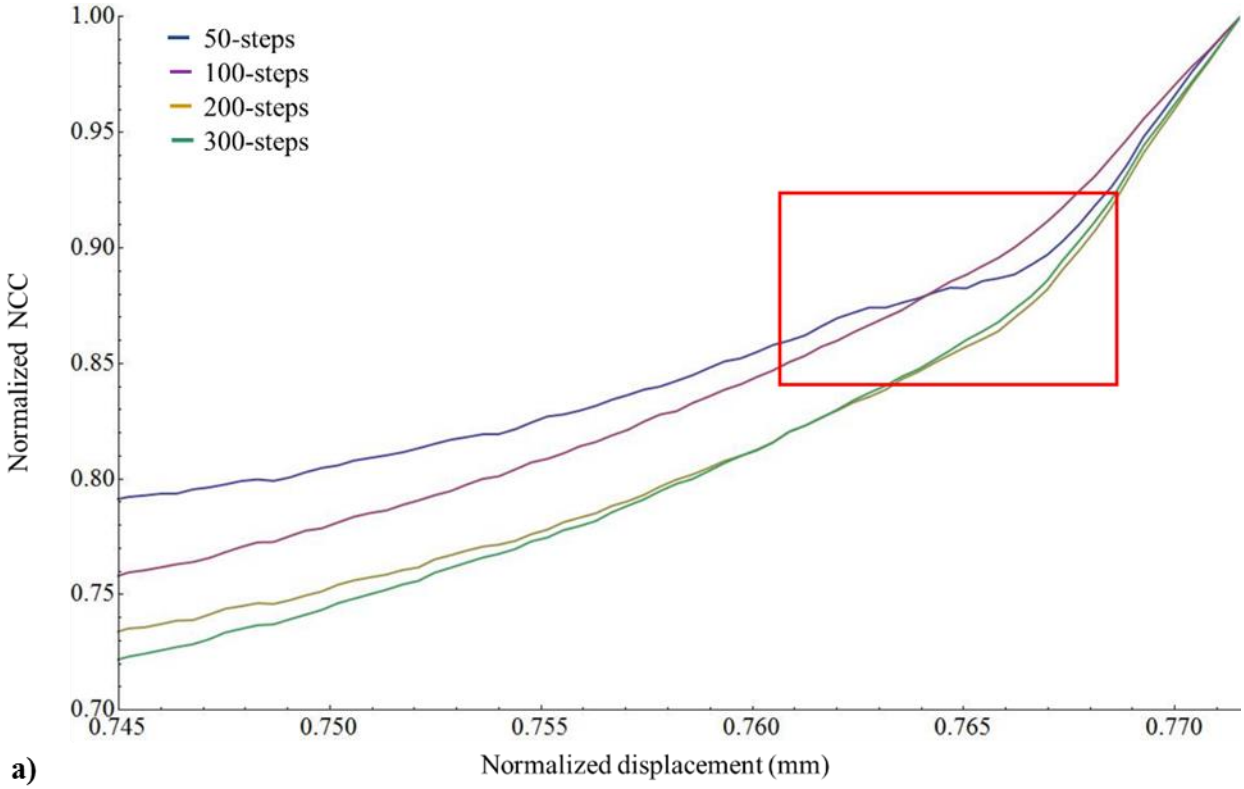
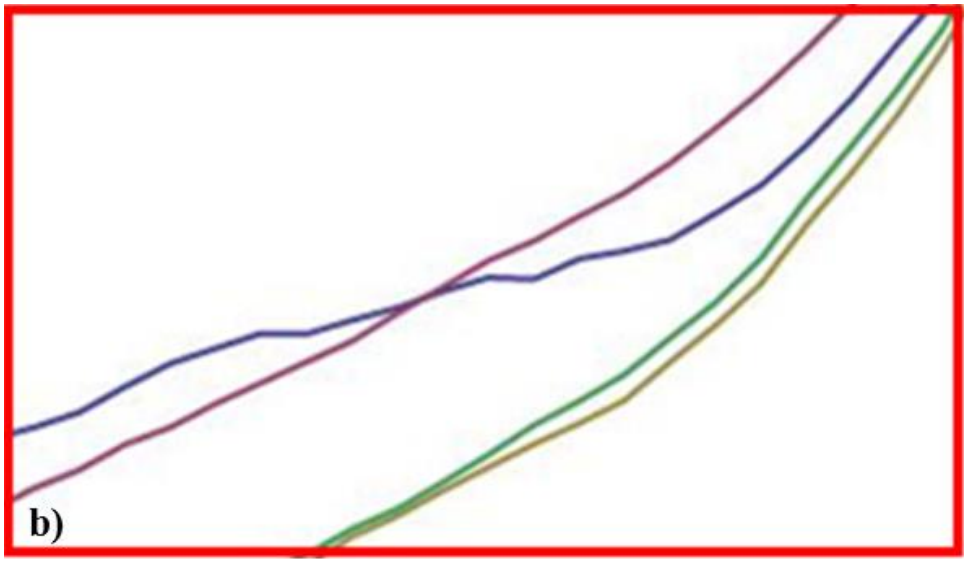


Figure 7.5. (a) Intensity-based NCC as a function (dimensionless) of a single translation parameter (millimeters). The curve appears smooth, but closer inspection in (b) reveals a noisy and non-differentiable function. The numerous local maxima and minima pose a challenge for the optimizers.

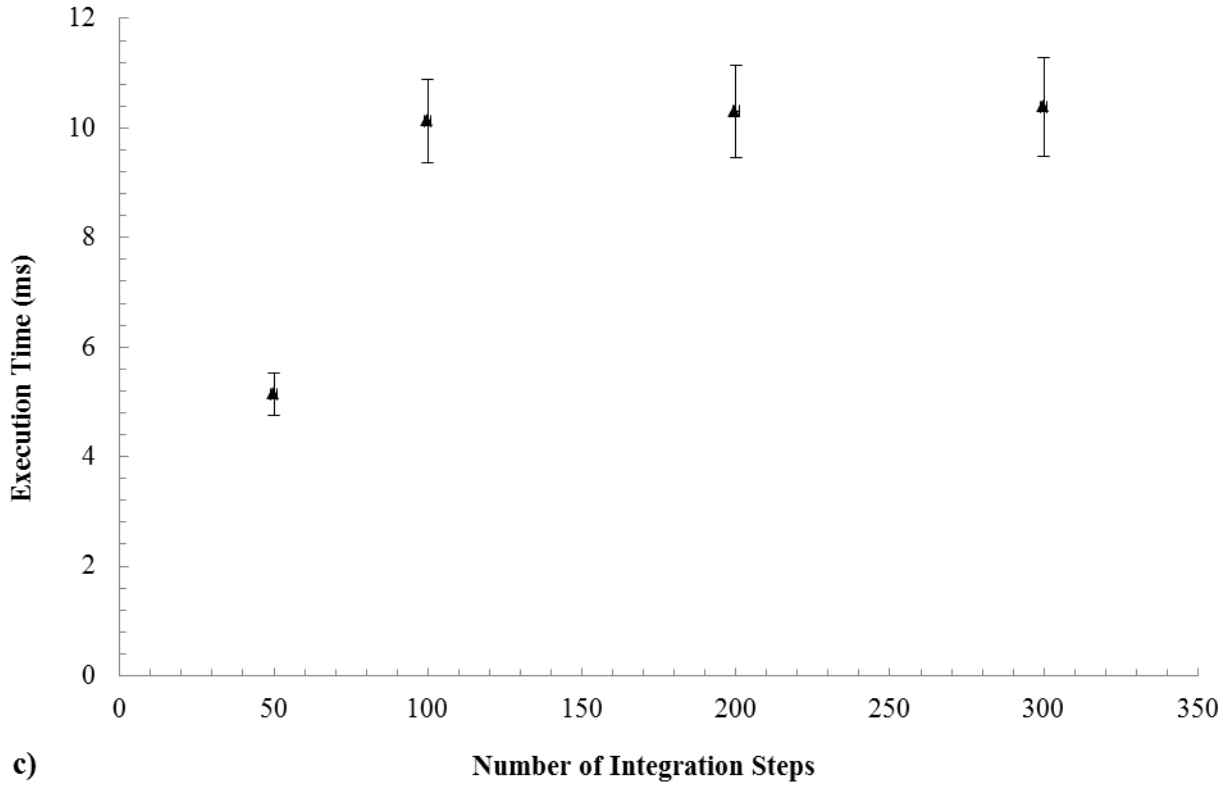
Inspection of plots in Figures 7.6a & 7.6b reveal that the difference in NCC noise is insignificant for the cases $n = 200$ and $n = 300$. The $n = 50$ and $n = 100$ cases. A marked difference in the appearance of function noise is highlighted with a red rectangle in Figure 7.6a. A value of $n = 300$ was chosen based on the convergence of function behavior for $n \geq 200$ and was used for all validation cases discussed in Chapter 8. Figure 7.6c provides DRR generation execution times with respect to the number of integration steps.



a)



b)



c)

Figure 7.6. NCC characterization. (a) NCC as a function of a translation parameter and the number of integration steps used during DRR formation. (b) An expanded view of the crop region in (a) that demonstrates NCC sensitivity as a function of integration steps (50, 100, 200, 300 steps). (c) DRR execution time as a function of the number of integration steps.

During optimization, merit function sampling step size – that is, the amount the optimizer perturbs the input parameters – plays a key role in determining whether the optimizer will converge on an ideal solution. If the optimizer is using too small of a step size and the merit function behaves as it does in Figure 7.5a, it is very likely that the optimizer will choose a set of function parameters that produce a solution that lies on a local maxima/minima. This type of behavior is illustrated in Figure 7.7b.

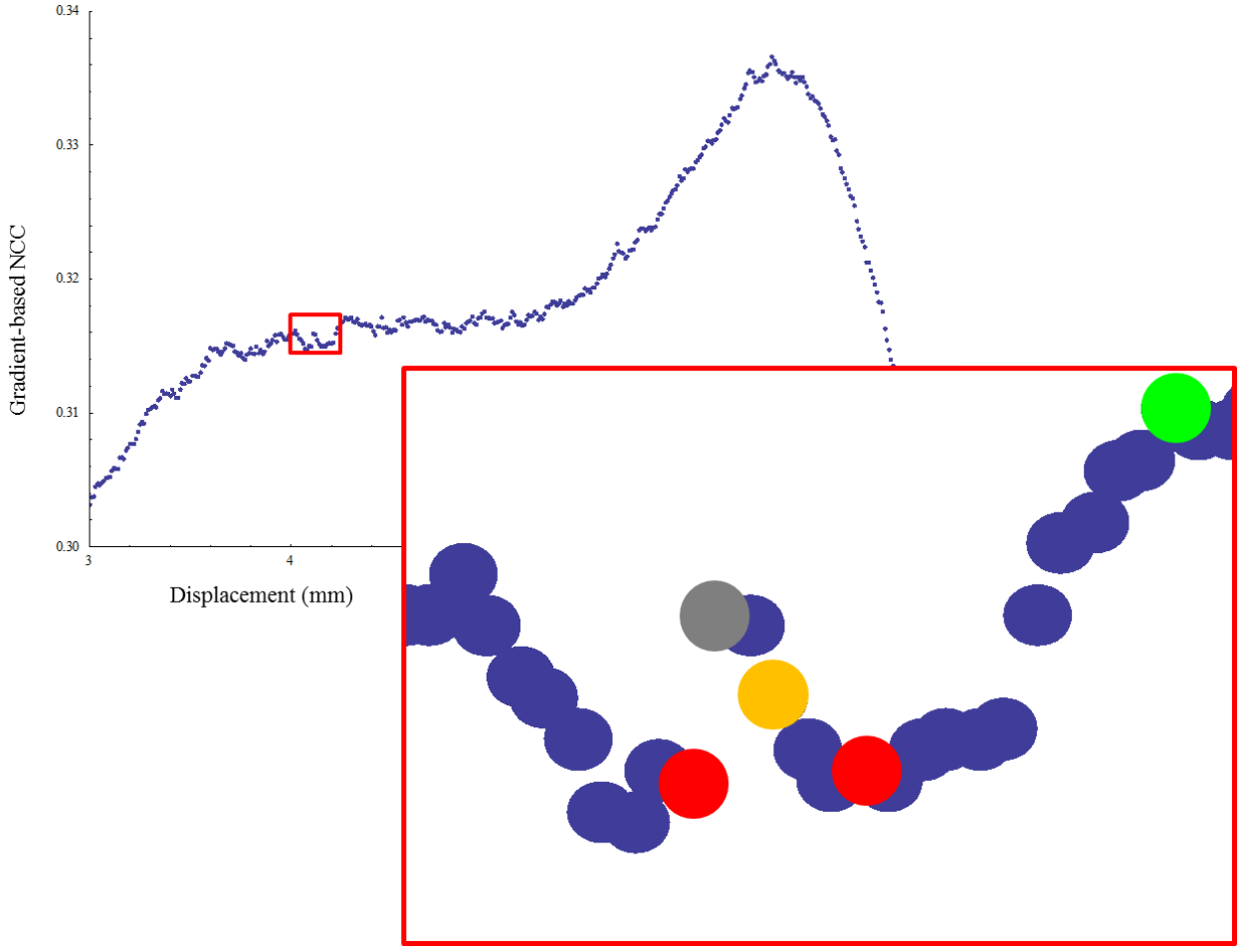


Figure 7.7. Example of falsely identified function maximum during optimization. We revisit the merit function from Figure 7.5 (upper left) and zoom in on the plot in the region within the red rectangle. All of the data points in the expanded view, regardless of color, were determined by sampling DRRACC's merit function. The orange data point represents the initial starting position. Here, the optimizer is testing the function by incrementing the initial position by ± a user-defined step size. Sampling the function at either red data point returns a lower correlation value than the starting position and so the optimizer reduces the step size and re-samples the function. The optimizer would falsely identify the sample location corresponding to the gray data point as the ideal, even though the location of the green data point is the true solution.

Both SNLP and CONDOR have several user-accessible optimization parameters that are set prior to initiating DRRACC. A list of the optimization parameters can be found in Table 7.1. The default values were either 1) recommended by the optimizer documentation or 2) determined via adaptive trial and error.

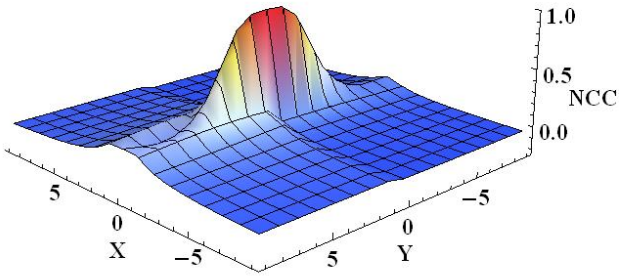
Table 7.2. Parameters for the SNLP and CONDOR optimizer suites and their default values used in DRRACC.

SNLP Parameters		
Parameter	**Description**	**Default Value**
showProgress	Print optimizer status to command window	No
tolerance	Tolerance to satisify first-order necessary conditions	1.00E-06
lineSearch	Perform a simple back-tracking line search	Yes
patternSearch	Perform a simple pattern search	No
patternSearchTol	Pattern search tolerance	1.00E-06
patternIterations	Maximum number of pattern search iterations	3
patternDelta	Pattern search initial step size	1.00E-05
snlpIters	Maximum number of optimizer iterations	500
showFunctionGrad	Print merit function gradient to command window	No
showConstraintGrad	Print constraint gradient to command window	No
constraints_R	Rotational constraints	10 degrees
constraints_T	Translational constraints	15 mm
gradientStepSize	Initial gradient step size	7.50E-03
CONDOR Parameters		
Parameter	**Description**	**Default Value**
rhoBegin	Initial distance between sample sites	2.0
rhoFinal	Final distance between samples sites (stopping criteria)	1.00E-04
condorIters	Maximum number of optimizer iterations	500
upperBounds	Constraint upper bounds	10 degrees; 15 mm
lowerBounds	Constraint lower bounds	-10 degrees; -15 mm

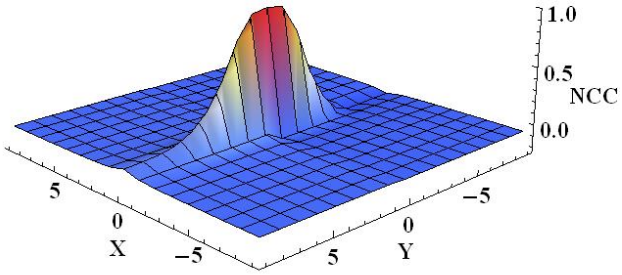
7.4.2 Parameter Sensitivity

The two-parameter translation study revealed large, relatively flat regions surrounding the narrow band of parameter values that would provide meaningful results during optimization. The narrow band is located along the line $x = 0$ in Figures 7.8a, 7.8b and 7.8d, and along the line $y = 0$ in Figures 7.8c, 7.8e and 7.8f. Within these narrow bands of nonzero NCC values, the optimal DRR configuration is located at an NCC value of 1.0 for parameter values of (0, 0), which is

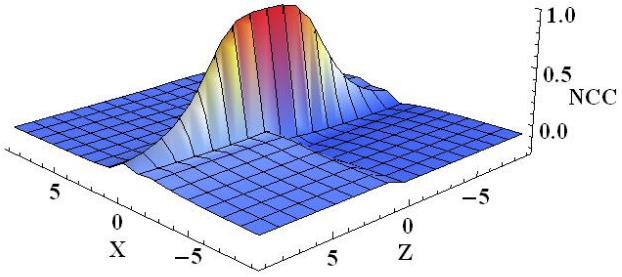
located at the center of each plot. The presence of the large, flat regions – e.g., locations that are not along the line $x = 0$ for Figures 7.8a/b/d or $y = 0$ for Figures 7.8c/e/f – increase the complexity of optimization. This result implies that convergence to the optimal position may require careful alignment of the bones prior to engaging the optimizer. This implication supports the use of the positioning GUI described in the previous chapter by providing real-time visual and quantitative feedback during manual alignment.



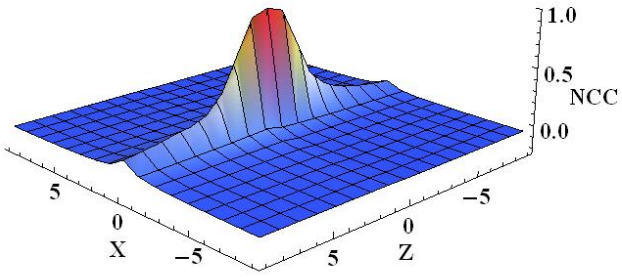
a)



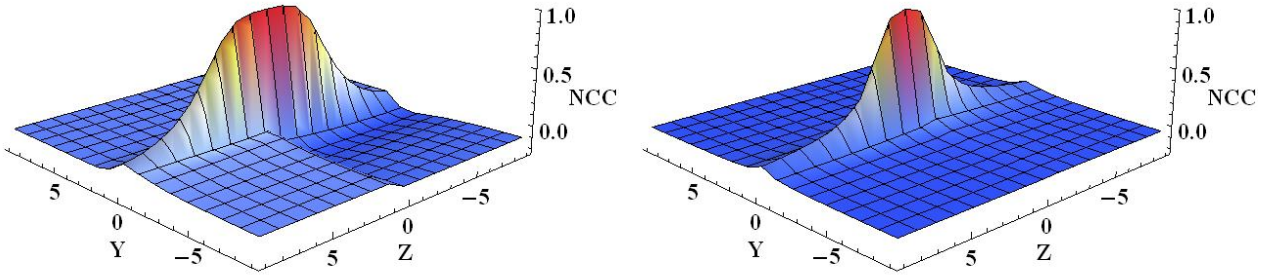
d)



b)



e)



c) f)

Figure 7.8. Two-parameter NCC characterization. Plots of the translation (a-c) and rotation studies (d-f). a) xy, b) xz, c) yz, d) xy, e) xz, f) yz.

Chapter 8: Validation of DRRACC

DRRACC and CPUDRR were developed independently to enable simultaneous validation of the two software toolkits, however, this chapter will focus on introducing and discussing the methods used to validate DRRACC for image accuracy, intensity, saturation, blur, distortion and functionality. The following list provides a brief mention of validation techniques used to evaluate DRRACC:

1. Image formation – used to validate DRRACC's image formation model to ensure that object location in CUDADRRs was correct.
2. Pixel-by-pixel intensity comparison – used to validate DRRACC's image formation model to ensure that pixel intensities were computed accurately.
3. Image linearity – used to compare DRR and fluoroscope intensity to identify any nonlinearities in the images (i.e., over- or under-saturation).
4. Image Geometry – used to validate DRRACC's image formation model to ensure that blur and distortion were not present in DRRs.
5. Application to real imaging data:
    a. Stage translation – used to quantify DRRACC's ability to track pure translation.
    b. Stage rotation – used to quantify DRRACC's ability to track pure rotation.
6. Performance – used to quantify DRRACC's computational efficiency and compare it to CPUDRR execution times.

8.1 Methods

8.1.1 Image formation

DRRACC's parallel approach to image formation (described in Chapter 4) was evaluated via comparative analysis of fluoroscope images, CUDADRRs, CPUDRRs and a Mathematica implementation for DRRs (synDRRs) to validate DRR correctness in terms of object location within the image. The physical biplane fluoroscopy system calibration is carried out using a 3D (calibration) block created from a stable radiolucent polymer (R1/HG3000, GoldenWest MFG., Inc.; Cedar Ridge, CA) [55]. The calibration block has 15 radiodense tantalum beads of varying diameters permanently seated within it at known locations. True bead centroids and diameters were determined to within 0.007 mm using a coordinate measuring machine (CMM, Global Performance Model, Hexagon Metrology; North Kingstown, RI) [55].

The biplane system was used to capture the calibration block as a stereo pair of fluoroscope images (Figures 8.1a & 8.1b) and the in-plane coordinates of the bead centroids were determined using an in-house edge-detector implemented in the CPUDRR software. Both CPUDRR and DRRACC were employed to generate independent pairs DRRs corresponding to a CT scan of the calibration block. The edge-detector script was applied to the CPUDRRs to find the bone centroids.

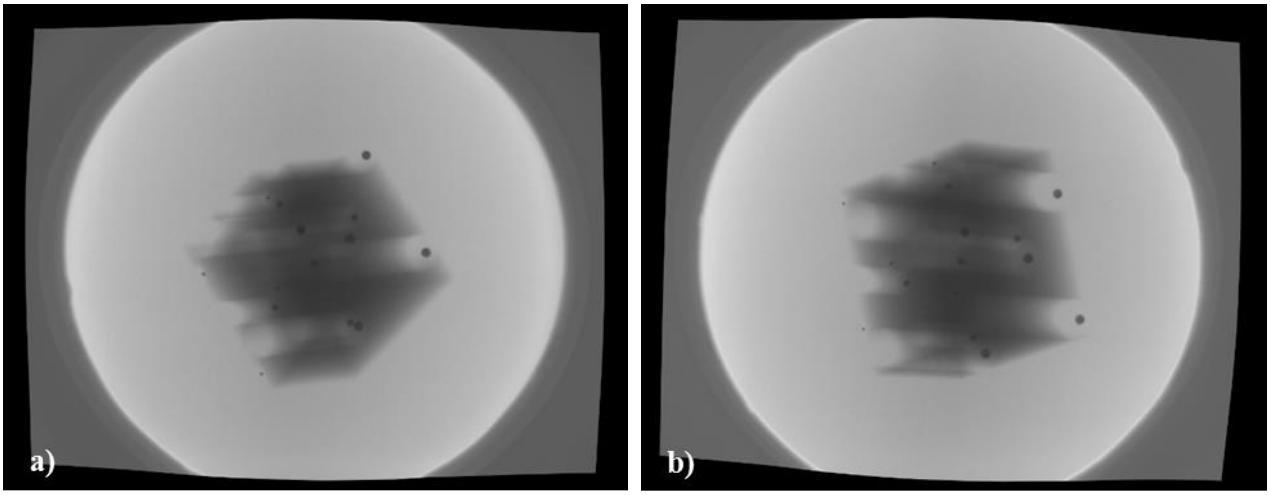


Figure 8.1. Calibration block fluoroscope images. (a) Blue and (b) green fluoroscope images of the 3D calibration block containing 15 tantalum beads.

CUDADRRs were imported to Mathematica (Wolfram) [71] and bead centroid locations were approximated by visually identifying the pixel closest to the center of the bead. Next, synthetic DRRs (synDRRs) of the calibration block were created in Mathematica by constructing a digital model of the biplane system using the same input passed into both CPUDRR and DRRACC (Figure 8.2). This approach provided a secondary means for validating DRRACC's image formation method; if the CUDADRRs and synDRRs varied by more than round-off error, one or both of the two systems would be at fault of generating incorrect DRRs and further debugging would be needed to isolate the issue. On the contrary, if both sets of DRRs were in good agreement, it would be highly unlikely that both image formation methods would exhibit the same errata, thus validating the formation methods against each other.

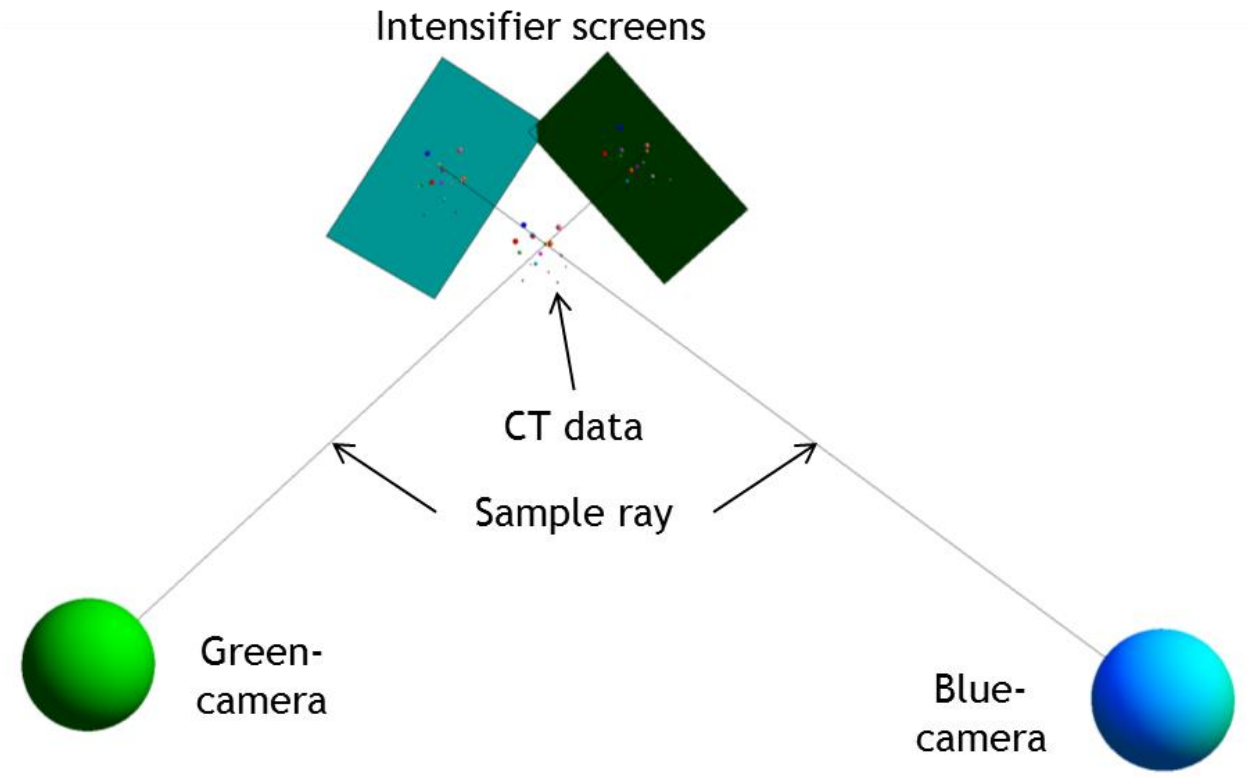


Figure 8.2. 3D model of the biplane system constructed in Mathematica.

After the synDRRs were generated, the in-plane coordinates of the bead centroids were identified with the same method used for the CUDADRRs. At this point, four sets of image pairs prepared for the image formation validation study: 1) fluoroscopes, 2) CPUDRRs, 3) CUDADRRs and 4) synDRRs. Image formation validation was performed by computing the Root Mean Square Error (RMSE) between the locations of the bead centroids for the following cases:

1. CPUDRRs vs. fluoroscopes – validate CPU[24]
2. CUDADRRs vs. fluoroscopes – validate DRRACC

[24] It should be noted that the biplane fluoroscopy system used by our group has been thoroughly validated; the primary purpose of this trial was to validate DRRACC's image formation process against the CPU's. However, the fluoroscope images also undergo both bias and distortion correction, and so this experiment served a dual purpose in confirming that bias and distortion correction were being applied properly. Bias and distortion correction are discussed in more detail later in this chapter.

3. CUDADRRs vs. synDRRs – secondary validation of DRRACC

The RMS errors for the 3D Euclidean distance between the bead centroids were computed using the following equation:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(d_i - f_i)^2} \tag{8.1}$$

where n is the number of beads, and d_i & f_i are the scalar coordinates of the bead centroids in the DRR and fluoroscope, respectively (results can be found in Section 8.2.1).

8.1.2 Pixel-by-Pixel Intensity Comparison

Recall that DRRACC computes the intensity of a given pixel in a DRR by accumulating the values stored in voxels corresponding to a segmented object defined by a 3D density file derived from CT imaging data. It was hypothesized that problems related to DRRACC's method would manifest in subsequent DRRs in the form of incorrect pixel intensities. To validate the accuracy of CUDADRR intensity, a study was performed on a synthetic CT dataset to compare pixel-by-pixel intensity with CPUDRRs.

The synthetic CT data was created by first forming a binary label file containing a set of five nonzero "plus signs" interspersed within a 3D array of zeros (Figure 8.3). (Note that the "plus signs" were chosen as a simple geometry possessing multiple faces and edges.) The label file was then applied to a 3D array of random floating point numbers to generate a synthetic

Figure 8.3. Geometry used in synthetic CT data.

density file. By construction, the synthetic CT data was devoid of background noise (soft tissues and other artifacts), which enabled a more direct comparison between CUDADRRs and CPUDRRs. Both DRRACC and CPUDRR were employed to produce DRRs of the synthetic CT data using the Identity matrix as the input transformation; a pixel-by-pixel intensity comparison ensued.

8.1.3 Image Linearity

Fluoroscope images undergo pre-processing via in-house software developed in MATLAB [55] to correct the pincushion effect and magnetic lens distortion [56, 72]. The software also applies a bias correction to account for variable pixel saturation across the fluoroscope image due to X-ray beam distribution [56, 72]. It was postulated that fluoroscope images were being subjected to appropriate bias- and distortion-correction via the in-house software.

Validation of the bias- and distortion-corrections was accomplished through a comparison of CPUDRRs and CUDADRRs to fluoroscope images. Scatter plots of fluoroscope versus DRR pixel intensity for both CPUDRRs and CUDADRRs were created to evaluate whether the relation between pixel intensity was linear or nonlinear. A nonlinear relation would indicate that the biplane system was over- or under-saturating pixel intensity and that the correction software was not properly accounting for distortions. DRRs were aligned using manual positioning followed by invocation of the optimizer for final registration. DRR intensities were plotted on the x-axis and fluoroscope intensities were plotted on the y-axis.

8.1.4 Image Geometry

It was necessary to confirm that CUDADRRs displayed the correct geometry, e.g., without distortion or blur. DRRACC was used to create CUDADRRs and register them via optimization to a corresponding pair of fluoroscope images. Validation of image geometry was performed by

isolating a single column of pixel intensities for both the fluoroscope images (Figures 8.4a & 8.4b) and the CUDADRRs. The profile plots could then be analyzed to identify whether distortion or blur was present in either the fluoroscope images or CUDADRRs.

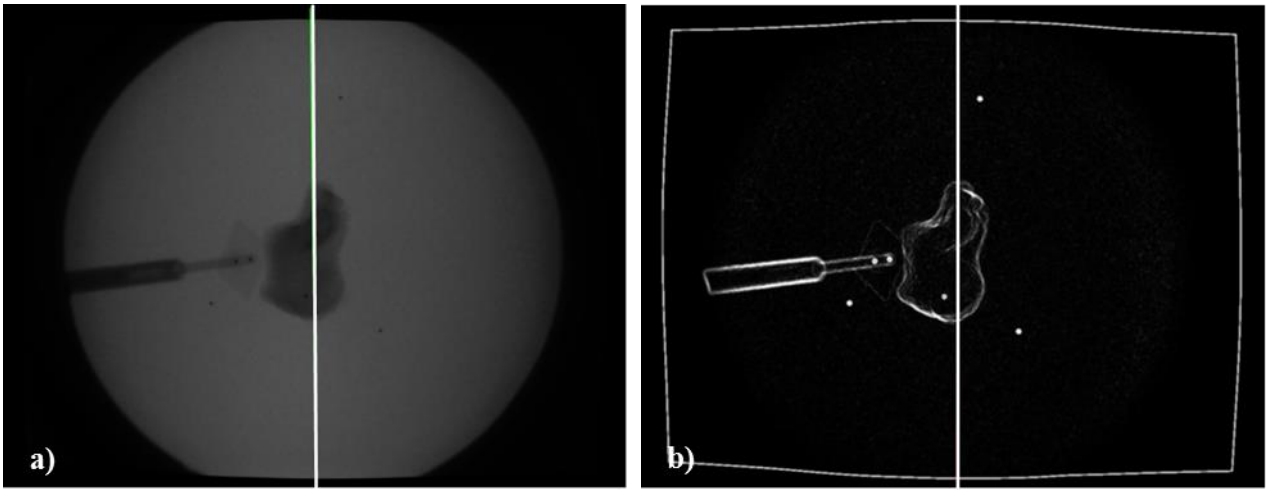


Figure 8.4. Intensity versus gradient images. (a) A representative intensity-based fluoroscope image used to create profile plots. (b) The gradient magnitude of the image in (a). The white line in (a) & (b) represent the sample location for extracting data to construct a profile plot. Note that the images are not actually rectangular, which is due to the image intensifier shape.

The Multi-Rigid software package is used to segment bone from soft tissues in the CT volume by employing a complex algorithm involving graph cuts and level sets [57]. The software assigns label values to voxels based on the results of the segmentation process. Amidst this process, the user can define a level-set parameter that dilates (or erodes) the zero level set (i.e., the surface) of the segmented bone volume thereby increasing (or decreasing) the bone's size as represented by the label file. The effects of the tunable level set parameter became evident when CUDADRRs were generated using an incorrectly sized label file (i.e., too few voxels); the fluoroscopic images appeared to be larger than the aligned DRRs (Figures 8.5a & 8.5b). The excessive appearance of red surrounding the circled bone in Figures 8.5a & 8.5b suggested that the label file being used to represent the segmented CT volume was excessively eroded by the segmentation software. Profile plots for five different dilations were created using intensity-based images.

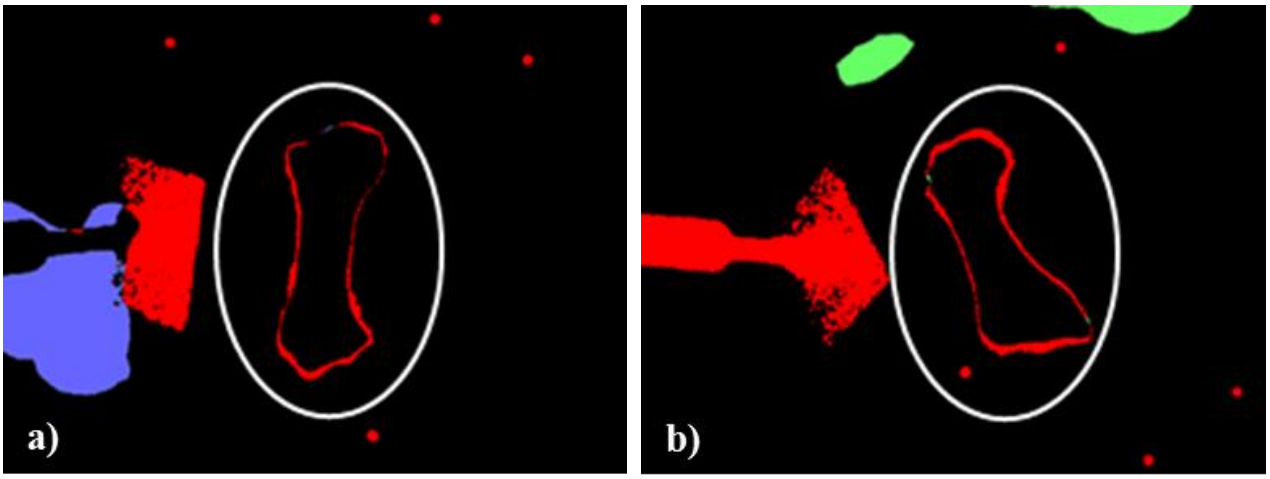


Figure 8.5. Superimposed images of DRRs with corresponding fluoroscopes. Objects unique to the fluoroscope are shown in red and objects unique to the DRRs shown in blue/green. (a) Blue-camera, (b) green-camera. This image processing technique allows for visualization of size variations between the fluoroscope images and DRRs. Ideally, images will have similar quantities of red and blue/green outlining the bone of interest.

8.1.5 Stage translation

A first metatarsal was encased in high density radiolucent foam (Figure 8.6a) and attached to a wand (Figure 8.6b). Radiodense tantalum beads were embedded into the foam to enable marker-based validation of the biplane fluoroscope system and CPU processing software [55]. The wand was translated using a 1-micron precision stepper-motor for static translation validation [55]. The bone underwent pure translation using a linear stage adjusted by a micrometer, and fluoroscopic images were captured at the following increments (mm): 0, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 3, 4, 5, 10 and 15. DRRACC was employed to compute DRRs at each stage interval. RMS error was determined by comparing the true stage position to the Euclidean distance of the three absolute translations found by registering the DRRs to the corresponding fluoroscope images. These data were also compared to the CPU results for the same study.

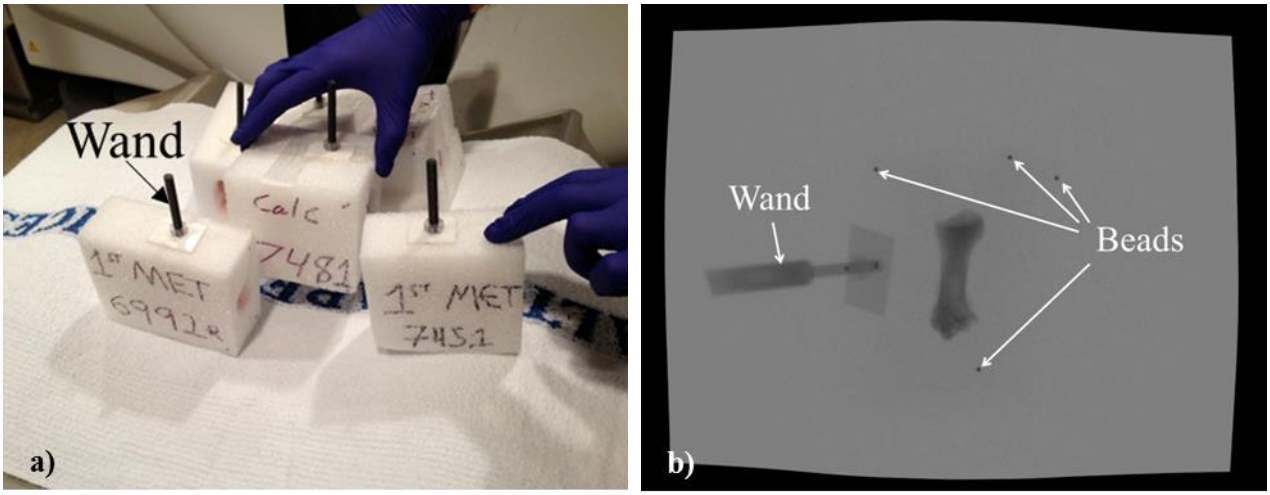


Figure 8.6. Bone preparation for stage validation studies. (a) Preparing the sample bones by encasing them in a high density radiolucent foam and attaching a wand for precision transformation via a 1-micron stepper-motor [56]. (b) An example of a fluoroscope image for one of the bones in (a).

8.1.6 Stage rotation

The calcaneus in Figure 8.7 was encased in high density radiolucent foam and attached to a wand. The wand was rotated using a 1-micron precision stepper-motor for static rotation validation [56]. The bone was subjected to pure rotation and fluoroscopic images were captured at the following increments (degrees): 0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 3, 4, 5 and 15. DRRACC was employed to compute DRRs at each stage interval, similar to the translation study.

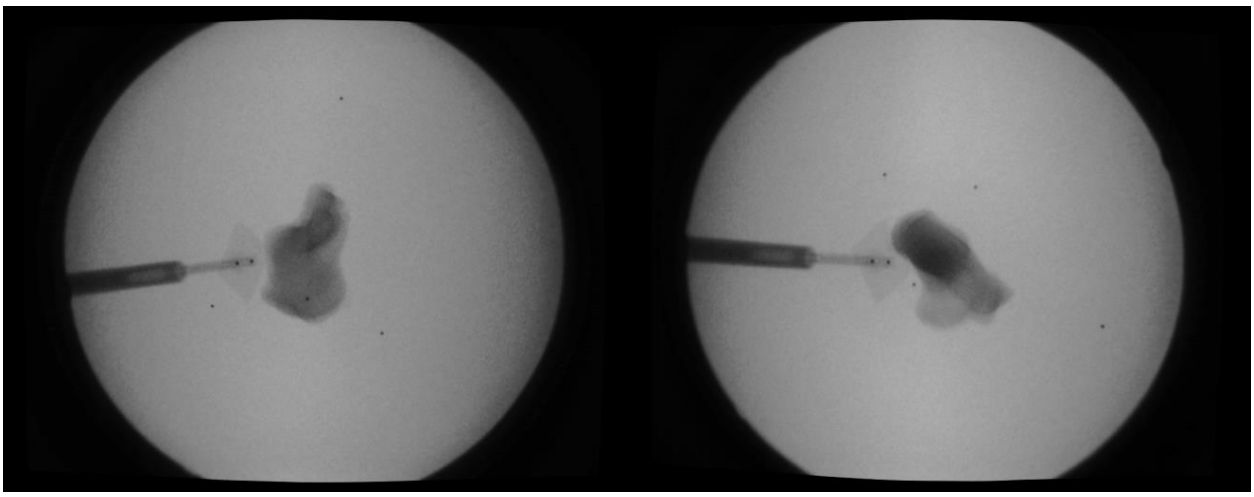
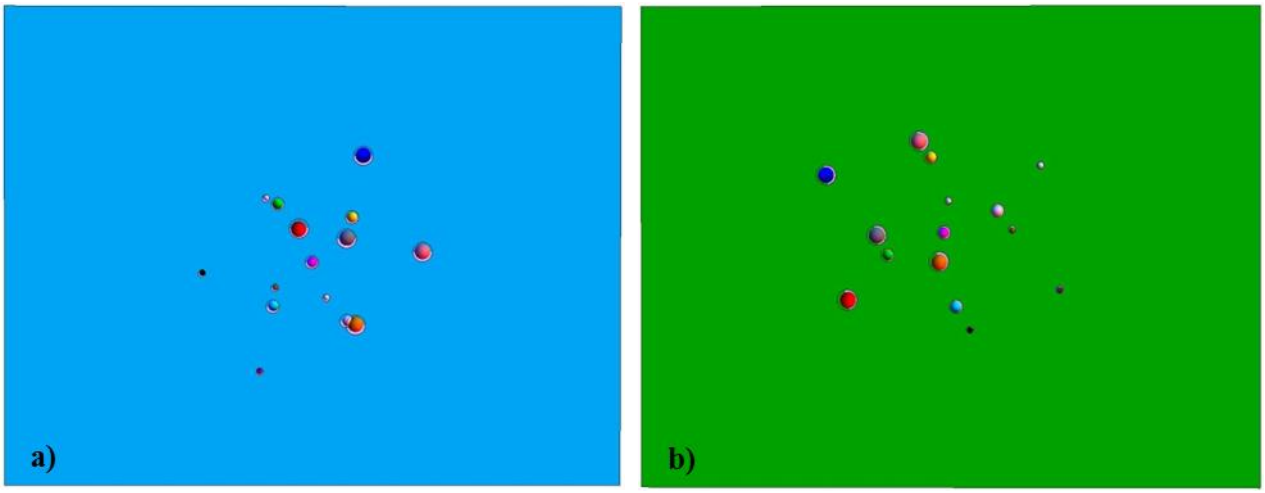


Figure 8.7. Stereoscopic fluoroscope image pair for calcaneus 7451.

8.2 Results

8.2.1 Image Formation

The CUDADRRs generated for the 15-bead system calibration block agreed to within machine precision with the synDRRs (Figures 8.8a & 8.8b). CUDADRRs were also compared to the corresponding fluoroscope images (Figures 8.8c & 8.8d). DRRACC's RMS errors were 0.1696 mm and 0.4535 mm, for the blue- and green-cameras, respectively. Corresponding RMS errors for the CPUDRRs were 0.5331 mm and 0.4115 mm, for the blue- and green-cameras, respectively. Differences in RMS errors between CUDADRRs and CPUDRRs can be attributed, at least in part, to the methods used for identifying bead centroid coordinates within the DRRs. CPUDRRs use an edge-detection algorithm to identify the approximate boundary of each bead, estimate bead radii, and determine bead centroids. On the other hand, bead centroids were identified in CUDADRRs by visually selecting the pixel that best represented the center of the bead. The study served to validate DRRACC's image formation method and in fact, the results suggest that CUDADRRs are more accurate than CPUDRRs.



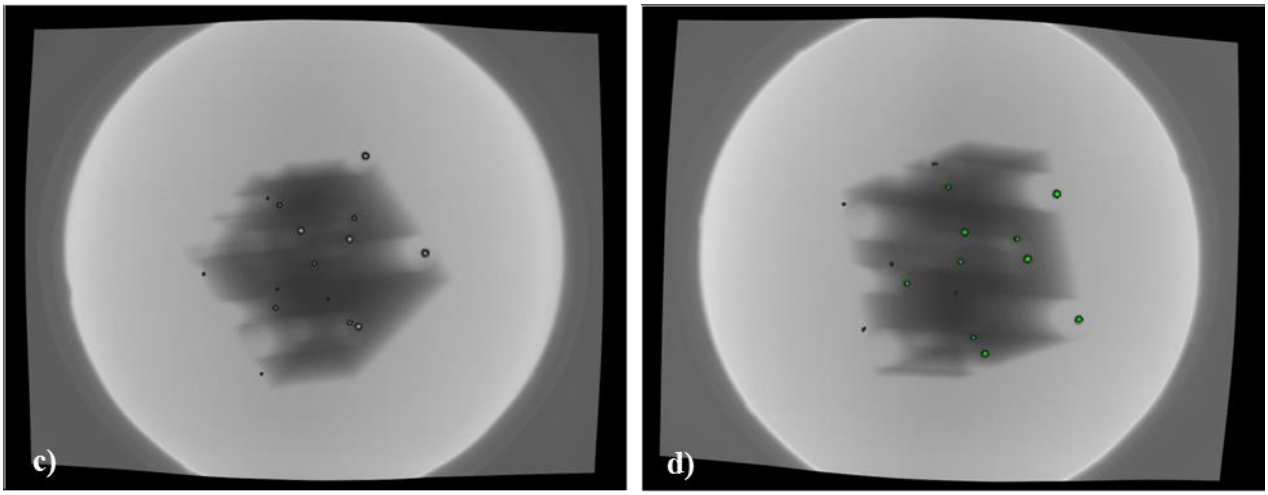


Figure 8.8. Image formation validation. (a) Validation of the blue- and (b) green-camera synDRRs against synDRRs. The spheres in the image pair are the digitally projected beads using simple geometry in Mathematica[25] and the nearly indistinguishable circles surrounding the spheres are the locations of the beads in the CUDADRRs. The beads align so well in most cases that the circle is not visible. (c) Validation of the blue- and (d) green-camera CUDADRRs. The DRRs are superimposed atop the fluoroscope images.

8.2.2 Pixel-by-Pixel Intensity Comparison

CUDADRRs and CPUDRRs of the synthetic CT data were found to be in good agreement as shown in Figures 8.9a & 8.9b. The circular objects in Figures 8.9a & 8.9b locate the centers of the projected objects from CUDADRR, while the "plus" signs locate the centers of the projected objects from CPUDRR. The pixel-by-pixel intensity comparison yielded a mean pixel intensity error between CUDADRRs and CPUDRRs of 9.646×10^{-7} Hounsfield Units. This result suggests that there are minimal discrepancies between CUDADRRs and CPUDRRs with respect to both projected pixel location and pixel intensity.

[25] The Mathematica notebook created for image formation validation can be found in Appendix F.

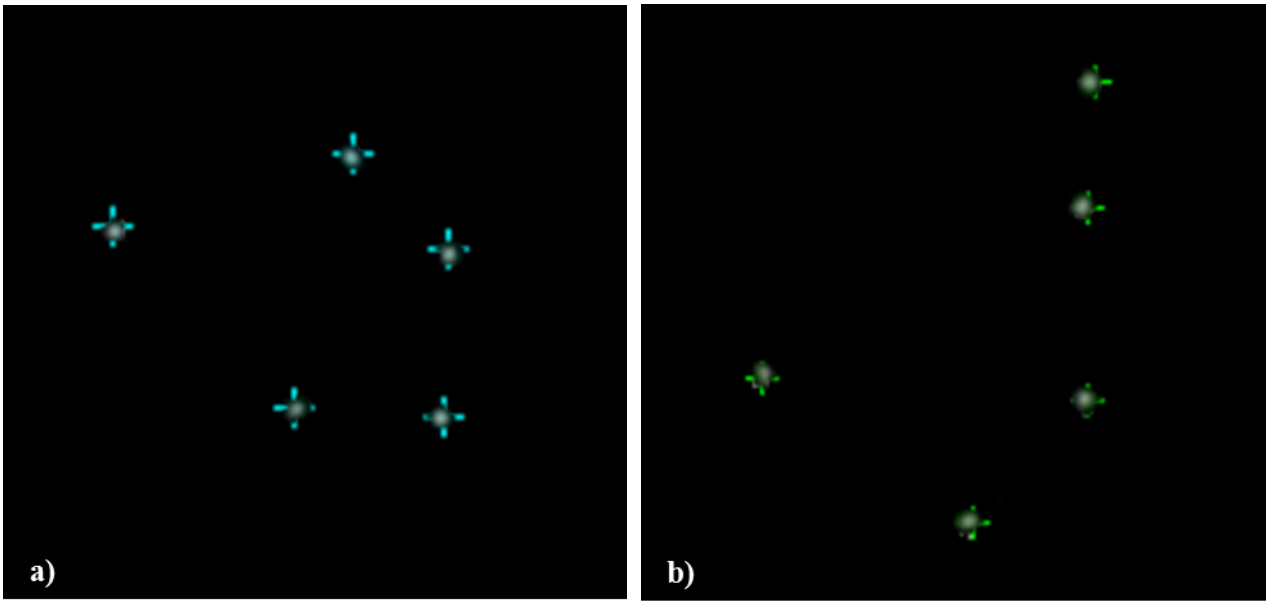


Figure 8.9. Pixel-by-pixel intensity comparison. (a) Superimposed blue- and (b) green-camera CUDADRRs and CPUDRRs of the synthetic CT data used in the pixel-by-pixel intensity comparison study. Projected objects in CUDADRRs are represented by circles, while projected objects in CPUDRRs are represented by "plus" signs. Note that the images have been magnified to highlight the agreement between CUDADRRs and CPUDRRs; locational discrepancies are misleading due to magnification.

8.2.3 Image Linearity

The scatter plots generated by both software platforms are shown in Figures 8.10a-d. The plots reveal a strong linear relation between the DRR and fluoroscope intensities, suggesting that 1) the biplane fluoroscope system is capturing images without noticeable errors, 2) the distortion and bias correction software does not significantly alter the linearity of fluoroscope pixel intensities, and 3) CUDADRR and CPUDRR pixel intensities are similar across real imaging data[26]. However, the plots also suggest that a curve used to fit the data would need to match the apparent change in slope; the nonlinear change in slope is consistent with the levels of pixel saturation expected at large intensities. The behavior suggests that fluoroscope pixel intensities are slightly saturated at locations exhibiting large intensity values. Saturation of the fluoroscope

[26] Note that the perceived differences between the CUDADRR and CPUDRR scatter plots can be attributed to the sampling approach; only a subset of the data was visualized for clarity.

pixel intensities can be caused by the pincushion effect, magnetic lens distortion, or application of insufficient bias correction methods. At this point, the nonlinearity has not been severe enough to justify scaling to account for saturation.

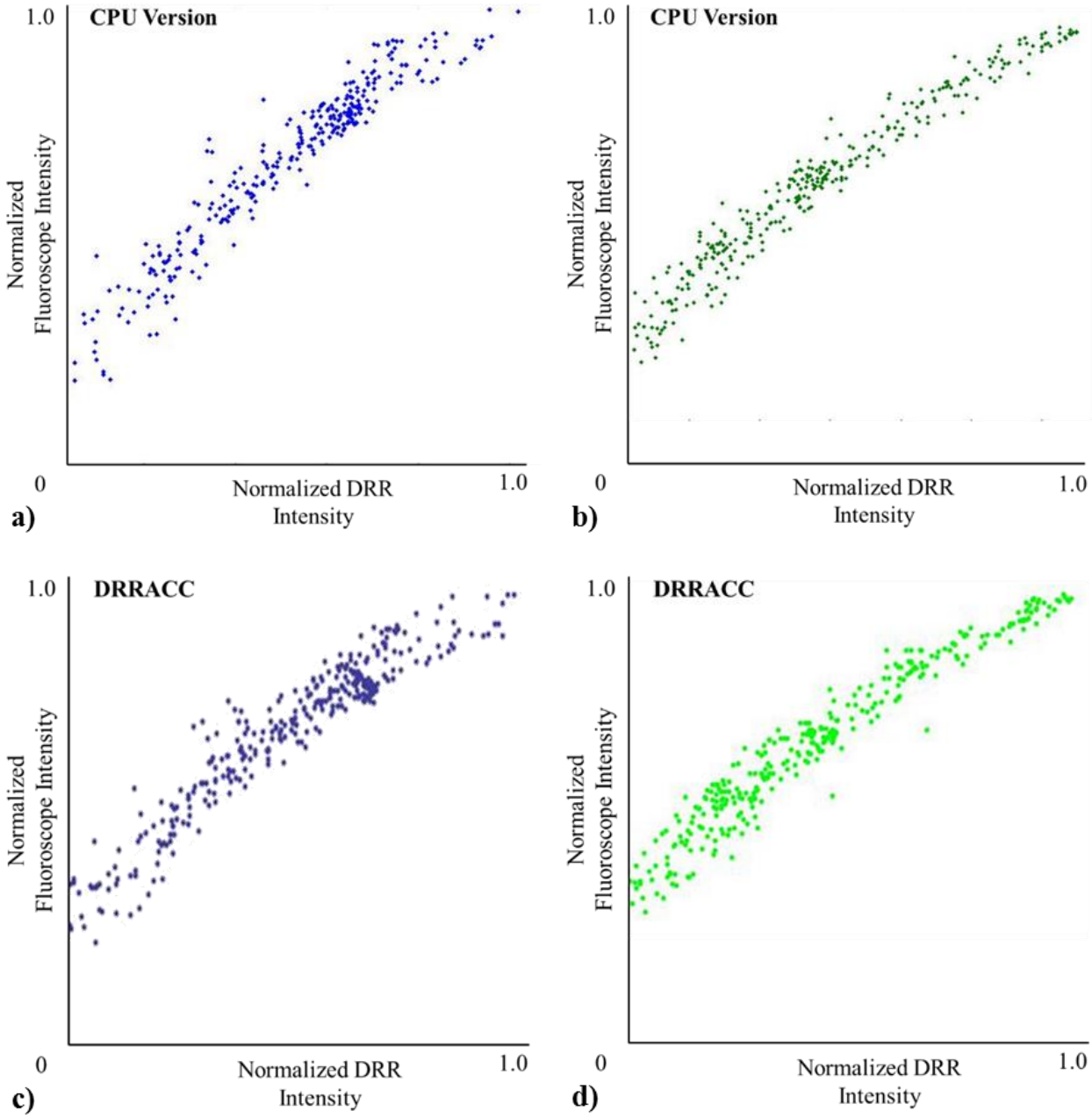
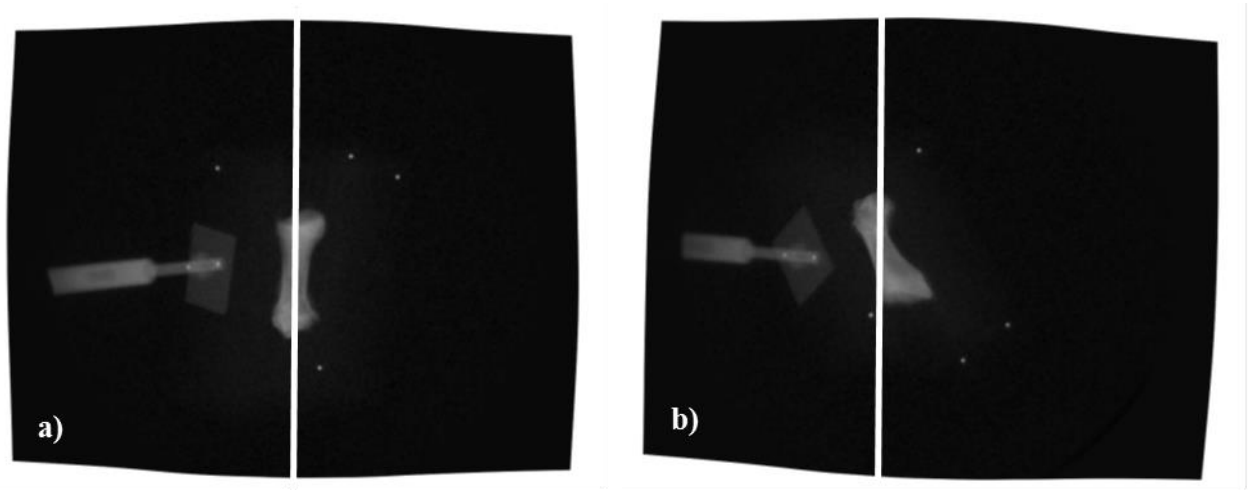
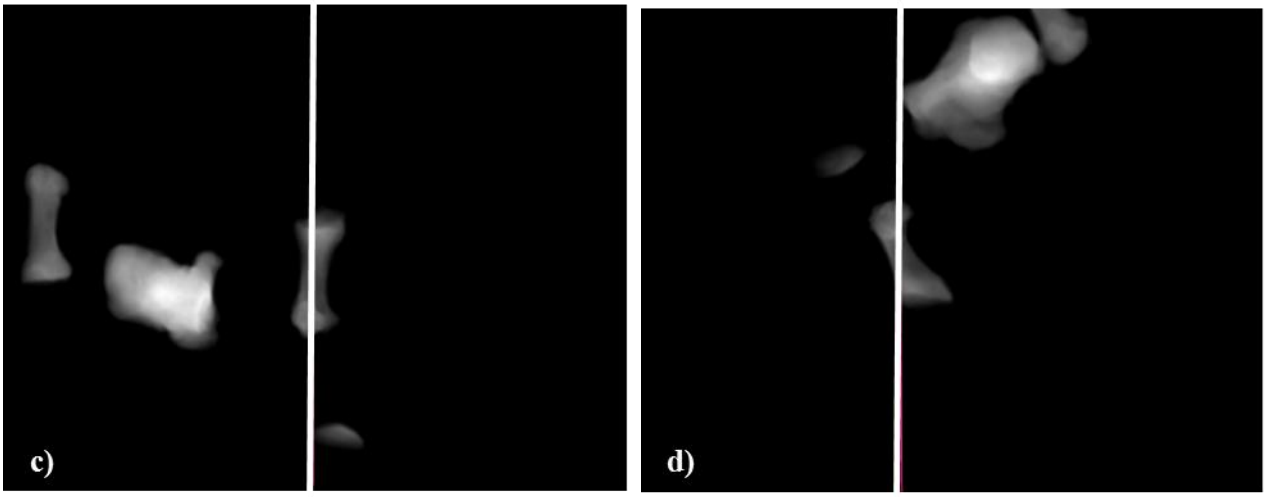


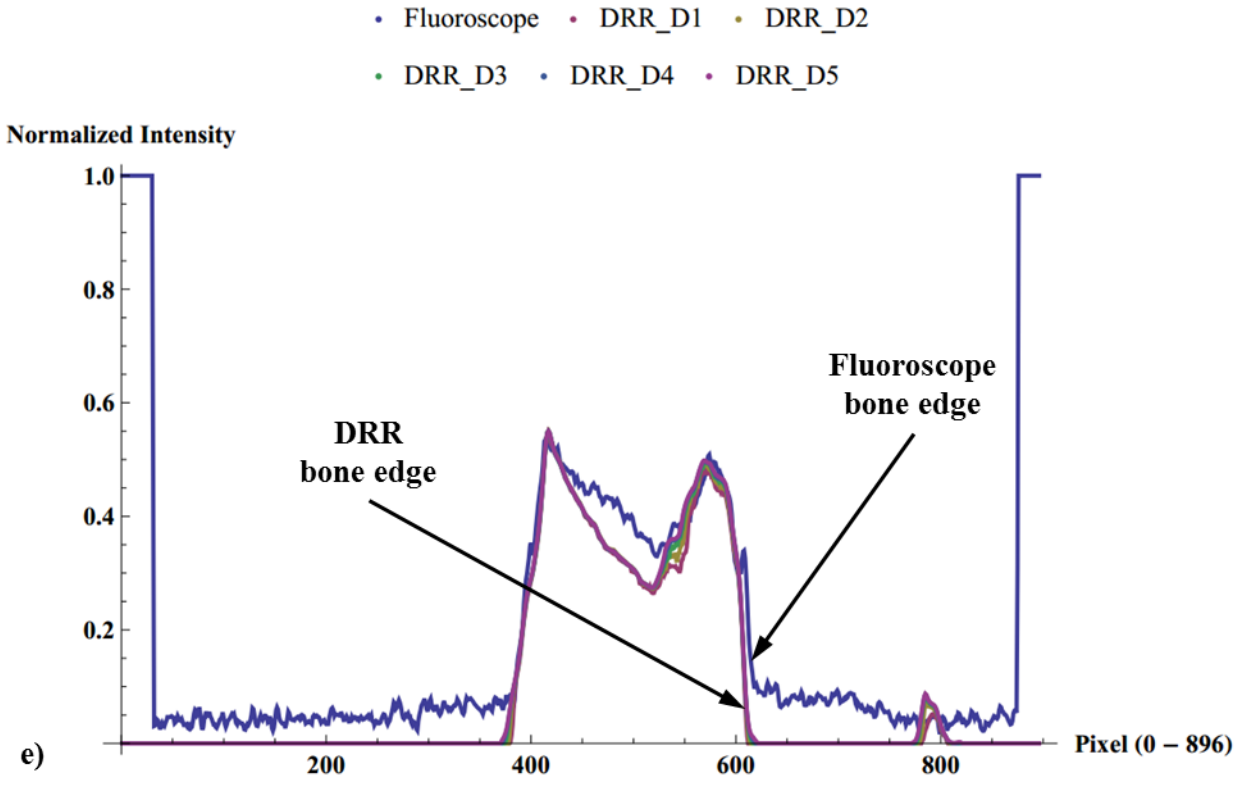
Figure 8.10. Scatter plots of DRR intensity versus fluoroscope intensity. (a) Blue- and (b) green-camera CPUDRR intensity versus fluoroscope intensity, and (c) blue- and (d) green-camera CUDADRR intensity versus fluoroscope intensity. Every 100th data point was used for these plots.

8.2.4 Image Geometry

Evidence to support the hypothesis that excessive dilation was being applied to the label file was found by generating a series of profile plots (Figures 8.11e-f). The profile plots were produced by isolating a column of pixel intensity from both the fluoroscope images (Figures 8.11a-b) and the CUDADRRs (Figures 8.11c-d). The profile plots reveal that aside from minor differences in maximum pixel intensity, the change in DRR size was not strongly influenced by a change in label file dilation. This result can be distilled from the profile plots by viewing the location of the edges in the DRRs; a large change in DRR size would be expressed by a change in profile width, measured from the edge located at pixel ~375 to the edge located at pixel ~625 for the blue-camera (pixel ~350 to ~550 for the green camera).




c)
d)


Fluoroscope
DRR_D1
DRR_D2
DRR_D3
DRR_D4
DRR_D5
Normalized Intensity
1.0
0.8
0.6
0.4
0.2
DRR
bone edge
Fluoroscope
bone edge
200
400
600
800
Pixel (0 – 896)
e)

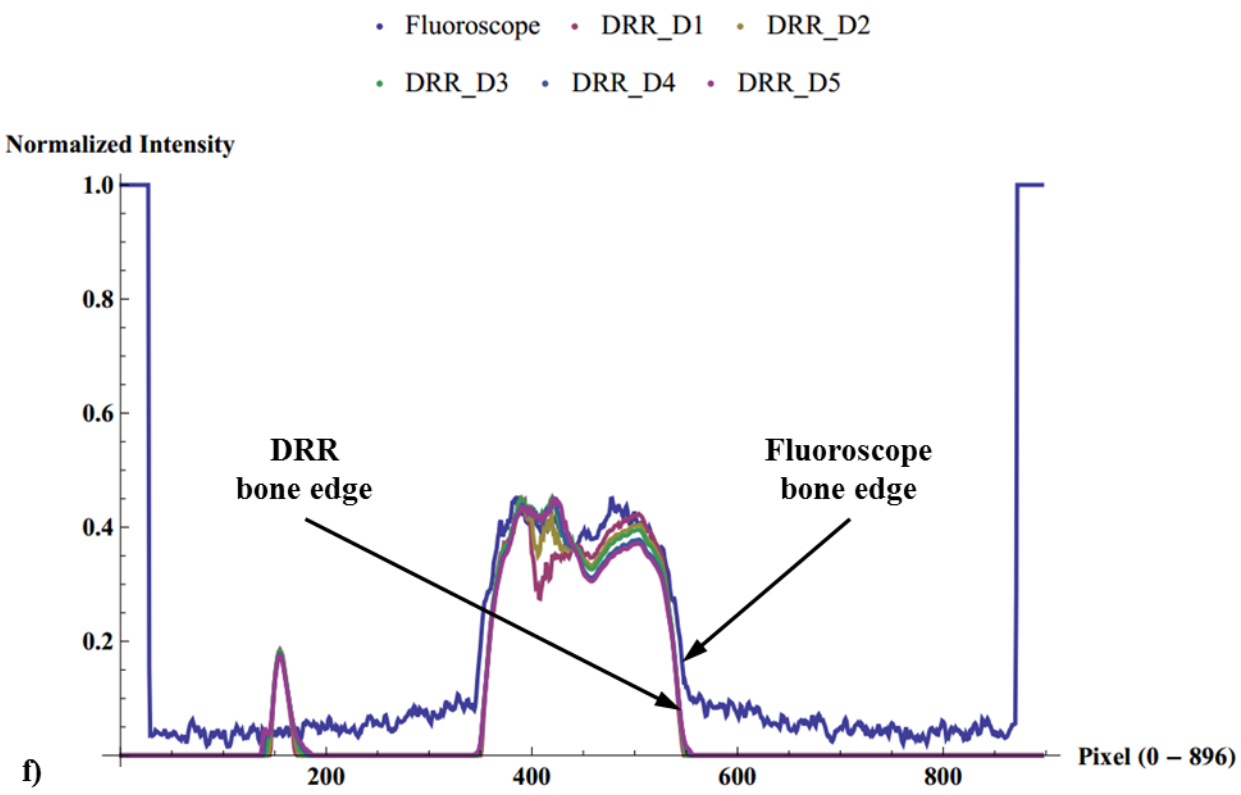
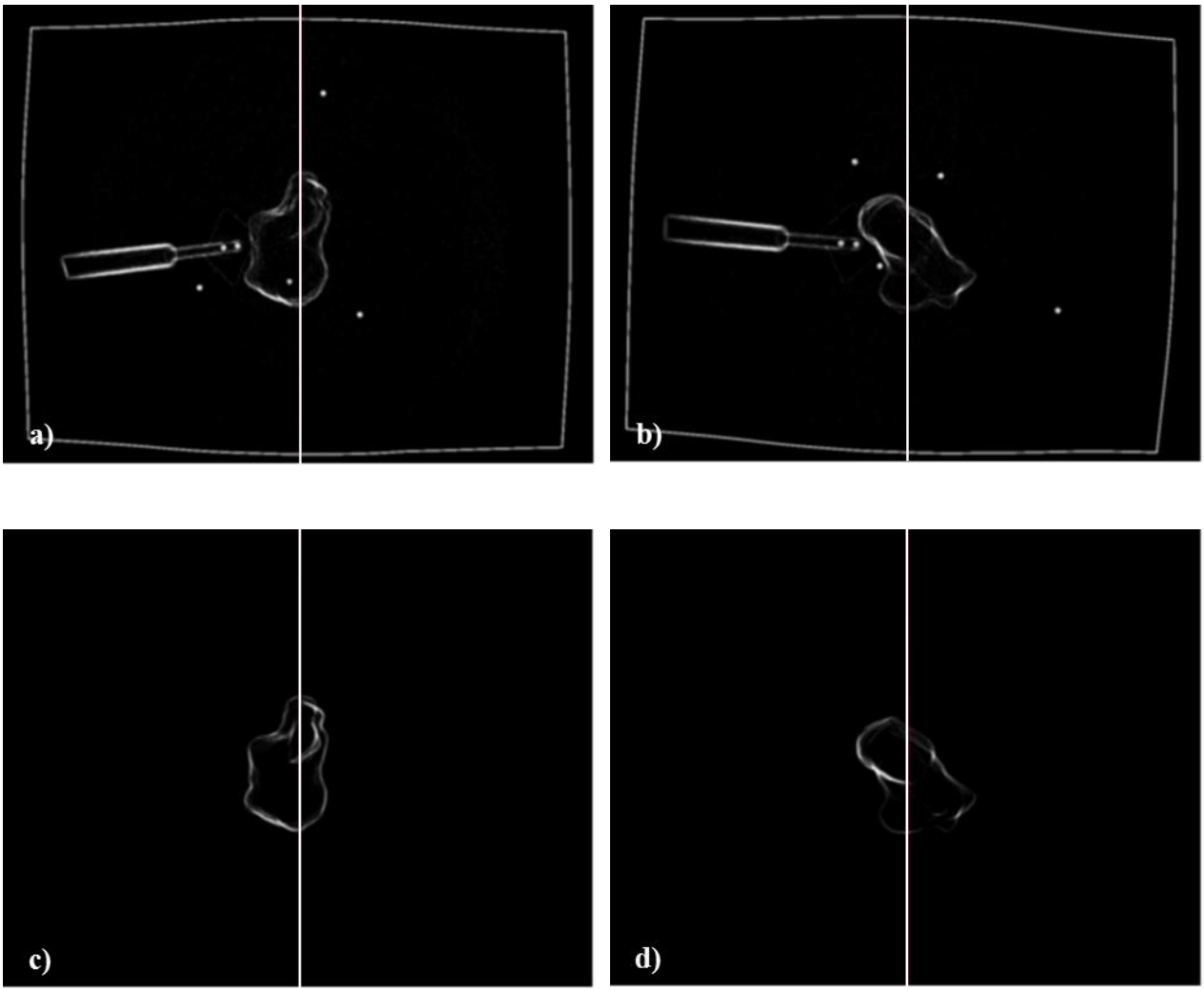
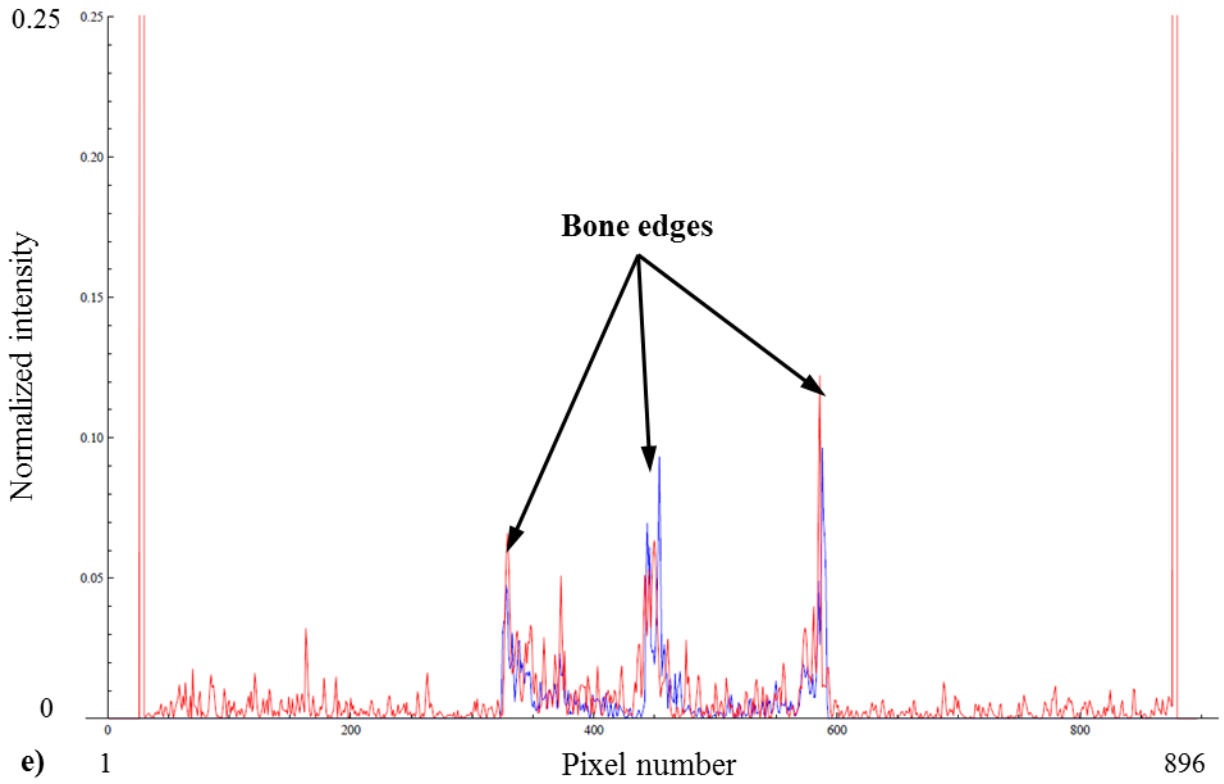


Figure 8.11. DRR sensitivity to label file size. (a) Blue- and (b) green-camera edge-based fluoroscope images. The lines intersecting the plots in (a) & (b) represent the columns of pixel intensities used for the profile plots. (c) Blue- and (d) green-camera CUDADRRs. (e) Blue- and (f) green-camera superimposed profile plots of the fluoroscope image and five DRRs with varying dilation factors. Bone edges are aligned indicating a successful registration. Dilation factor does not result in a significant change in bone width.

During the course of testing, it was observed that one of the two CUDADRRs (i.e., blue- or green-camera) was seemingly always "preferred" by the optimizer; NCC values for the preferred camera were consistently better than those of the non-preferred camera across an entire set of imaging data. That is, the optimizer would routinely focus on improving one camera over the other, regardless of starting configuration. A column of pixel intensity was extracted from each the edge-based fluoroscope images (Figures 8.12a & 8.12b) and edge-based CUDADRRs (Figures 8.12c & 8.12d). A set of profile plots comparing normalized pixel intensities for the fluoroscope images and CUDADRRs (Figures 8.12e & 8.12f) were created to test the idea of a bias-optimizer, however evidence did not support this postulation.

Concrete evidence rejecting this hypothesis did not present itself until DRRACC was relied on to process the stage translation and rotation data. During analysis, it was found that the average blue- and green-camera edge-based NCC values for the stage translation study were 0.6352 and 0.7363, while for the stage rotation study they were 0.5410 and 0.3823, respectively. These findings disproved the notion of the optimizer biasing one camera over the other.




0.25
0.20
0.15
0.10
0.05
0
Normalized intensity
Bone edges
0
200
400
600
800
e)
1
Pixel number
896

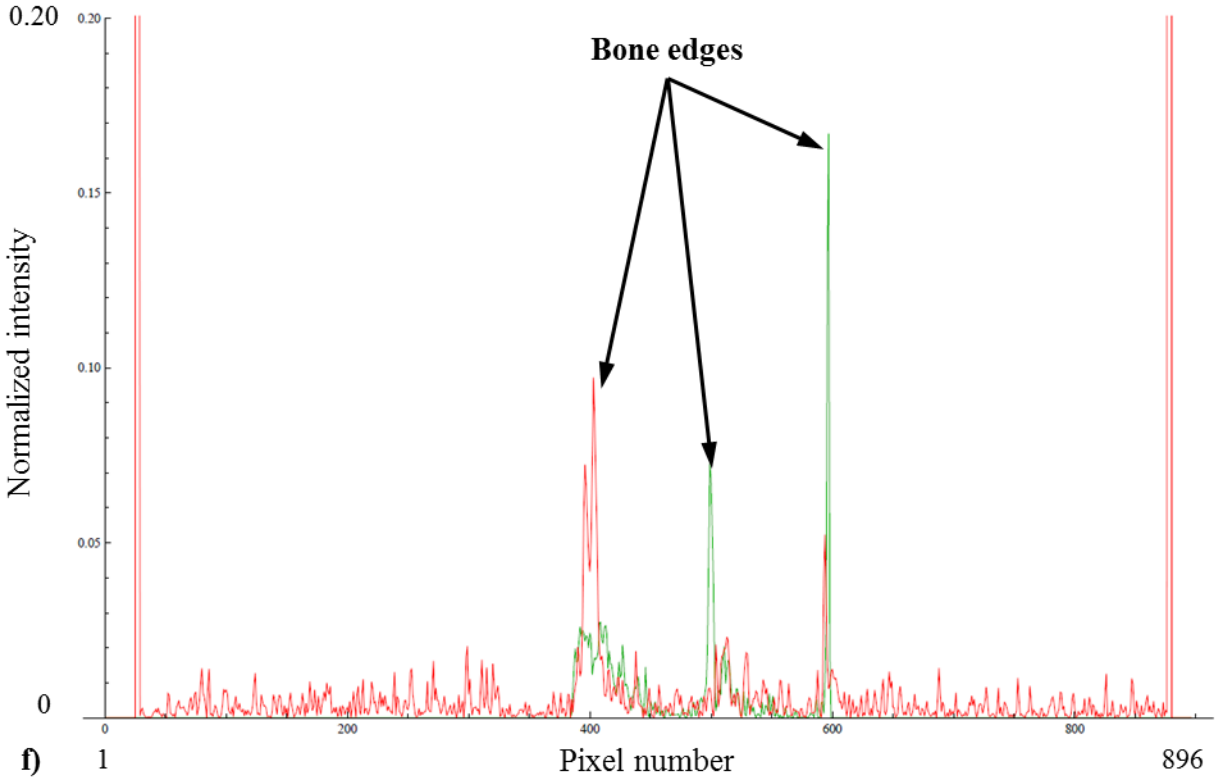

0.20
0.15
0.10
0.05
0
Normalized intensity
Bone edges
0
200
400
600
800
f)
1
Pixel number
896

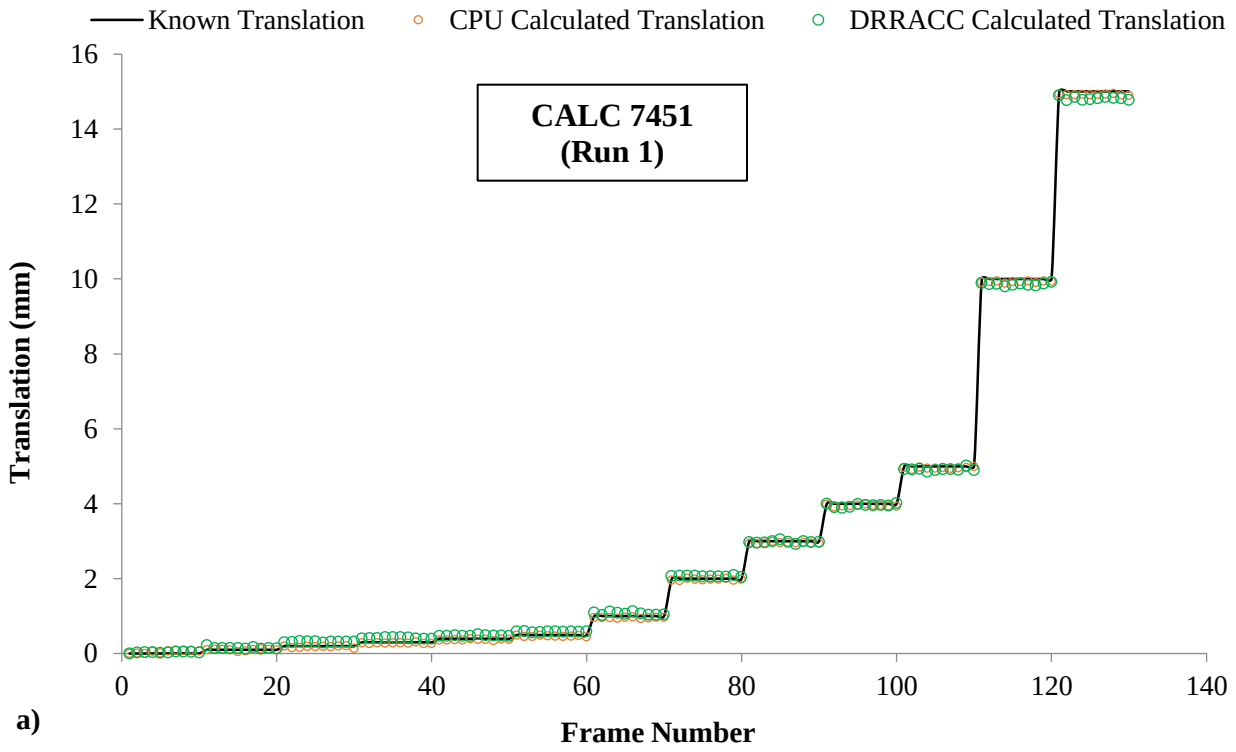
Figure 8.12. Edge-based profile plots. (a) Blue- and (b) green-camera edge-based fluoroscope images. The lines intersecting the plots in (a) & (b) represent the columns of pixel intensities used for the profile plots. (c) Blue- and (d) green-camera edge-based CUDADRRs. Profile plots for the (e) blue- and (f) green-cameras. Alignment of bone edges (peaks) are in good agreement, indicating a successful registration.

8.2.5 Stage Translation

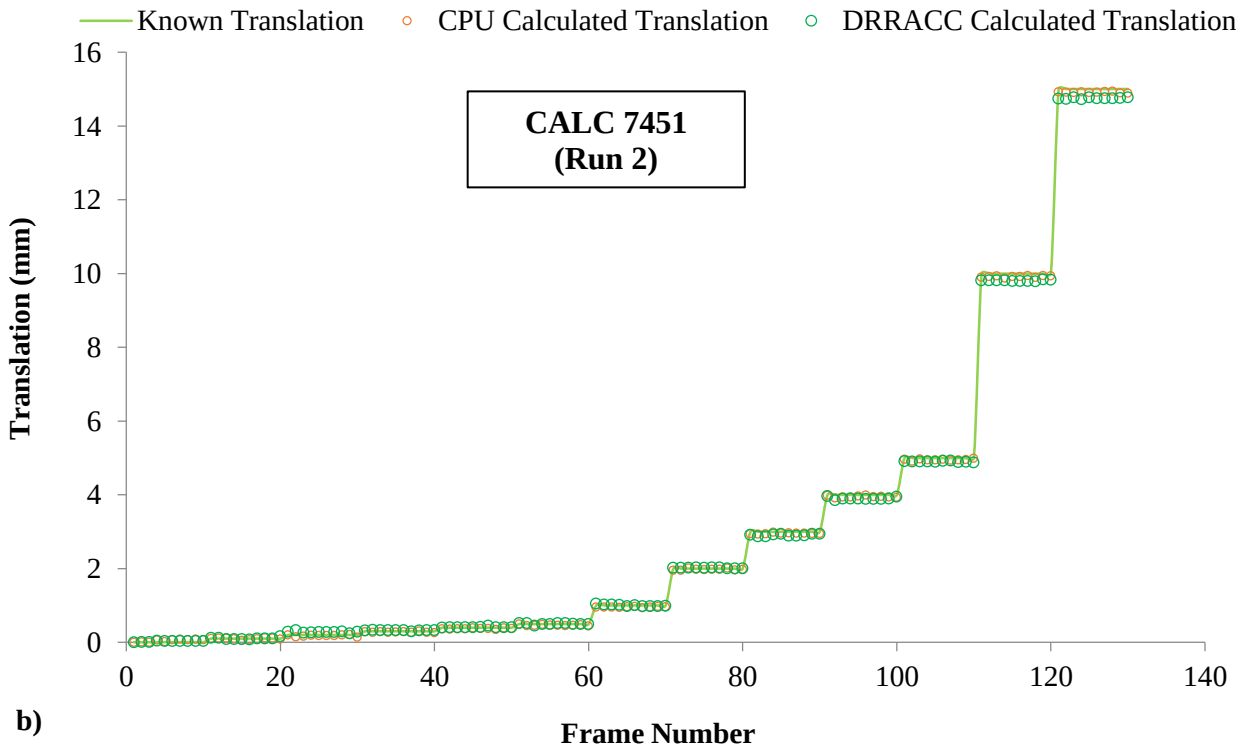
Figures 8.13a-g (calcaneus 7451 (run 1), calcaneus 7451 (run 2), calcaneus 7481, 1st metatarsal 6992R (run 1), 1st metatarsal 6992R (run 2), 1st metatarsal 7451, talus 7451 and talus 7481) compare the known translation of the bones to the translations identified by CPUDRR and DRRACC using the CONDOR optimization suite. RMS errors were computed using the absolute difference in translation from the known position to the optimized position. The results of the study are summarized in Table 8.1. DRRACC achieved sub-millimeter precision for the six stage translation data sets processed. Raw data for each study can be found in Appendix D.

Table 8.1. Summary of the stage translation validation study.

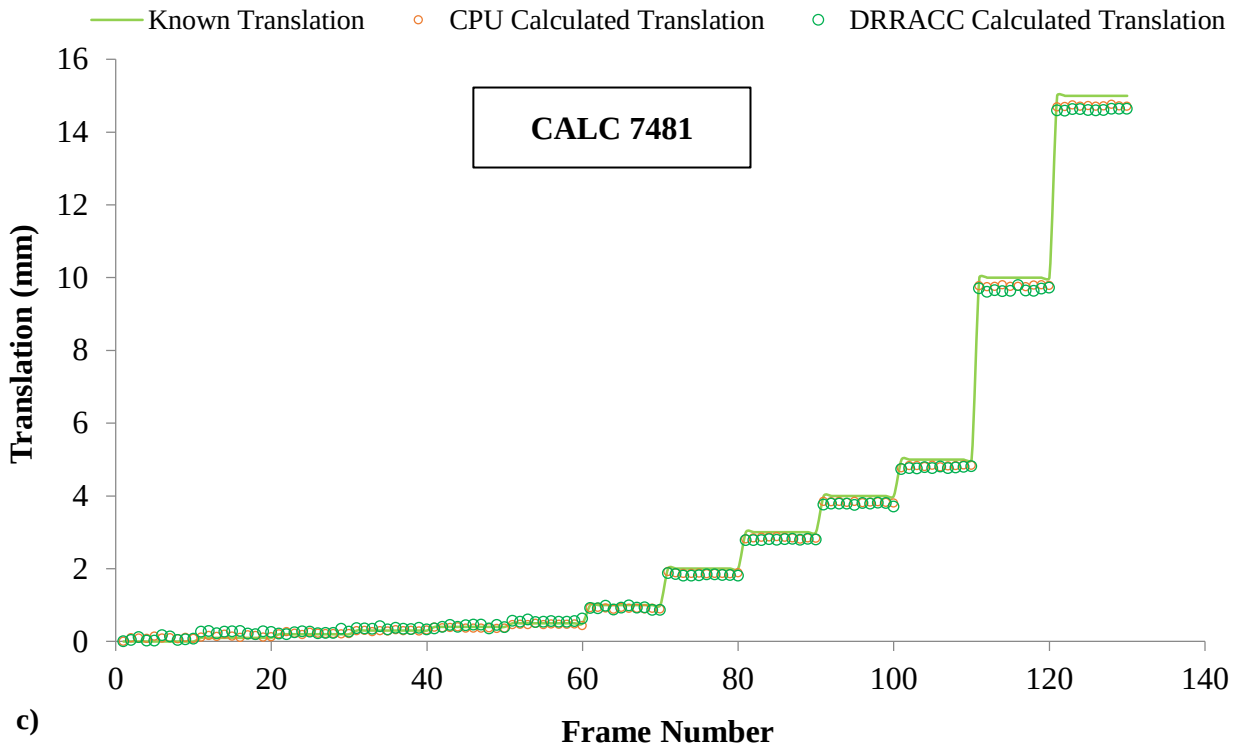
RMS Errors (mm) - Stage Translation		
Bone	**CPUDRR**	**DRRACC**
1st MET 7451	0.038422	0.044050
1st MET 6992R	0.027973	0.051372
TALUS 7451	0.045905	0.071553
TALUS 7481	0.046832	0.084175
CALC 7451	0.047742	0.047742
CALC 7482	0.142880	0.194979


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
CALC 7451
(Run 1)
Translation (mm)
Frame Number
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120
140

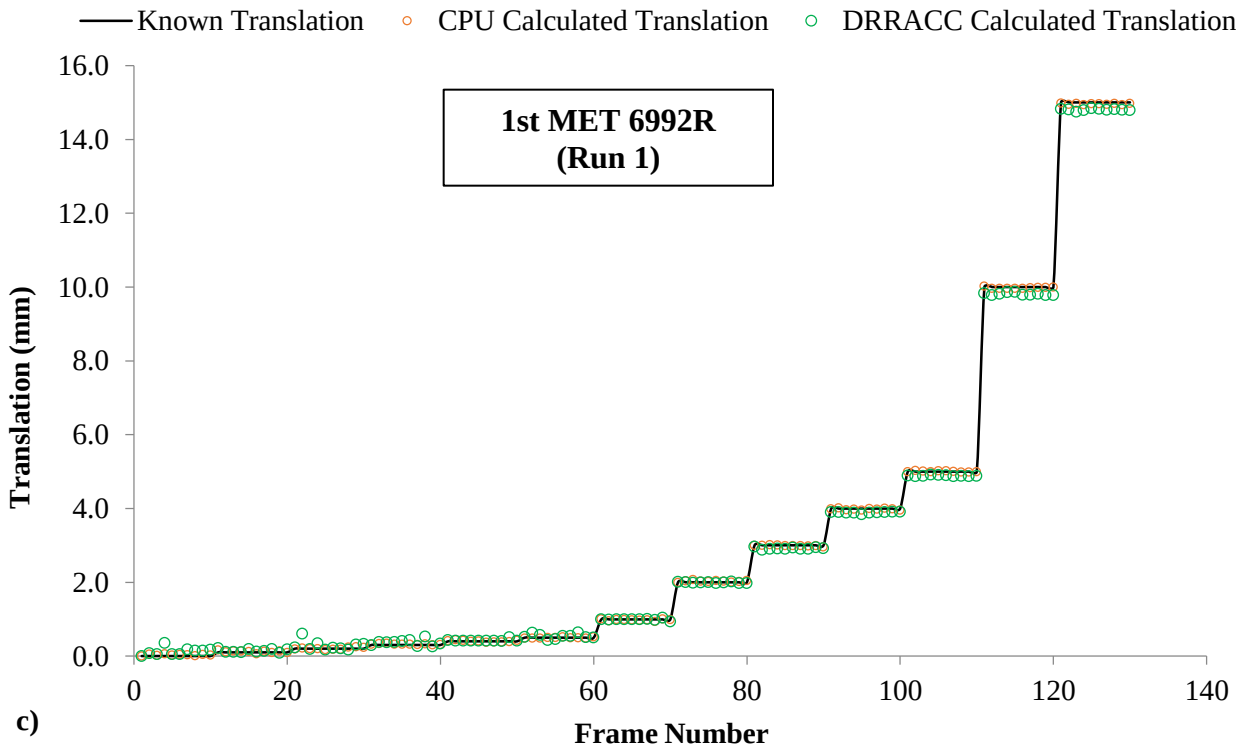
a)


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
CALC 7451
(Run 2)
Translation (mm)
Frame Number
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120
140

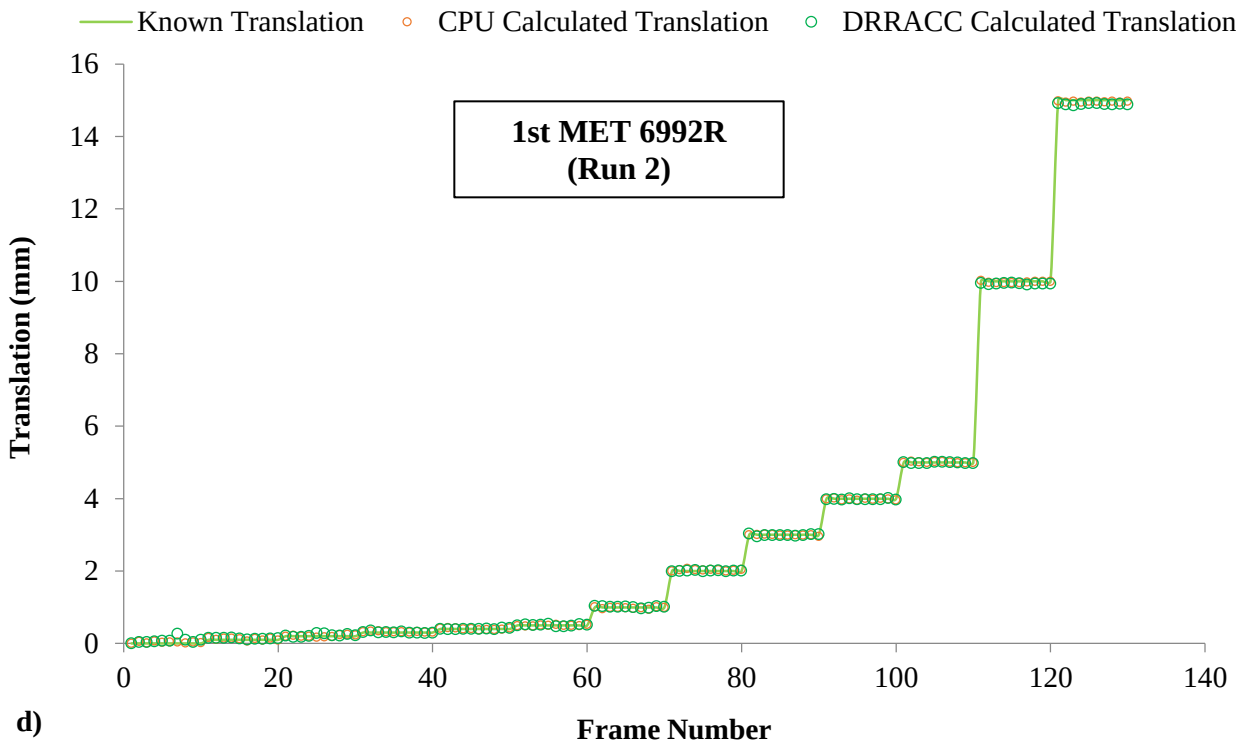
b)


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
CALC 7481
Translation (mm)
Frame Number
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120
140

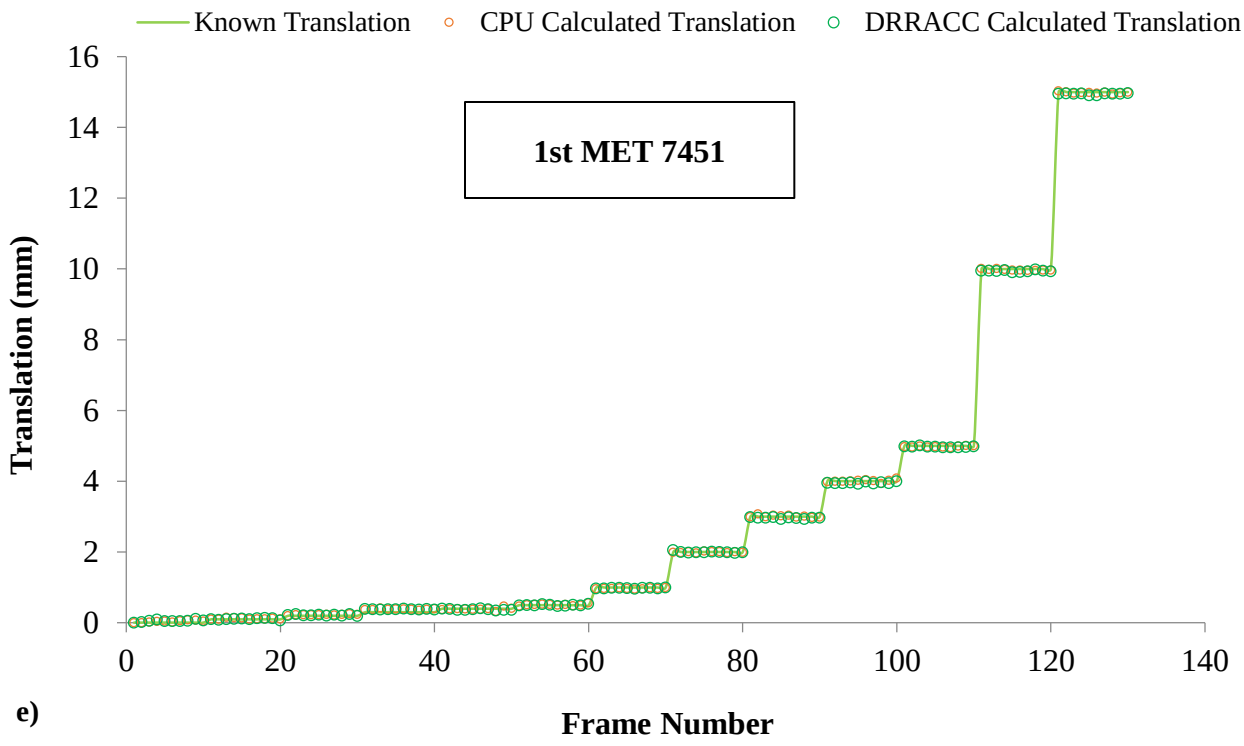
c)


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
1st MET 6992R
(Run 1)
Translation (mm)
Frame Number
0.0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
0
20
40
60
80
100
120
140

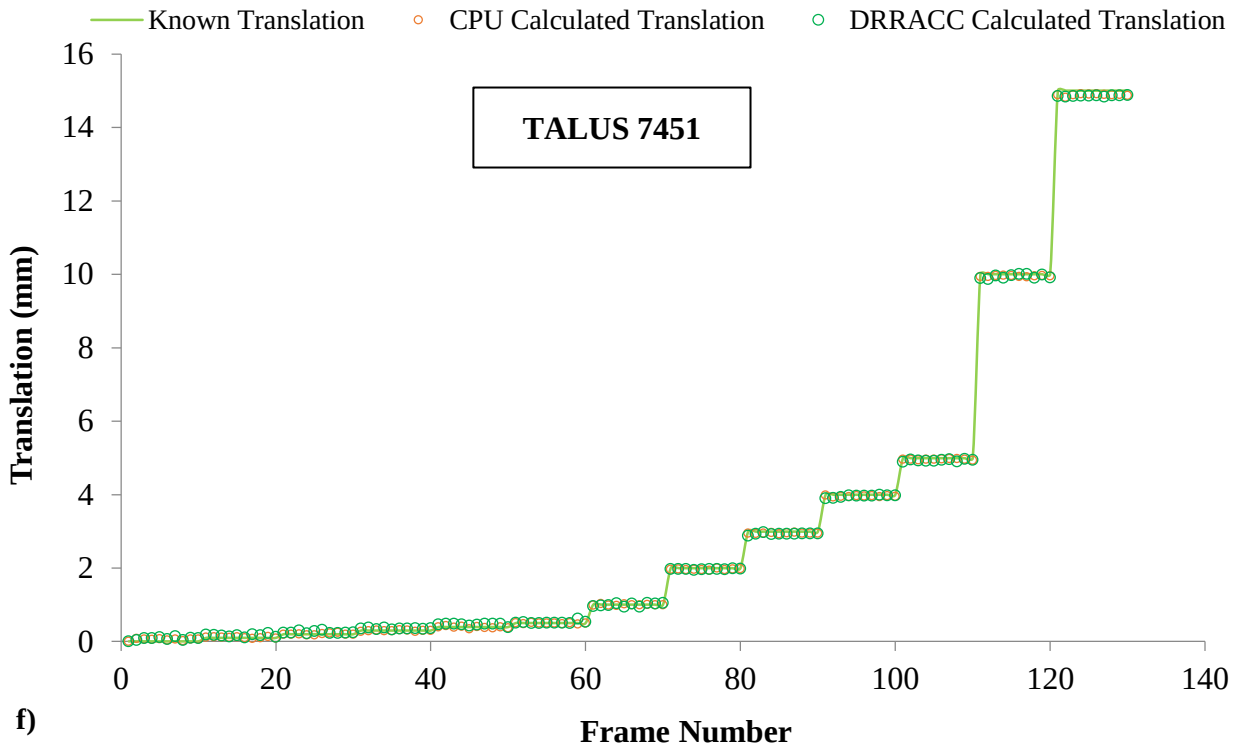
c)


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
1st MET 6992R
(Run 2)
Translation (mm)
Frame Number
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120
140

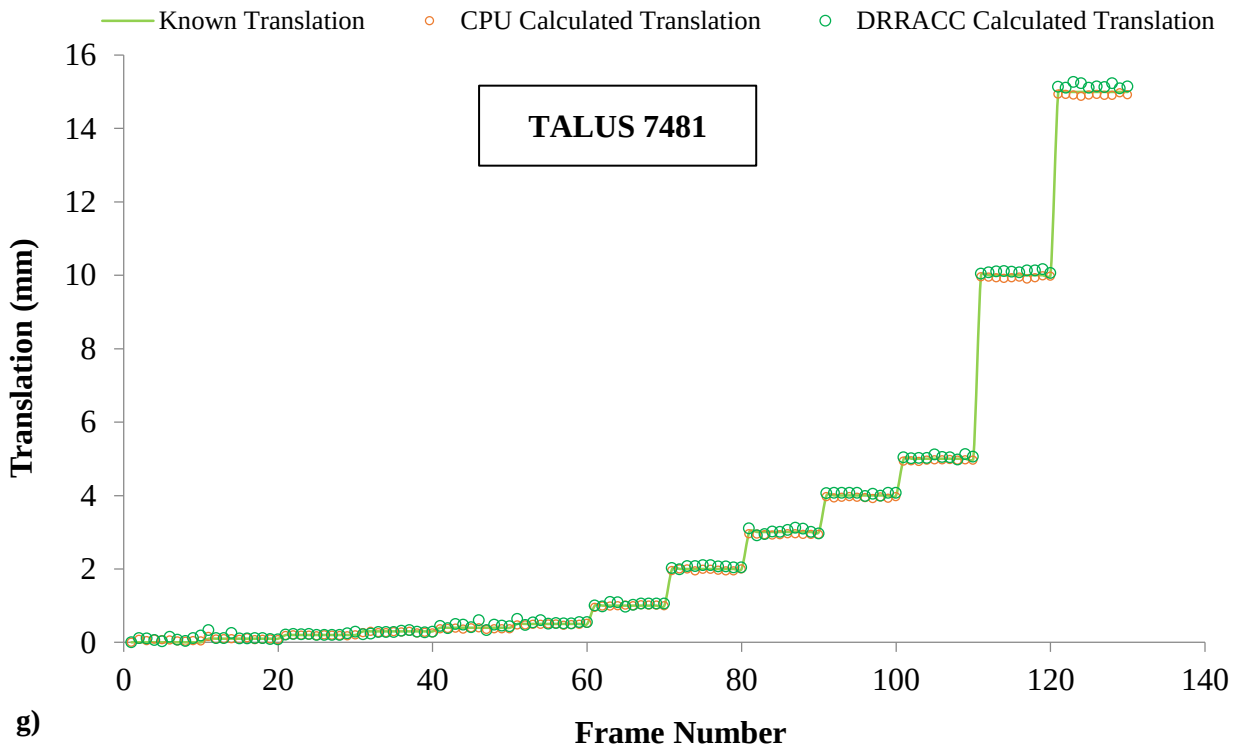
d)


Known Translation
CPU Calculated Translation
DRRACC Calculated Translation
1st MET 7451
Translation (mm)
Frame Number
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120
140

e)



f)



g)

Figure 8.13. Stage translation validation. (a,b,c,d,e,f and g) Stage translation results for calcaneus 7451 (run 1), calcaneus 7451 (run 2), calcaneus 7481, 1st metatarsal 6992R (run 1), 1st metatarsal 6992R (run 2), 1st metatarsal 7451, talus 7451 and talus 7481, respectively.

8.2.6 Stage Rotation

Figures 8.14a & 8.14b each plot two CPUDRR runs and a DRRACC run against the known rotation for calcaneus 7451. RMS errors were computed using the bone's *screw axis* and *screw angle* (sometimes referred to as the Euler axis and angle). The screw axis is a vector that an object can rotate about by the screw angle to recover 3D transformations without explicitly applying translation or decomposing the rotation matrix into Euler angles [73]. Table 8.2 summarizes the results of the study. CPUDRRs produced RMS errors of 0.1195 degrees ± 0.0202 degrees and 0.1544 degrees ± 0.0377 degrees, while DRRACC performed slightly better at 0.0759 degrees ± 0.0081 degrees. DRRACC achieved sub-degree precision for the stage rotation study. Raw data and standard deviations for each study can be found in Appendix E.

Table 8.2. Summary of the stage rotation validation study.

RMS Errors (mm) - Stage Rotation

Bone	CPUDRR Run 1	CPUDRR Run 2	DRRACC
CALC 7451 Run 1	0.119454	0.154383	0.075858
CALC 7451 Run 2	0.119454	0.154383	0.082276

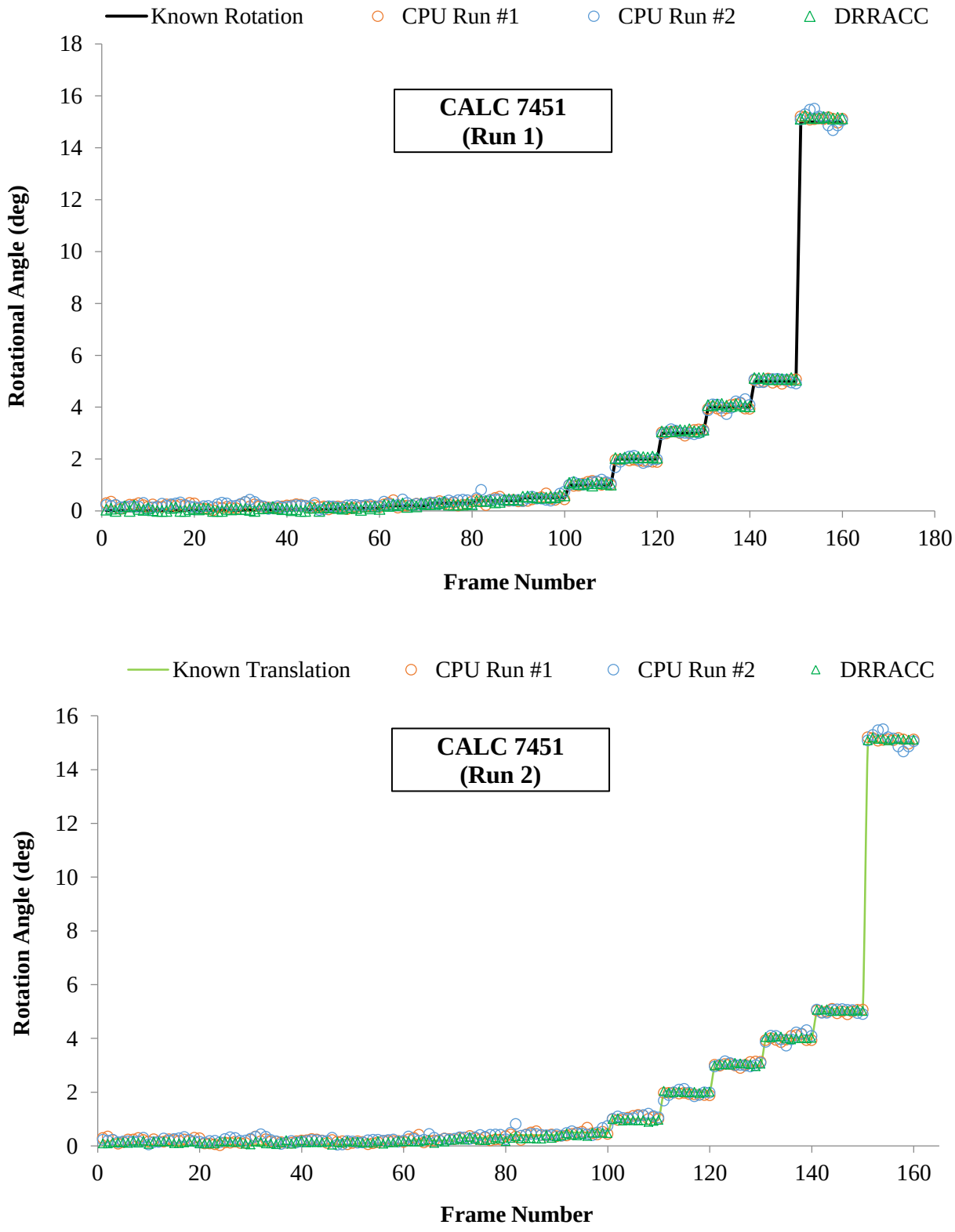


Figure 8.14 Stage rotation results for calcaneus 7451 (a) run 1 and (b) run 2 for DRRACC.

8.3 Toolkit Performance

8.3.1 Timings for DRR generation

By employing subroutines from the DRRACC toolkit, we computed DRRs from a CT scan of a cadaveric foot. An 1152 x 896 pixel image for the full foot including label '0' (soft tissues) is shown in Figure 8.15a. A similar image for the fully segmented foot without the soft tissues is shown in Figure 8.15b[27]. Using the full voxel extent as the bounding box, an average single bone DRR for the full 160 x 339 x 439 voxel CT data was computed for an 1152 x 896 pixel intensifier in ~13.1 ms (NVIDIA Tesla C2050 GPU). To realize even faster DRR generation, we reduced the size of the bounding box by storing the minimum and maximum voxel locations for each bone. The average compute time for a single bone utilizing the axis-aligned bounding box was ~1.3 ms (NVIDIA Tesla C2050 GPU).

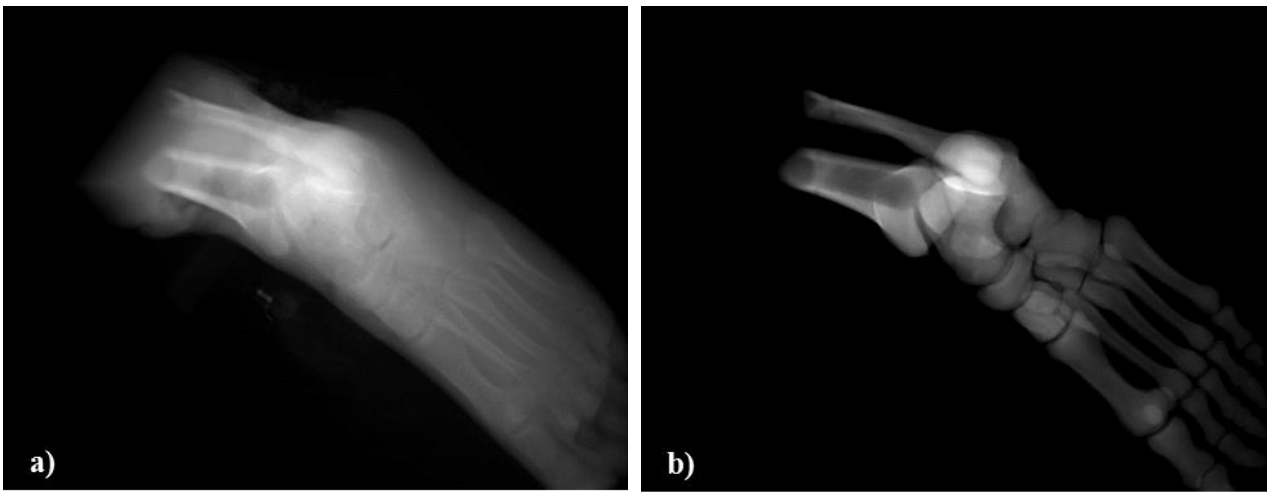


Figure 8.15. Visualization of soft tissues. a) Shows a DRR for the full CT data set, including label '0' (soft tissues), while b) shows the same DRR adjusted to include only the segmented bones. A total of 28 DRRs were generated and composited to produce these images.

For comparison, a CUDADRR was generated using a more typical 512 x 512 pixel screen (Figure 8.16). The change in resolution provided a significant reduction in overall runtime and an

[27] Note that the DRR computed by our toolkit is visually similar to a true X-ray image.

average single bone DRR was computed in less than 0.6 ms. All 28 DRRs were generated in less than 16.2 ms, while compositing operations accounted for 1.1 ms (Tesla).

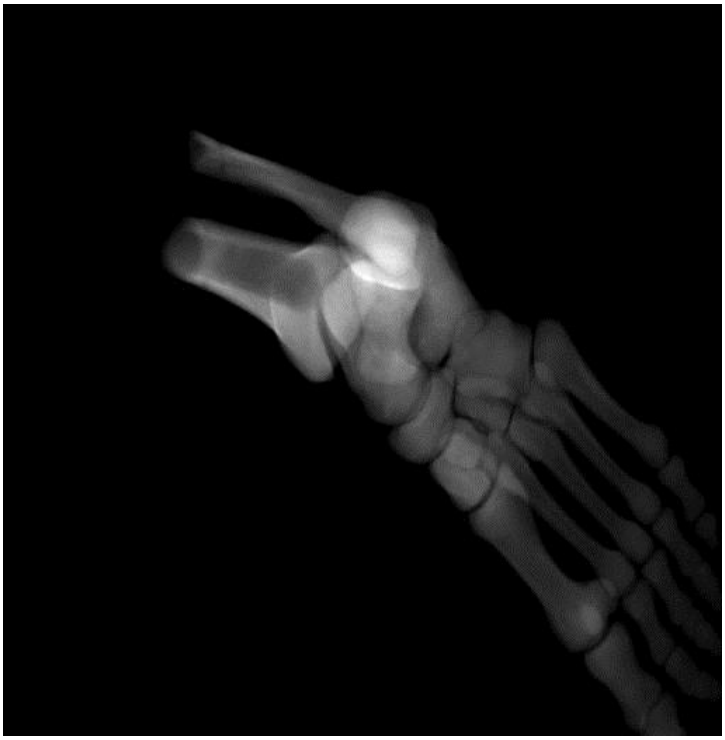


Figure 8.16. Composited DRR. Includes all 28 bones on a 512 x 512 pixel screen generated in less than 16.2 ms, which includes individual DRR construction for each bone.

An example of a single bone DRR is shown in Figure 8.18a. A comparison of the full voxel extent and individually aligned bounding box is shown in Figure 8.18b. During an optimization step, typically only a single bone is transformed, thus yielding computing times of ~1.3 ms for each DRR update. Even faster compute times will be possible with the use of oblique (non-aligned) bounding boxes.

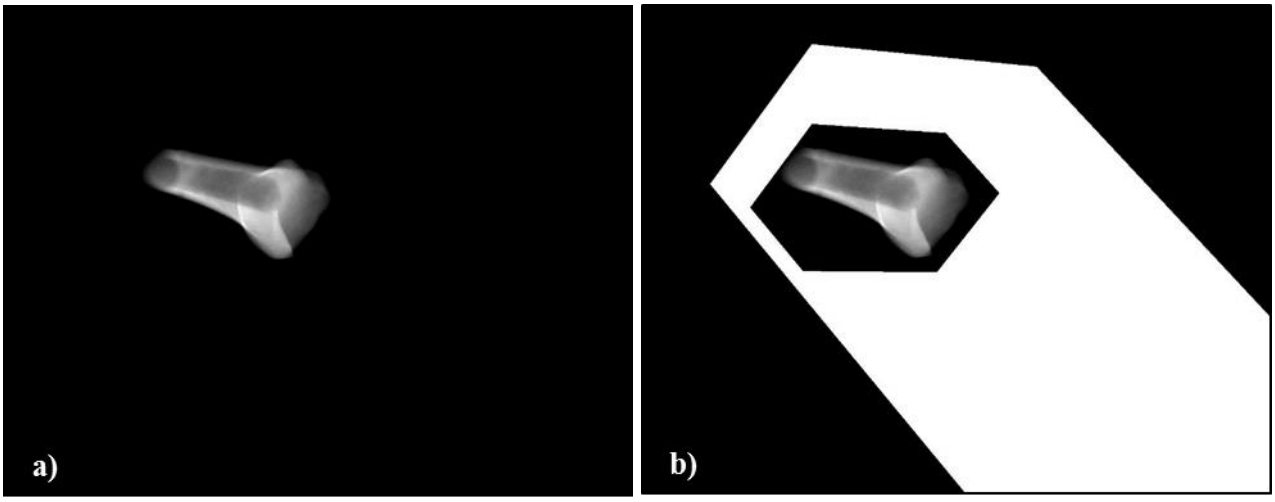
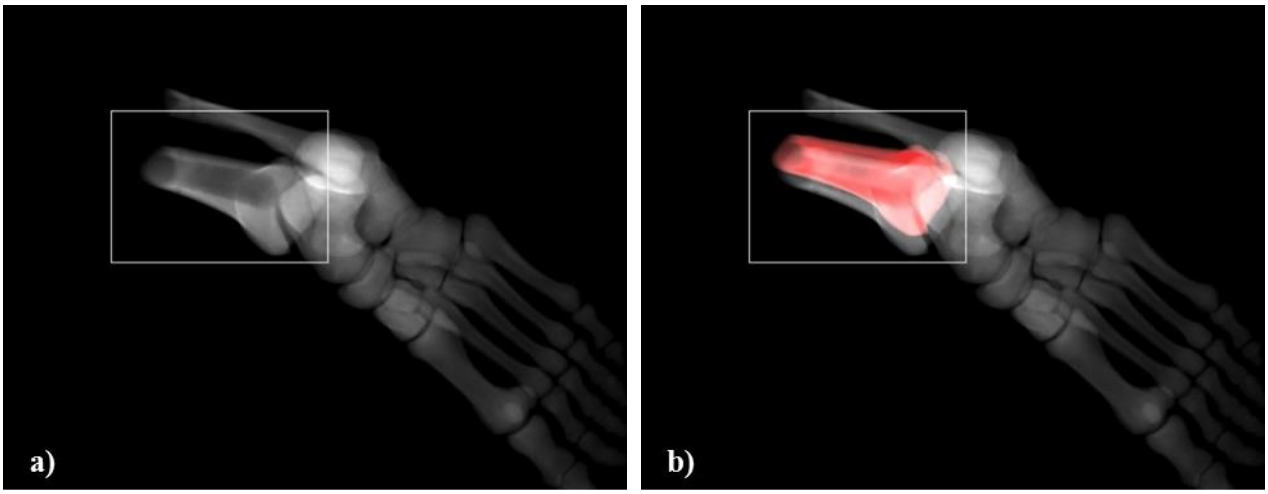


Figure 8.17. Visualization of a bone's bounding box. a) A single-bone DRR for the tibia and b) shows the same DRR with projection of the bounding boxes onto the screen. The full voxel extent is shown in white, while the axis-aligned bounding box is shown within in black.

In Figure 8.18a, we show the DRR with all bones visible and the cropped region of interest (ROI) highlighted by the white rectangle. In this region the NCC is computed while independently transforming the (Figure 8.18b-8.18d). As depicted in Figure 8.19, computation of the similarity measure is a function of the number of pixels in the ROI.



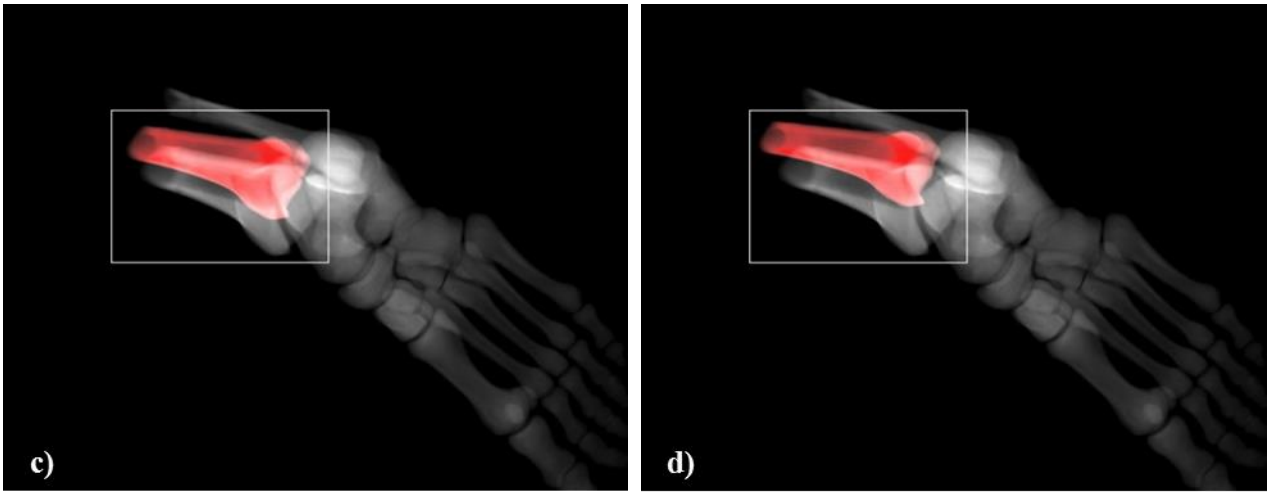


Figure 8.18. A sequence of transformed DRRs. Figure (a) shows the full 1152 x 896 pixel image with crop region, (b-d) show single bone transformation.

Initial timing experiments were performed on three GPUs; the GT440 with 96 processors, the GTX460 v2 with 336 processors and the Tesla C2050 with 448 processors. Table 8.3 provides a comparison of the timings for three commercially available GPUs. The Tesla card provided the fastest timings in all cases except for the NCC computation.

Table 8.3. Timings for a 160 x 339 x 439 CT data set projected onto an 1152 x 896 pixel screen. Note the significant reduction in computation time when implementing individually aligned bounding boxes.

CT volume: 160 x 339 x 439 (~23.8 x 10^6) voxels || **DRR image size: 1152 x 896 (~1 x 10^6) pixels**						
GPU	# of processors	DRR	Full CT volume box	Individually aligned box	Composite	NCC (256^2 pixels)
GeForce GT 440	96	Single bone (avg.)	59.9 ms	6.47 ms	--	6.45 ms
		Full foot (28 bones)	1678 ms	181 ms	10.6 ms	
GeForce GTX 460 v2	336	Single bone (avg.)	17.3 ms	1.87 ms	--	3.21 ms
		Full foot (28 bones)	483 ms	52.4 ms	5.37 ms	
Tesla 2050	448	Single bone (avg.)	13.1 ms	1.33 ms	--	6.62 ms
		Full foot (28 bones)	368 ms	37.3 ms	4.29 ms	

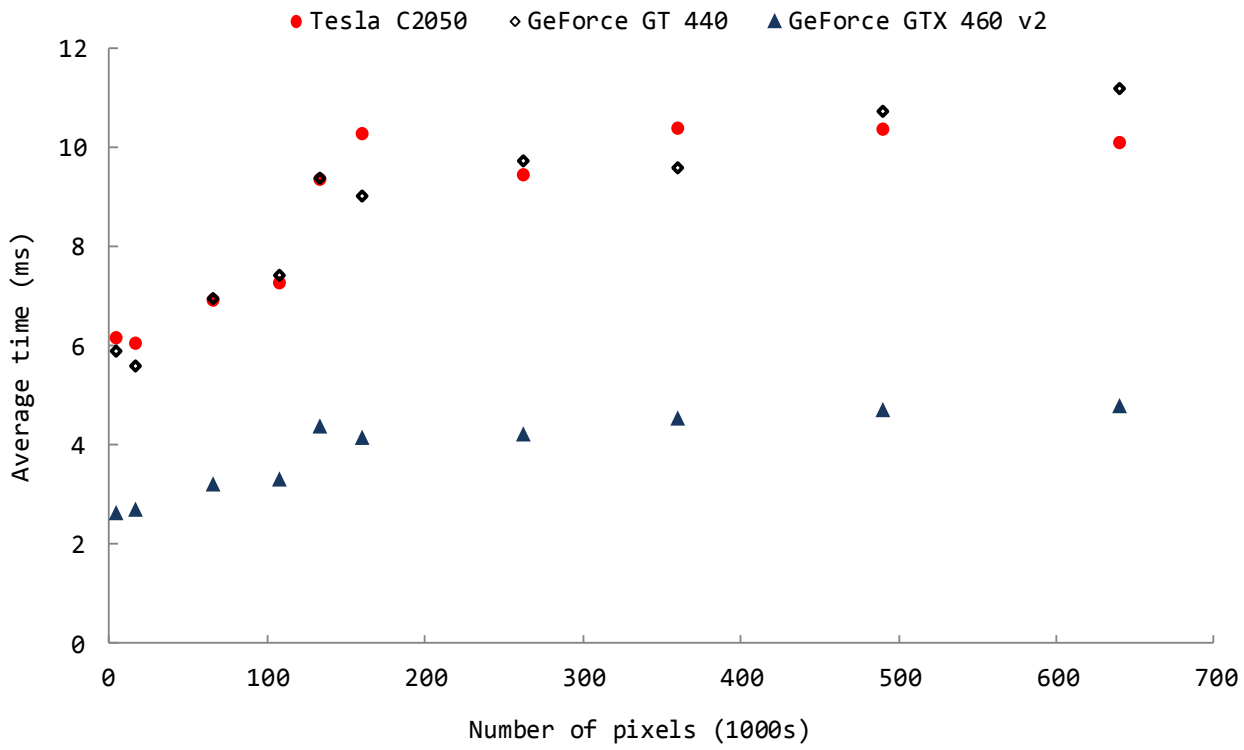


Figure 8.19. NCC timings as a function of the number of pixels included. Timings (averaged over 1000 iterations) for the NCC computation as a function of the number of pixels in the region of interest. Note that the NCC computation does not scale with the number of streaming multiprocessors (see Table 2.1).

DRRACC was designed to accelerate the bottleneck in the 2D-3D registration problem; the generation of DRRs. The CPU version of the code averages 750 ms per DRR, while DRRACC averages as little as 0.033 ms[28] per DRR (Tesla K20 GPU), a 10,000-fold speedup, for a CT stack with ~23.8 x 10^6 voxels and an intensifier screen with ~1 x 10^6 pixels. Even using the GPU card with the least number of CUDA cores we have available (GeForce GT440 – 96 cores), DRRACC is capable of averaging 6.47 ms per DRR, still a mind-numbing 100-fold speed up over the CPU implementation (that includes MATLAB script).

[28] This timing was performed using the test bone (tibia) shown in Figure 8.18.

Table 8.4. Relative timings for five GPUs used during testing.

Card	TeraFLOPS	Number of CUDA Cores	Relative Timings (ms)
GT 440	0.311	96	6.47
GT 640M	0.48	384	5.02
GTX 460 v2	1.045	336	1.87
Tesla C2050	1.288	448	1.33
Tesla K20	**3.52**	**2496**	**0.033**

Table 8.4 summarizes the relative timings and card information. It is worth noting that all GPUs except the Tesla K20 are considered to be consumer cards and as such, are not optimized for parallel computing [44]. The reported 10,000-fold speedup for DRRACC is realized by employing a NVIDIA Tesla K20 GPU with 2496 CUDA cores. While this speedup will be capitalized on during DRR computations, the overall time to process a 60 frame gait record will not decrease by an equivalent amount due to other operations, such as compositing and merit function evaluation, that take place within DRRACC.

Figure 8.20a plots the relative time needed to compute a single DRR as a function of the number of teraflops (trillions of floating point operations per second) reported by NVIDIA [44, 45]. Figure 8.20b plots the relative time needed to compute a single DRR as a function of the number of CUDA cores reported by NVIDIA. The figures illuminate the fact that CUDA cores alone cannot be used to predict timing and suggest that teraflop ratings provide a more reliable estimate. However, many other factors not considered in this analysis (e.g., Compute Capability) should be assessed prior to making any assumptions on related to relative GPU timings.

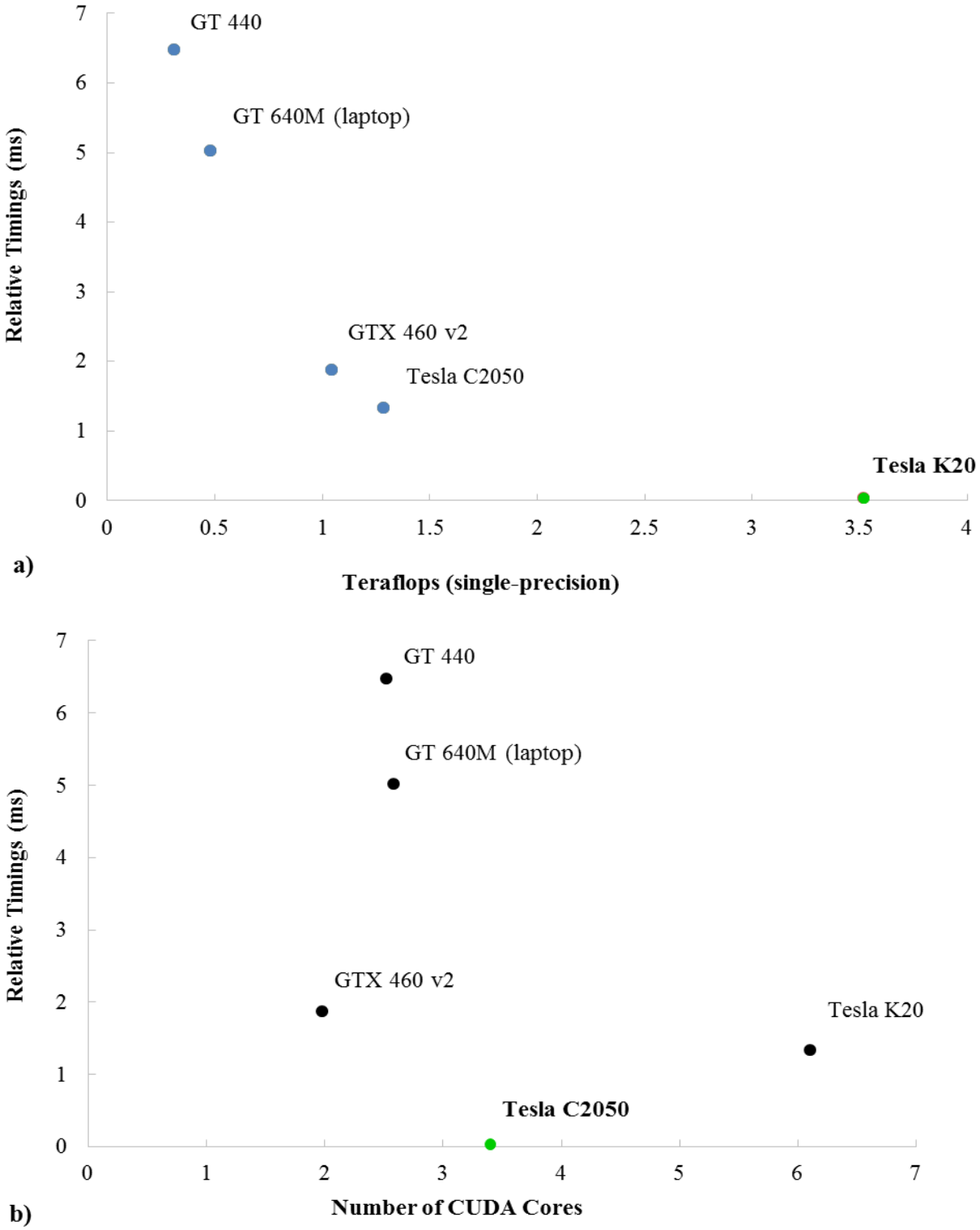


Figure 8.20. Relative timings based on teraflop rating and number of CUDA cores. (a) Relative single-bone DRR timings (averaged over 1000 iterations) for DRRACC versus the manufacturer-reported GPU teraflop ratings. (b) Relative single-bone DRR timings for DRRACC versus the number of manufacturer-reported CUDA cores. The stark differences in timings between the GT 640M, GTX 460 v2 and Tesla C2050 reveal that CUDA cores alone cannot be used to accurately predict performance gains.

8.3.2 Stage translation and rotation

The stage translation and rotation studies contained between 130-170 frames of imaging data. An automated batch processing algorithm was implemented in DRRACC to facilitate "set it and forget it" data analysis. A built-in timer function was used to record the runtime of DRRACC over the course of the entire validation set, which included all computations associated with DRR generation, image composition, and NCC computation, in addition to all of the optimizer's operations. Impressively, DRRACC's timings for processing the full 130-170 frames were measured in minutes, rather than in hours as was the case for the CPU code. DRRACC averaged 30 frames per minute with total runtimes ranging from ~5-6 minutes, versus the CPU code that averages 0.2 frames per minute with a total runtime of ~14 hours..

8.3.3 Performance Scaling with Multibone Optimization

After validating DRRACC's precision and demonstrating a 2½ order of magnitude reduction in computation runtime during single bone optimization, it was logical to question how that performance might scale with problem complexity. Unlike CPUDRR, DRRACC is capable of running multibone optimizations, meaning that an entire foot comprised of 28 bones can be registered simultaneously in a single invocation of the optimizer.

DRRACC was employed to process a living subject's gait data. Initially, a single bone was chosen to determine the baseline timing (Figure 8.21), with subsequent tests including one additional bone per iteration up to a total of four bones. Figure 8.22 plots total optimization time as a function of the number of bones. It should be noted that the quality of registration for the multibone experiments was relatively poor. Nonetheless, these data were used to quantify the effect of multiple bones being optimized in parallel.

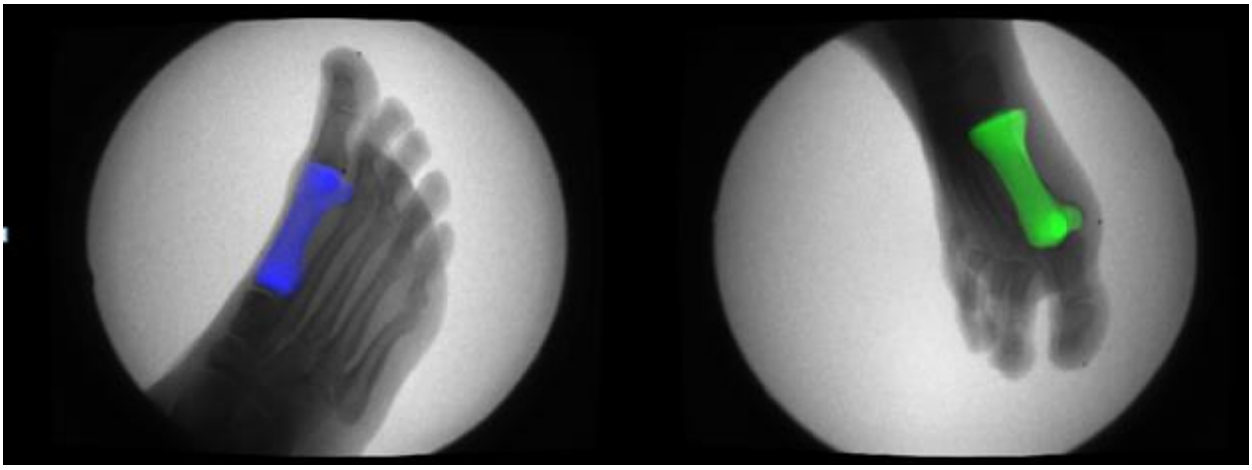


Figure 8.21. Single bone used during benchmark testing for multibone optimization. The multibone optimization routine included the medial cuneiform, and the first ray (metatarsal, and proximal and distal phalanges).

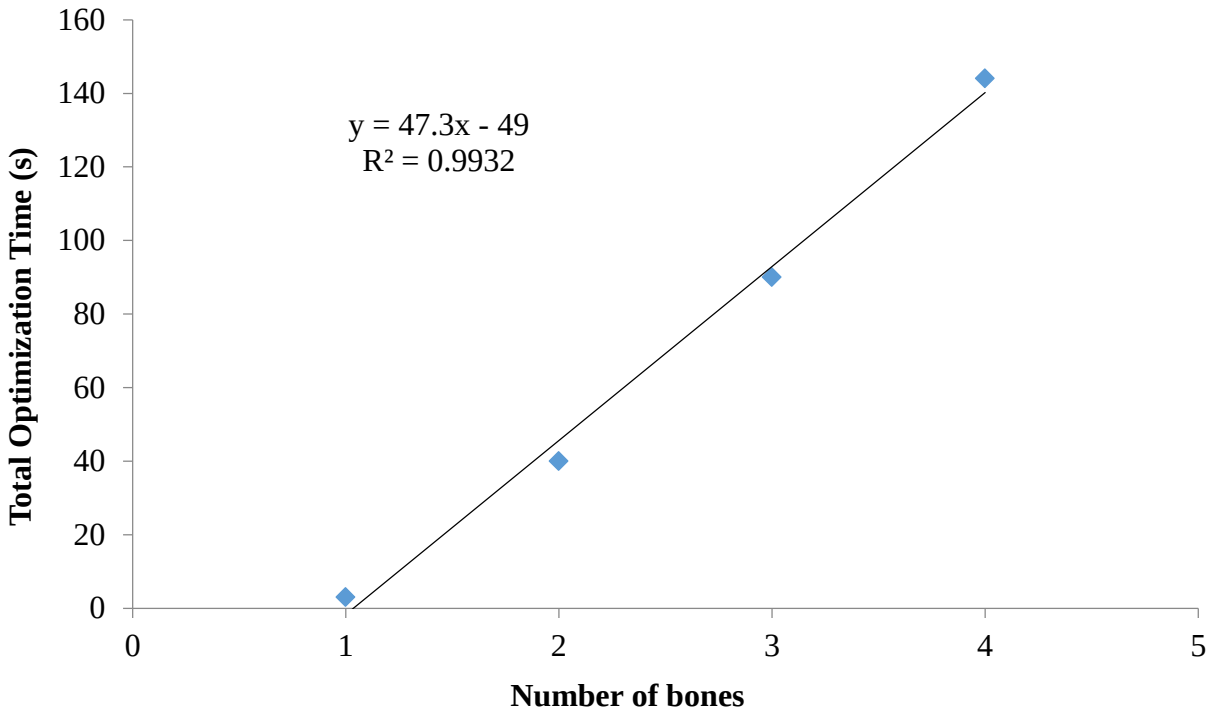


Figure 8.22. Total optimization time as a function of the number of bones being optimized. The large coefficient suggests that DRRACC's multibone functionality may reduce single bone optimzation performance gains by over an order of magnitude (150x to 6.5x).

The results shown in Figure 8.22 indicate that optimization time scales linearly with time for the first four and suggests that processing of four bones – using the multibone approach would require ~7 hours (2.4 minutes/frame x 170 frames / 60 minutes) versus ~57 hours for the CPU's

current single bone approach (5 minutes/frame x 4 bones x 170 frames / 60 minutes). This brings the 150x speedup down to a noteworthy, but slightly less satisfying speedup of ~8x.

8.3.4 Card Analytics with Visual Profiler

A brief study was performed using the NVIDIA Visual Profiler [NVIDIA]. This powerful tool can be used to characterize CUDA C/C++ applications in terms of CPU and GPU activity, memory transfers, CUDA launches and more [NVIDIA]. The profiler is especially useful when trying to determine the utilization of Streaming Multiprocessors. Performance metrics such as the ratio of GPU activity time to total application time and memory throughput were helpful in identifying bottlenecks. Figures 8.23a-b provide screen-captures the Visual Profiler after it was applied to DRRACC with the GUI (8.22a) and without the GUI (8.22b). The region defined by a series of vertical, red line segments encompassed by the green rectangle in Figure 8.23a represents times DRRACC execution when the GPU was actively computing updates to DRRs for subsequent display on the GUI. Conversely, the region in the green rectangle in Figure 8.23b reveals that DRRACC is barely utilizing the GPU; this was expected as DRRACC without the GUI simply computes the DRRs once before completing.

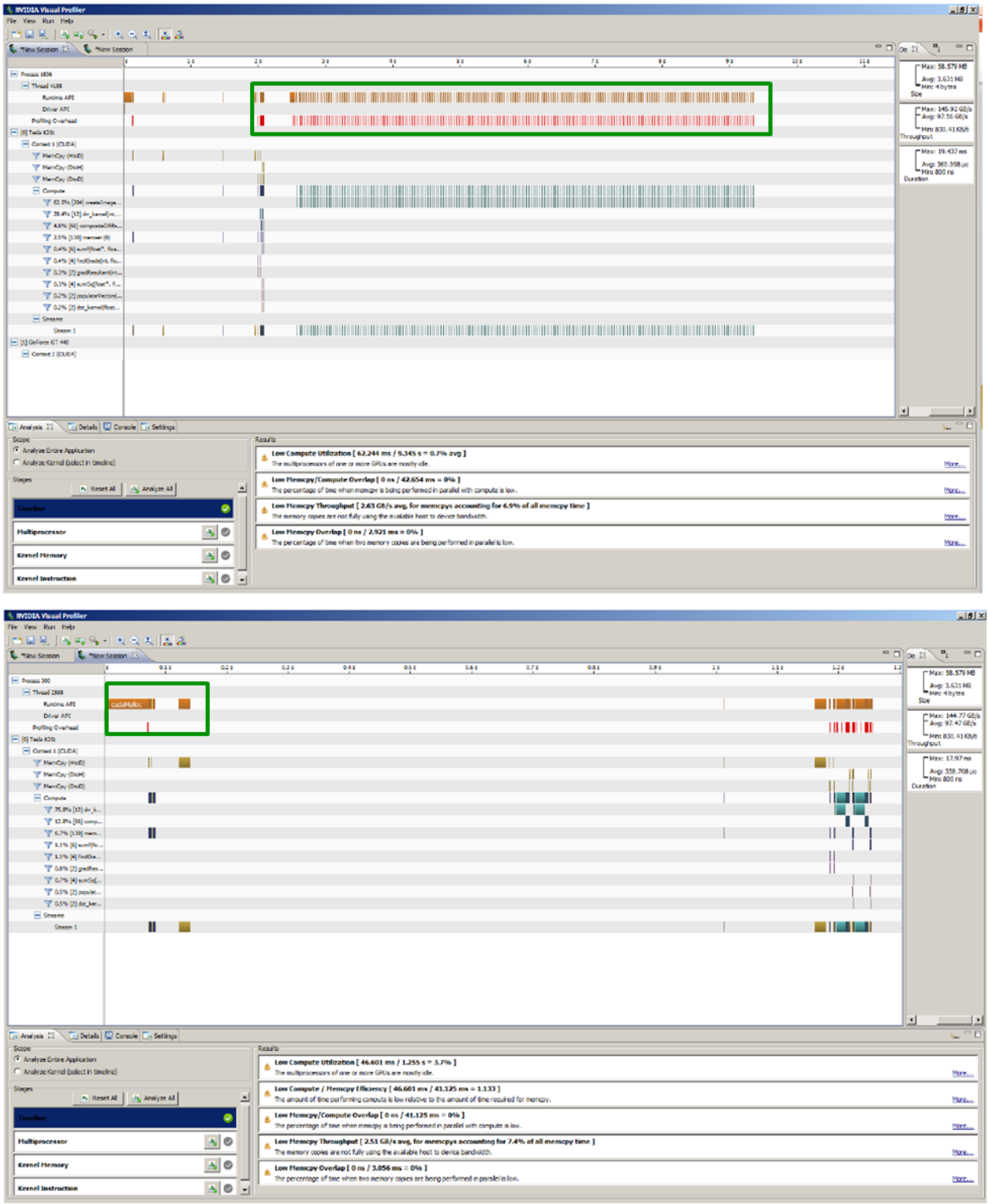


Figure 8.23. Screen-captures of NVIDIA's Visual Profiler. (a) The profiler was employed to analyze DRRACC with the GUI and (b) without the GUI.

Chapter 9: Parallelized Distance-Based Tools

The often heavy dependence on user interaction, particularly when modeling cartilage, repeatedly leads to excessive time spent when creating anatomical soft tissue models. Cartilage models based on user distinction of bone and soft tissue frequently suffer from educated, however subjective, isolation of tissue in each slice of the CT scan or MRI stack. Manually identifying soft tissues can be a daunting task, commonly requiring the user to carefully examine hundreds of slices and characterize cartilage on a pixel-by-pixel basis. Automated approaches to cartilage generation are needed to not only reduce user-bias introduced to models, but also to increase overall computational efficiency and modeling capacity for subsequent analyses. These improvements in biological modeling will improve the quality of the models, increase the effectiveness of the research, and minimize wasted time spent on monotonous tasks.

Currently in the forefront of the automated modeling movement are distance-based approaches that generally rely on computing the distance field for a large data set (i.e., a set of polygonized bone models). The construction of a finite distance field on such a data set is extremely expensive, computationally and temporally, and may require days of runtime on a typical computer. While these methods are far more efficient than manual segmentation, they are lacking in terms of customizability; global parameters require each joint to be modeled identically. Furthermore, the application base is limited to modeling contact surfaces and cartilage elements. Researchers continue to develop increasingly accurate models of the human anatomy; the inclusion of joint capsules in a Finite Element (FE) model is a prime example. Preliminary studies have shown that using joint capsule models during kinematic FE analysis enables the simulation of synovial fluid, an anatomical artifact that is often overlooked in

computationally derived models. This finding may prove to be essential for accurately representing joint articulation.

One of the major drawbacks to modeling biological structures is the need to efficiently manipulate large data sets; a brute-force approach for creating a signed distance field for a single triangulated bone model easily requires over 200 million Euclidean distance computations. With a large majority of software packages, including most Computer Algebra Systems (CAS), running on the Central Processing Unit (CPU), massive computations become quite cumbersome and can run for days at full CPU capacity. An approach that is gaining traction in today's digital world is that of parallel processing by utilizing the computer's Graphics Processing Unit. GPU computing provides an elegant way to transform serial programs into parallel programs – essentially simultaneous rather than sequential computing. The advantage is extreme computational effectiveness; runtime gains of up to 2 orders of magnitude can be expected when CPU code is rewritten to run on the GPU. Computations that take minutes when executed on the CPU can now be performed and displayed in real-time. This concept, real-time computing, will permit accelerated advancements to the field of automated computational biomechanical modeling.

9.1 Literature Review

9.1.1 Point to Triangle Distance

Finding the minimum distance from an arbitrary point to a triangle in Euclidean three-space is a common problem faced by computational geometrists working with polygonal meshes, or a collection of ordered polygons that define the surface of an object [74-81]. Applications span a variety of fields ranging from the generation of (un)singed *distance fields* [75, 79-85], to measuring errors between surfaces [78, 86], to patient specific modeling of human articular

hyaline cartilage [36, 38], to real-time robot path planning [87]. While the actual computation of a point to triangle (PTT) distance may at first seem trivial, a number of caveats often result in a complicated and inefficient solution. One of the complexities associated with this problem is due to the fact that for a single triangle and a single point in space, there exists up to seven unique distances between the two entities. Figure 9.1 represents, pictorially, the seven *Voronoi regions* of a triangle; one for each edge, one for each vertex, and one for the plane of the triangle. The PTT distance computation is fundamental to a variety of novel and exciting applications, and serves as the foundation to many complex algorithms. Of the many algorithms that rely on this seemingly trivial distance computation is one that will be discussed in great depth in this dissertation proposal, distance field generation. A distance field is a scalar field that, for each point within the field, provides the signed or unsigned distance to the nearest point on the surface of an object that lies within the field [82].

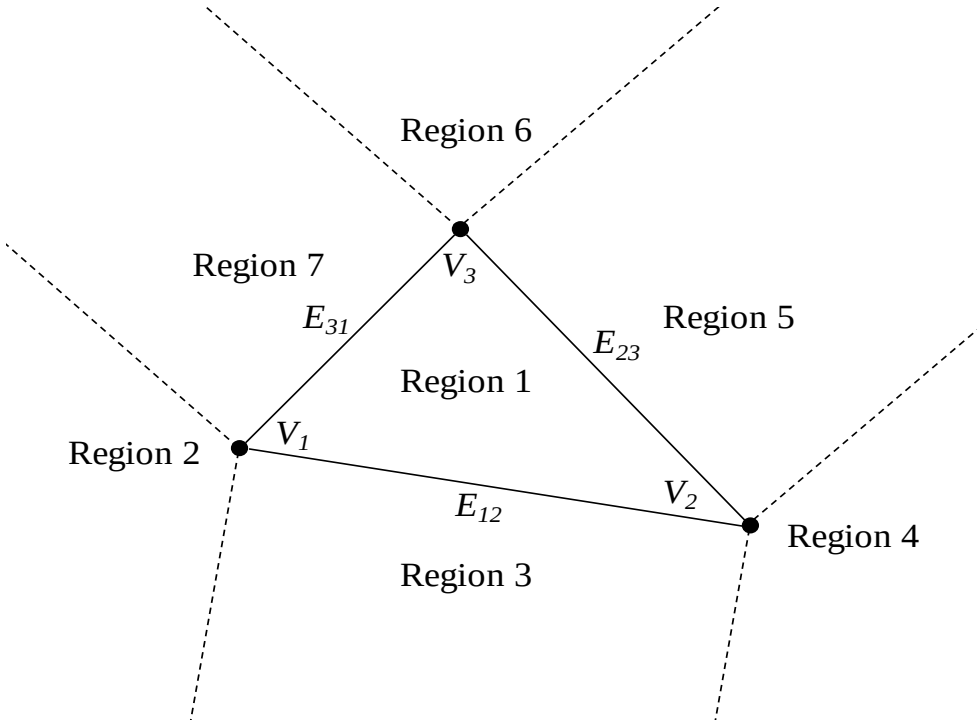


Figure 9.1. The 2D generalized Voronoi diagram (GVD) for a triangle.

Perhaps one of the most widespread and most cited point to triangle (PTT) distance computations was developed by Jones [76]. Jones proposed two methods for computing the distance from an arbitrary point to a triangle. The first, a 3D method, involves projecting the point onto the plane of the triangle and determining whether the point lies within the bounds of the triangle. If the point lies within the triangle, e.g., the trivial case, the distance is found simply by computing the perpendicular distance from the point to the plane of the triangle using standard formula. If the point does not fall within the trivial case, the nearest edge or vertex on the triangle is found by determining the point's location in relation to three vectors that are orthogonal to each other and pass through the three vertices of the triangle. Each of the three vectors requires four inner products and two divisions. Once the point's relative location is known with respect to the triangle's features, the distance is computed. This algorithm is significantly less efficient than his 2D approach described in the following paragraph.

Jones' second approach, a 2D method, determines the distance by first translating and rotating the triangle such that vertex 1 lies on the origin, vertex 2 lies on the *z*-axis, and vertex 3 lies in the *yz*-plane (Figure 9.2). Distance calculations proceed with the determination of whether or not the point lies within the bounds of the triangle. For the nontrivial case—again, when the point lies outside the bounds of the triangle—the triangle's closest feature to the point is determined by the edge equation presented by Pineda [88]. One drawback to the 2D version is the fact that all results are in the transformed space, and for accurate nearest feature identification, an additional transformation back to the original coordinate system is necessary. This 2D approach only requires one squared distance to be computed and thus is much more efficient than his 3D approach.

Transformation of the triangle and/or the point into a new coordinate system is commonly used when finding the PTT distance. Fuhrmann's [81] approach is similar to Jones'; the 3D problem is converted into a 2D problem by way of triangle coordinate transformation. The author's method is dissimilar in that he removes one rotation from the process. As mentioned above, the method described by Jones in [76] requires that vertex 1 lie on the origin, vertex 2 lie on the *z*-axis, and vertex 3 lie in the *yz*-plane. The author removes the second step of Jones' 2D approach: vertex 2 is not required to lie on the z-axis, which results in one less rotation operation leading to a more efficient algorithm. However, the process relies on sluggish transformation matrix operations that are applied to the original coordinates of the triangle, thus reducing the computational efficiency. In addition, the point must be transformed into the same coordinate system as the triangle, further reducing the overall efficiency of the algorithm.

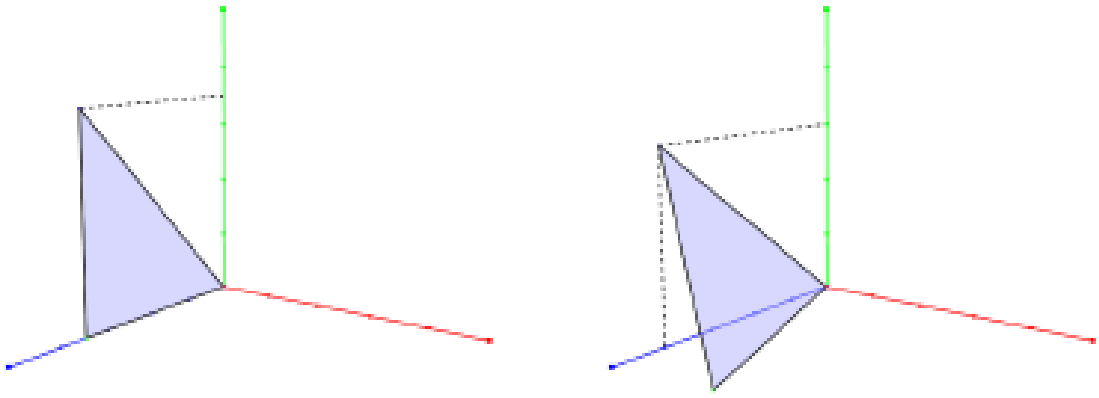


Figure 9.2. Point-to-triangle distance transformation strategies. Comparison of the transformation approaches proposed by Jones [76] (left) and Furhmann [81] (right). reproduced from [81].

Payne and Toga [75] suggest an approach that is also very similar to that of Jones'; the original 3D problem is decomposed into a 2D problem by transforming the triangle coordinates. However, this transformation is completed into the *xy*-plane. The point is then classified as either inside or outside of each edge, which determines the Voronoi region the point lies in (see Figure 9.1). If the point lies within the bounds of the triangle, the distance is simply the perpendicular

distance from the point to the plane, prior to transformation into an alternative coordinate system. If the point lies outside of the bounds of the triangle, the authors identify the nearest feature and calculate the in-plane distance to the point. Finally, the total distance from the point to the triangle is found by means of the Pythagorean Theorem; the square root of the perpendicular distance squared plus the in-plane distance squared. Again, the added burden of coordinate transformation reduces the computational efficiency of the algorithm, but this approach is overwhelmingly common in practice.

Eberly [74] suggests transforming the triangle to a new coordinate system, with vertex 1 lying on the origin, vertex 2 lying on the Euclidean 2D coordinate (1, 0), and vertex 3 lying on the same coordinate system at (0, 1). Following this transformation, a squared distance function for any point lying on the triangle to the point is computed. The minimum PTT distance is found by optimizing the squared distance function. While the author's approach is unique, the already challenging problem of finding the PTT distance is further burdened by unnecessary transformations and inefficient optimizations.

Another transformation technique relies on the implementation of Barycentric Coordinates [79, 89]. It is worth noting the key advantage of using Barycentric Coordinates for this type of computation; Barycentric Coordinates enable a simple comparison of triangle areas created from the vertices of the original triangle to the point, and this comparison results in the identification of the nearest feature on the triangle. However, the point must lie within the plane of the triangle which requires yet a wasteful transformation. Following the transformation and identification of the nearest feature, standard distance formulas can be applied.

The methods proposed by Jones, Fuhrmann, Payne and Toga, and Eberly, in addition to the Barycentric Coordinate approach, all have a common theme; they require transformation of the

original triangle coordinates and, in most cases, the point, into a new coordinate system. The reason for relying on this transformation is logical in that it reduces the complexity of the problem, typically by a dimension. However, this step reduces computational efficiency, indicating that there is the capacity to improve the runtime of a PTT distance algorithm by removing the need to transform the triangle and point into a new coordinate system.

An alternative approach that does not require coordinate transformations has been suggested by Aspert et al. [78]: their PTT distance computation uses four pseudo-planes to segment the triangle into five Voronoi regions (Figure 9.3), as opposed to the more common six pseudo-planes and seven Voronoi regions [80]. This method does not provide *nearest feature recognition* from the triangle to the point, which may be necessary in some applications (i.e., collision detection, troubleshooting, etc.). Additionally, there seems to be a mistake when the authors describe the origin of the four pseudo-planes defining the five Voronoi regions; there is no explanation regarding the computation of the fourth plane. The lack of information makes it very difficult to reproduce their method for comparative purposes.

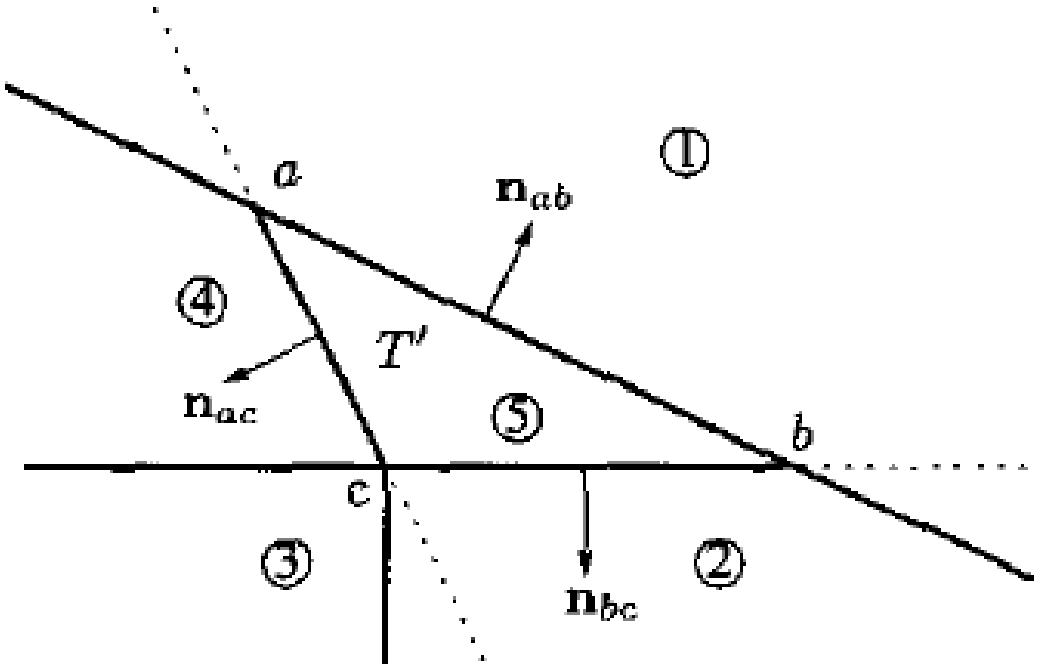


Figure 9.3. Aspert's version of the triangle's Voronoi diagram. Reproduced from [78].

9.1.2 Distance Fields

A distance field is a scalar field that, for each point within the field, provides the signed or unsigned distance to the nearest point on the surface of an object that lies within the field [75, 79-85, 90-92]. Distance fields span a broad application set ranging from computer graphics [82] to path planning [84, 87] to biomechanical modeling [36]. Sud et al. [84] described the problem of computing distance fields in two loosely characterized ways. The two types of problems are classified by data input; the first type of input for the object contains information on a grid or voxel (i.e., a bitmap or skeletal model), whereas the second type of input contains surface information in the form of a triangulated model (i.e., *.stl or *.ply file) [84]. For the purposes of this dissertation research, only the latter will be examined in detail. A cross-section of an object's distance field is shown in Figure 9.4.

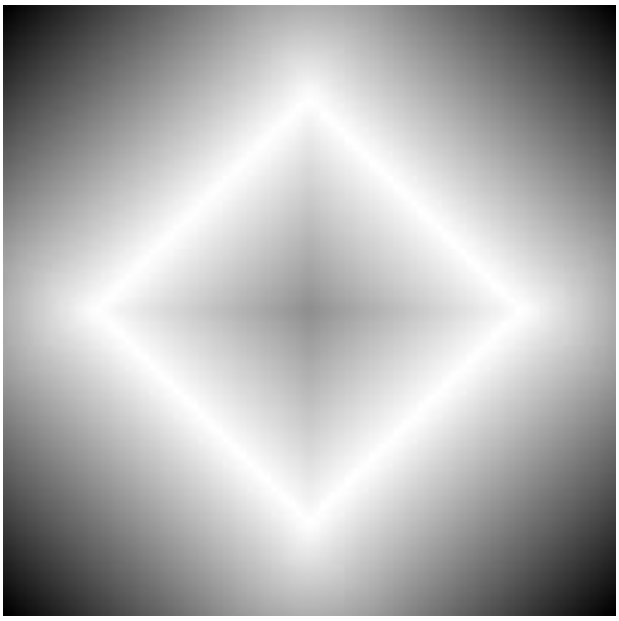


Figure 9.4. A finite distance field for a tetrahedron centered at (0, 0, 0). The cross-section was taken at z = 0 and the grid resolution was 2003 points. The thick white line indicates the object's surface (distance = 0). As distance increases on either the outside or inside of the object, the field's color tends to black. The thin, dark cross centered in the square represents the tetrahedron's geometric skeleton, which will be introduced in a subsequent section.

Hoff et al. [93] propose a type of brute-force approach that, in essence, computes a discrete generalized Voronoi diagram (GVD) by meshing the 3D distance function for each geometric primitive contained within the input. Line segments are broken down into two endpoints and a line, and curves are approximated by line segments. This approach, also known as *distance meshing* [92], has been implemented in a GPU environment, allowing for efficient and interactive computation of 3D distance fields (see Figure 9.5). Volumes require the distance field to be computed in a slice-by-slice manner. However, this method does not produce a signed distance field and the tessellation of the individual distance functions introduces a finite, albeit controllable, error.

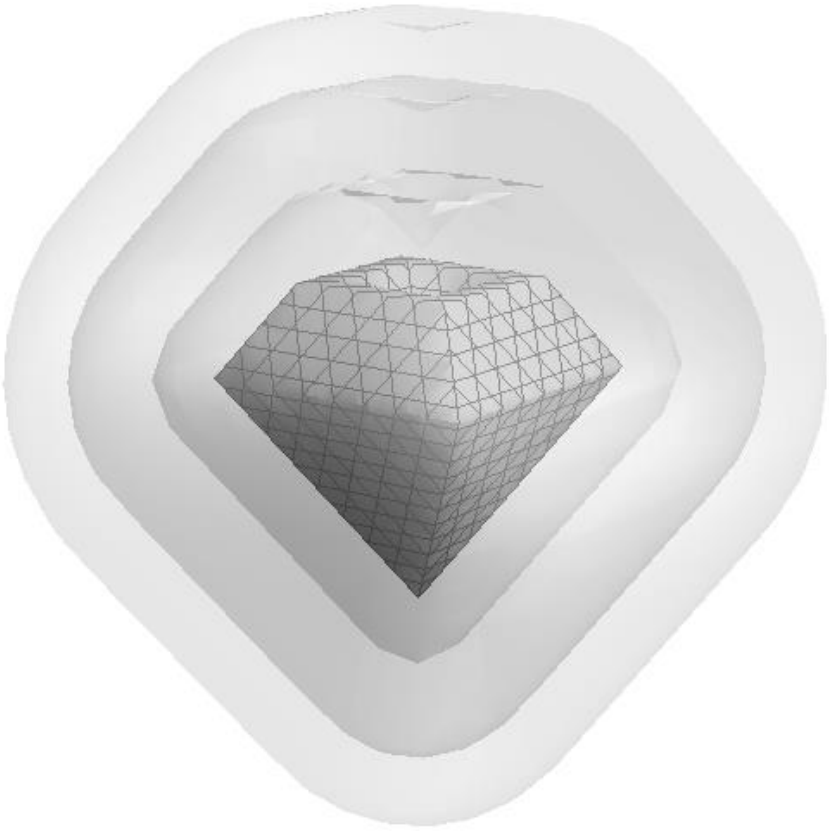


Figure 9.5. A family of offset surfaces (distance field) for the object at the center.

Mauch [90, 94] introduced the characteristic scan conversion (CSC) method that relies on three particular polyhedra. These polyhedra are based on geometries that are created by 3D Voronoi regions for a triangle. A triangle's three characteristic primitives include the face of the triangle (i.e., a plane), the edges, and the vertices, with the corresponding characteristic polyhedra being a triangular prism, a wedge, and a cone, respectively. The author suggests using characteristic polyhedra that are slightly larger than the 3D Voronoi regions; a Voronoi region contains exactly the points that are closest to the geometric primitive in question, whereas Mauch's [90]characteristic polyhedra contain at least the points closest to the geometric primitive [90]. This results in an overlap of the characteristic polyhedra and this overlap eliminates the possibility of missing a grid point during distance field generation due to computational roundoff errors. Each geometric primitive in the polygonized model is converted to its respective

polyhedron. The characteristic polyhedra are then scan converted, with the closest point on the manifold and the corresponding distance being stored. This method is limited to a distance shell that is specified by the user, however, it provides an accurate approach for creating signed distance fields.

Sud et al. [84] integrated the algorithms presented by Hoff et al. [93] and Mauch [94] to create a "cull and clamp" type algorithm for the fast computation of signed distance fields. The algorithm sweeps through the volumetric model, slice-by-slice, in the *z*-direction. The Voronoi diagram is computed for each "site" (i.e., geometric primitive) in each slice. Sites are "culled" if there is another site with a smaller distance to a particular grid point for a particular slice. Voronoi regions are "clamped" to reduce the number of computations. The authors claim an up to two order of magnitude reduction in runtime when compared to typical CSC algorithms.

Erleben and Dohlmann [92] present a GPU-based method that relies on tetrahedra to create a "narrow-band shell" distance field. Essentially, the algorithm isolates each triangle from the collection of triangles making up the surface of the object. The triangle in question is then enclosed within a bounding rectangle that is oriented on the same plane as the triangle. The user specifies the narrow-band distance and the rectangle is extruded the specified distance in both directions, thus creating what the authors refer to as an "oriented bounding box." This bounding box is decayed into five tetrahedra and a scan conversion algorithm is applied to generate the distance field. The algorithm produces accurate signed distance fields, however, the distance filed is limited to a user-specified narrow-band shell much like Mauch's CSC algorithm. The creation of a narrow-band shell distance field renders it unfit for some applications.

Danielsson [95] introduced *Euclidean distance mapping* using sequential algorithms and a four-pass scan distance propagation approach. Frisken et al. [91] present a method referred to as

Adaptive Distance Fields or simply "ADFs." ADFs enable higher resolution sampling at fine details on the surface model and lower resolution sampling at smooth locations. In addition to adaptive sampling, the authors discuss data storage hierarchy; the coupling of the two ideas results in an efficient distance field algorithm. Further discussion and approaches not discussed in this dissertation proposal can be found in the literature [79, 96, 97].

9.1.3 Cartilage Modeling

Previous attempts at modeling cartilage by our group have either required the user to fill the inter-bone space in the model with a solid body, followed by subtracting the bones and manually trimming the shape to simulate the joint [22, 23] or the user had to manually segment cartilage slice-by-slice on both sides of the joint [24, 25]. While the former method was anatomically-based, it was not physiologic (i.e., cartilage is not a plug between two bones). The latter method was much closer to the actual cartilaginous anatomy; however, it was time and labor intensive, and subject to discontinuities between slices, leading to excessively rough surfaces. Additionally, by manually segmenting the cartilage from the data, a layer of user-bias is instilled in the models.

Cheung et al. [26, 27] created a "geometrically accurate FE model" of the foot by way of MR image reconstruction. This method provides the investigator with an anatomical model that is limited by imaging and computational technologies, but relies on segmentation of the data. As previously mentioned, oftentimes this segmentation is performed manually resulting in subjective modeling. The authors employed cartilage elements within the interphalangeal joints, but refrained from including the cartilage elsewhere in the model for reasons not described. The cartilage elements connected the phalanges, essentially creating a continuous structure with layered stiffness and mechanical properties. Beyond the poorly defined cartilage elements was the lack of cartilage in the rest of the model. All other joints or bony articulations were modeled

as elastic bodies with frictionless surfaces and pliable contact stiffness. Their purpose was to use properties similar to that of cartilage, without actually modeling the cartilage.

The volumetric articular cartilage models generated by Anderson et al. [98] were created with a number of user involved tasks. First, “meshing planes” were projected onto the contact surfaces of the bone mesh. The authors used a tetrahedron, with the apex located below the surface of the triangulated bone (see Figure 9.6), to create pathways for the meshing plane projection. Each point on the meshing plane was projected in the direction of the apex of the tetrahedron, ultimately resulting in a quadrilateral surface superimposed on the articular face of the bone. Following this mapping, the quadrilateral surface and the bone surface were meshed together at common FE nodes. The volumetric cartilage mesh was then generated by extruding the surface mesh a user specified distance along the average point normals as determined by the neighboring quadrilaterals.

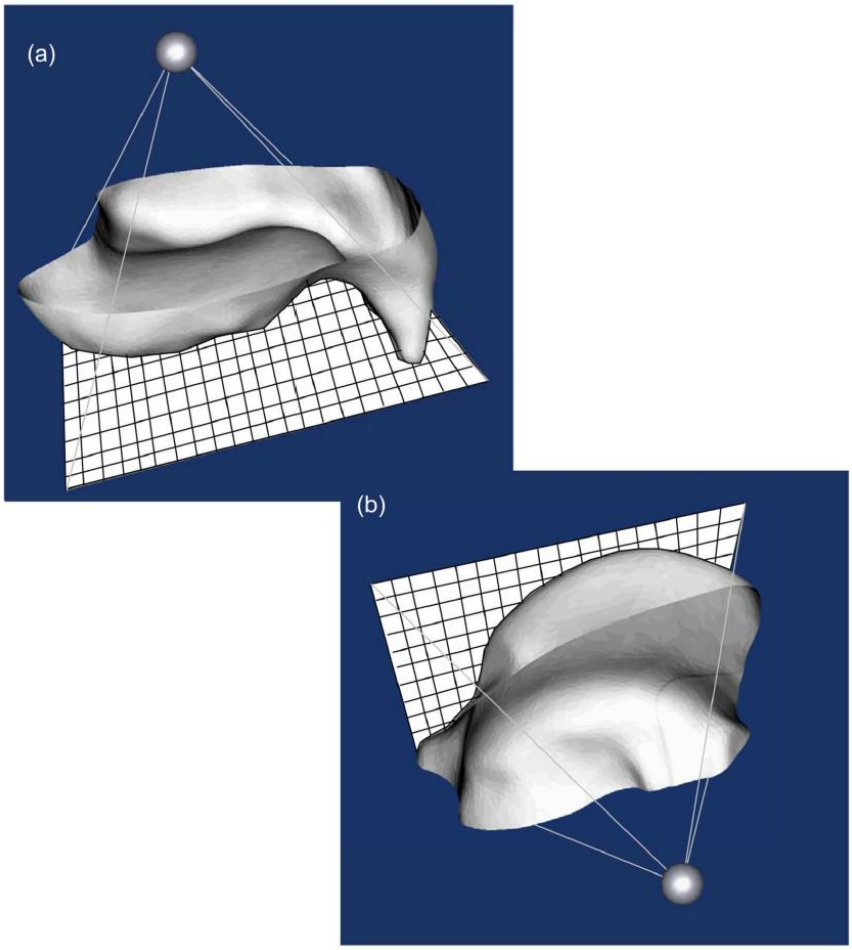


Figure 9.6. Anderson et al.'s approach to cartilage modeling. His approach involves constructing meshing planes onto the bone surface in the direction of the apex that is located below the bone surface. Reproduced from [98].

The previous work on cartilage modeling most relevant to this study was presented by Crisco et al. [36]. Their method identifies contact surfaces by computing the distance field of each bone. After the distance field has been constructed for the bones involved, Crisco et al. describe the use of a distance threshold; distances are computed for vertices on one bone to the adjacent bone. This enables the generation of isocontours which signify contact surfaces. Following the construction of the articular surface, cartilage maps in the form of "height-fields" are created by extruding the vertices along the respective bone surface normal by half the minimum inter-joint distance. The minimum inter-joint distance is defined as the minimum distance between two

bones in a joint. The contact surfaces identified by the author's approach are illustrated in Figure 9.7.

Figure 9.7. Contact surface maps generated by Crisco et al.'s distance field approach. Note that each joint relies on the global distance threshold. Reproduced from [36].

In summary, a number of viable approaches for producing volumetric articular cartilage elements are presented in the literature. The most common methods include manual segmentation and reconstruction of MRI data [24, 25], filling the articular gap with a solid [26, 27], filling the articular gap with a solid and subtracting away the bones [22, 23], articulation surface projections coupled with surface extrusions [98], and distance field mapping followed by contact surface extrusion [36]. The method we present in this paper extends the approach of Crisco et al. [36], by enabling joint-specific distance thresholds.

Cartilage Thickness

Muehleman and Kuettner [99] revealed that in diarthrodial joints, such as the first metatarsophalangeal joint (MTPJ), cartilage varies in thickness based on location and curvature. For example, the distal head of the first metatarsal is convex; centrally, or at and near the apex of the metatarsal head, the cartilage is thicker. Conversely, the cartilage on the first proximal phalanx, which articulates with the first metatarsal in the MTPJ, is shown to be thinnest centrally

and thickest peripherally [99]. El-Khoury et al. [100] performed a comparative study analyzing the accuracy of using MR imaging versus double-contrast multi-detector row CT arthrography in finding cartilage thicknesses in the human tibiotalar joint. The study involved a total of five ankles and cartilage thickness was physically measured at a total of 76 sites. The authors found cartilage thicknesses of approximately 0.4-2.1 mm (1.25 mm average) for the tibiotalar joint. Additionally, the authors revealed that cartilage thickness predictions based on multi-detector row CT versus MR imaging measurements was quite extreme; *R*-values of 0.91 versus 0.70 were found, respectively, when compared to the physical data. Athanasiou et al. [28] studied, in depth, the biomechanical properties, thickness, and histomorphometric characteristics of the first metatarsophalangeal joint. The study revealed that cartilage degeneration is likely not gender specific and that differences in joint cartilage may be attributed to lower limb dominance. The authors found through biphasic creep testing and histological cross sectioning a cartilage thickness average of 0.75 ± 0.21 mm for the first MTPJ.

In summary, joint modeling may require variable thickness cartilage to produce accurate joint kinematics and contact stresses during FE simulation. Cartilage thickness on the proximal surface of the talus is highly dependent on location and multi-detector row CT scans are more accurate at predicting cartilage thickness than MR imaging measurements. The first metatarsophalangeal joint exhibits a wide range of cartilage thicknesses with larger values being found at the apex of the metatarsal head and smaller values being located at the center of the proximal phalanx's articulation surface.

9.1.4 Geometric Skeletal Models

The geometric skeleton (medial axis in 2D or the medial surface in 3D) of a shape can be defined by the locus of points that are each unique to the center of a maximal circle (in 2D) or sphere (in

3D) that is contained within the bounds of the shape and tangent to, at minimum, two points on the shape's boundary [101-106]. This set of points minimally characterizes a geometric shape by encompassing the entity's distinctive boundary information. The skeleton always presents itself in one less dimension than the object it describes (i.e., a circle's skeleton is a point, an ellipse's skeleton is a line, and a tetrahedron's skeleton is the union of multiple sheets, etc.), which allows for the decomposition of complex geometries into the union of simple primitives [101, 102, 104]. This property, among others, makes the skeleton a popular subject of investigation. Skeletal models are commonly used in blending and mesh generation [102], image analysis [103], shape description [104], shape representation simplification [105], and identification of anatomical structures from medical tomography [106]. A skeletal model for an arbitrary geometry is illustrated in Figure 9.8.

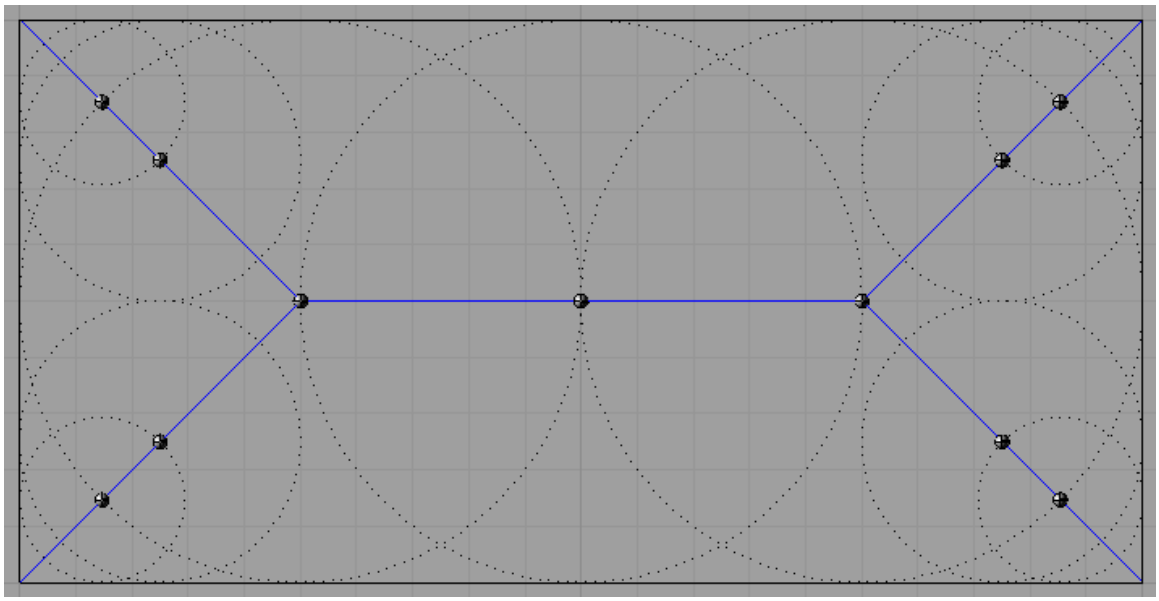


Figure 9.8. Visualization of a geometric skeleton. A rectangle's geometric skeleton in blue and maximal circles located at various nodes on the skeleton.

9.2 Applications

9.2.1 Point to Triangle Distance

The point to triangle distance (PTT) algorithm serves as the foundation for the tools proposed in this dissertation. This algorithm was developed in Mathematica [71] to compute the 3D distance field of a single triangle and is currently implemented on the CPU. The algorithm is comprised of many functions that are called by the main kernel. Generation of the triangle distance field proceeds using simple vector operations; a point in space is characterized as being closest to a particular entity. This approach is considered a brute-force method and essentially relies on classifying which Voronoi region the point is in [93]. The process of determining which entity the point lies closest to is referred to as *nearest feature recognition* (NFR). The beauty of our nearest feature recognition function is its simplicity; at most, the algorithm need only compute the normal to the triangle, nine half-space normals found with a cross product, and nine inner products. At minimum, the nearest feature can be determined with the normal to the triangle, two half-space normals, and a single inner product. The half-space normals are created by finding the normal to the planes that contain the edges of the triangle and the Voronoi regions, and are perpendicular to the plane of the triangle. Figure 9.9 presents the half-space planes used in determining half-space normals. It is worth noting that the half-space planes are not typically coplanar with the Voronoi regions, although special cases do arise that display this property (i.e., a right triangle).

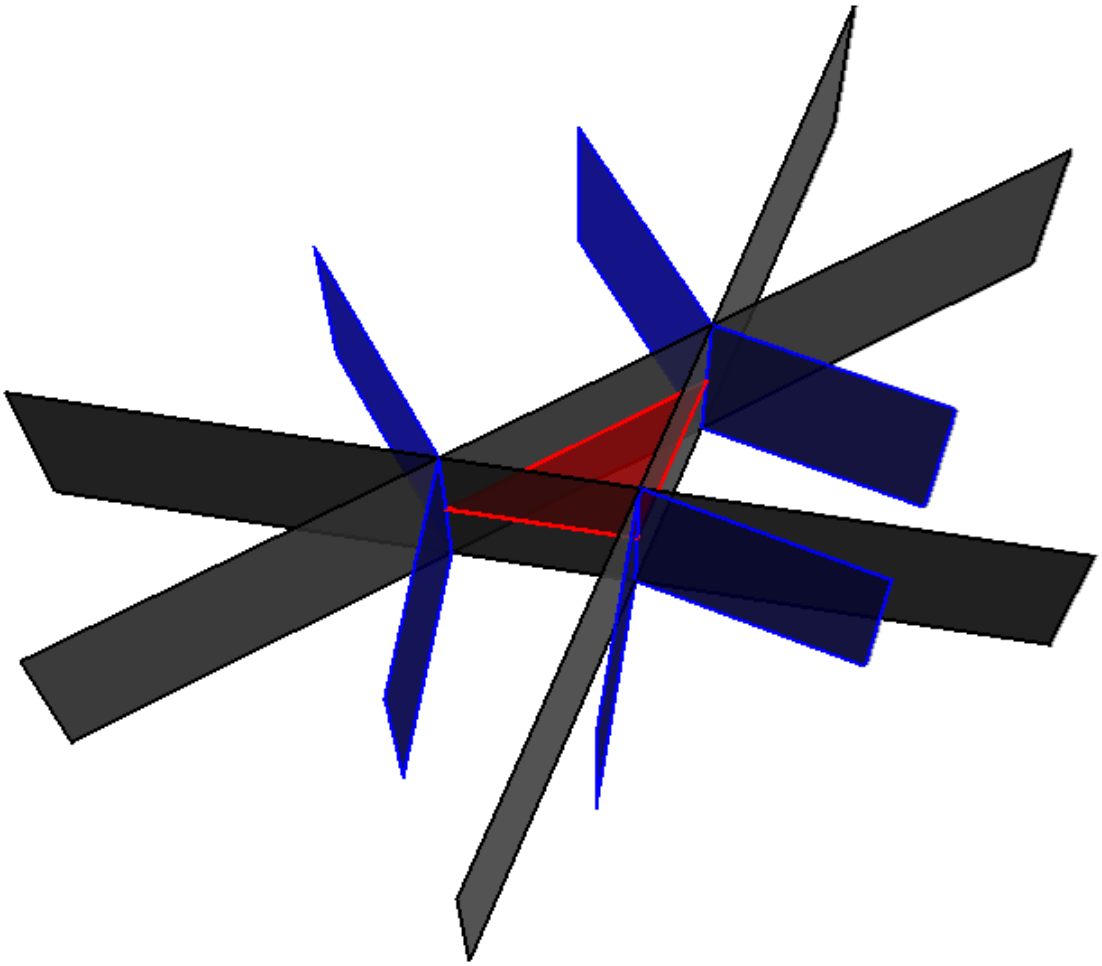


Figure 9.9. 3D Voronoi regions for a triangle. The triangle is shown in red. Half-space planes; the normal vectors to these planes serve as the testing criteria when determining the nearest feature on the triangle in the PTT distance algorithm. The sign of the inner product between the normal to the plane and the vector from a vertex to the point determines "which side of the edge" the point is on. Gray half-space planes determine nearest edge, while blue half-space planes (Voronoi region boundaries) determine nearest vertex. Note the image is in 3D for visualization purposes only.

Following NFR, the algorithm computes the distance using the appropriate distance formula. The subsequent pseudo-code provides a succinct representation of the process.

For p = n

If p on outside of $n_{e,i}$ // Which side of the triangle edge half-space

    If p on outside of $n_{v,i}$ // Which side of the Voronoi region half-space

    p is nearest to v_i // Nearest feature identified; compute distance;

end;

If p on outside of $n_{v,j}$ // Which side of the Voronoi region half-space

p is nearest to v_j // Nearest feature identified; compute distance;

end;

Else rotate right *i, j, k* and repeat above loop

Else rotate right *i, j, k* and repeat above loop

Else p is nearest to the plane of the triangle; compute distance;

end;

The CPU implementation of this algorithm runs with time complexity *O*(n) with respect to the number of points in the grid. This approach, while machine-precise accurate—an advantage over many PTT algorithms—is computationally inefficient if executed on the CPU.

A natural progression was to convert the Mathematica algorithm into CUDA code such that it could be implemented to run on the massively parallelized and computationally powerful GPU. This transformation has enabled the creation of real-time 2D distance fields for a single triangle on a 1000 x 1000 (1,000,000) point grid (Figure 9.10). The GPU handles the task extremely efficiently by eliminating loops and computing distances for each pixel simultaneously. The end result is an interactive 2D triangle distance field that can be manipulated by the user with keyboard and/or mouse commands, real-time.

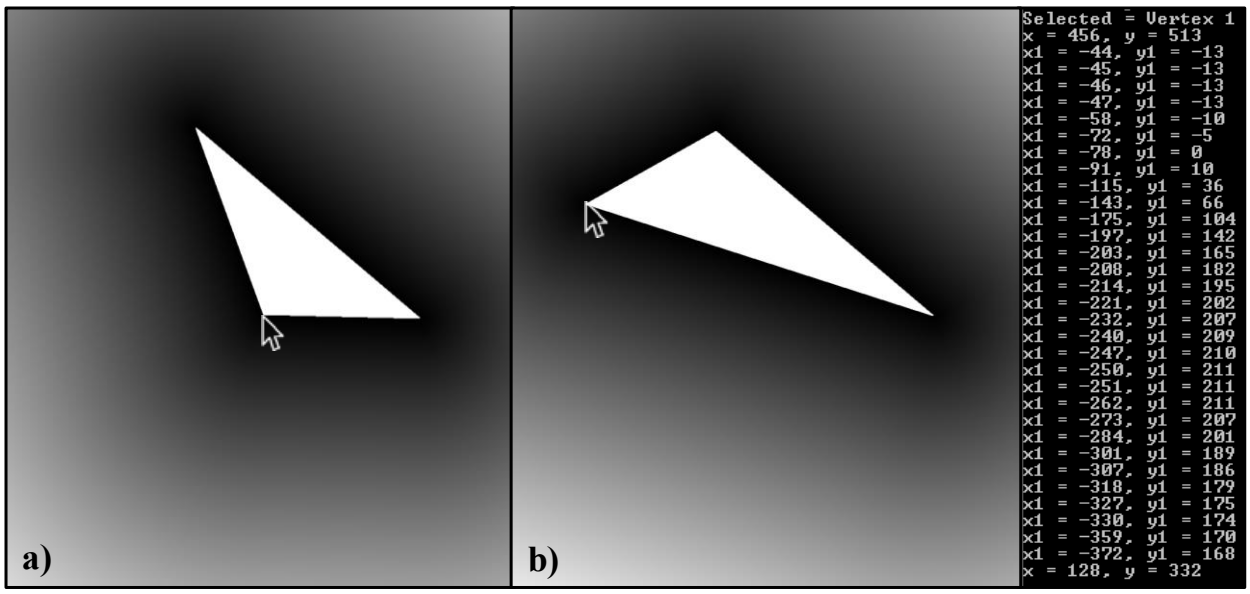
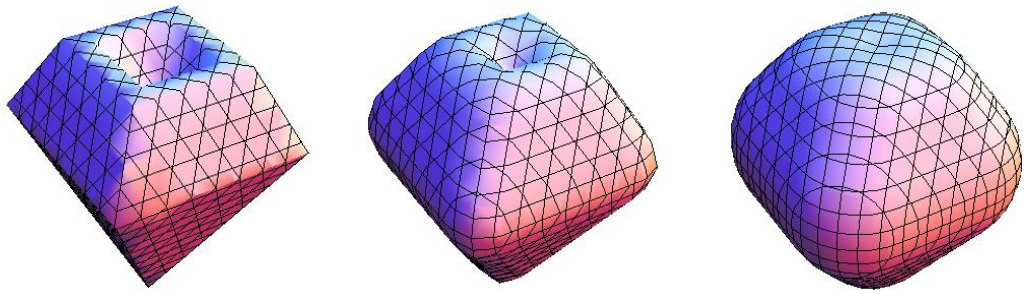


Figure 9.10. 2D distance field visualization for a triangle. The series of images above demonstrate the 2D distance field being manipulated real-time. a) Shows the triangle in its original orientation, b) shows the final position, determined by the selection of a vertex and movement by the mouse. The far right image is the feedback generated by the algorithm; it displays the vertex chosen, the initial coordinates, the path taken, and final coordinates. Distances are considered zero within the triangle (white). Distances near the triangle are shown in black and distance far from the triangle fade to white.

9.2.2 Distance Fields

The distance field, which again is a scalar field of signed distances to the nearest point on the surface of an object [82], is a powerful tool that serves as the workhorse for this dissertation research. Similar to the PTT distance algorithm—the distance field's foundation—the distance field acts as an application hub; the creation of an object's distance field can lead to many fascinating results. Generation of the distance field is currently limited to the CPU, but functional code is currently being utilized to create a number of test fields for further analysis. The distance field algorithm relies on the PTT code; essentially, a triangulated model (*.stl or *.ply) is read-in to the CPU version of the PTT algorithm. The function is then mapped onto the collection of polygons that makeup the surface model. The end result is the generation of a finite, signed, machine-precise accurate distance field. The next step in the progression of distance fields is the conversion of the code to operate on the GPU. This will be a monumental step in the direction of

automated computational biomechanical modeling. Examples of the original geometry and the offset surfaces created by our distance field algorithm can be found in Figures 9.11a-c.



Offset = 0 Offset = 0.15 Offset = 0.5

Figure 9.11. A series of offset surfaces (level sets). A 3D shape (left), with two offset surfaces for comparison (middle and right).

9.2.3 Cartilage Modeling – Single Position Study

The addition of cartilage elements in a finite element model prevents bone-on-bone articulation during simulation, thus providing more accurate information about joint kinematics. We present a semi-automated method for identifying joint articulation surfaces and creating volumetric articular cartilage elements based on patient-specific bone information. The approach identifies contact surfaces based on a joint-specific, user-specified distance threshold criterion applied to a polygonized set of bones. Volumetric cartilage elements are generated using half of the minimum inter-joint distance. We present the method and then apply it to the first ray of the human foot, which includes the medial cuneiform, first metatarsal, and first proximal and distal phalanges. Distance thresholds for the first ray ranged from 3.0 to 4.25 mm depending on the joint and the desired contact surface coverage. Inter-joint distances were found and applied to the contact surfaces to generate uniformly thick cartilage models. Average inter-joint distances of 0.46, 0.72, and 0.51 mm for the first interphalangeal, metatarsophalangeal, and cuneometatarsal joints, respectively, were determined heuristically. For the first metatarsophalangeal joint,

algorithmically computed thickness is in agreement with histologically measured cartilage thickness reported in the literature.

Introduction

This chapter was adapted from a publication in the Medical & Biological Engineering & Computing Journal [107], which is the reason for the sudden shift away from the typical formatting of this dissertation.

The work presented further develops the preliminary findings of a method for semi-automated patient-specific articular contact surface identification based on patient-specific bone information. Minimal user interaction is then used to generate volumetric cartilage model. These cartilage elements are created for use in a complete and anatomically detailed FE model of the human foot [23-25]. Previous attempts at modeling cartilage by our group have either required the user to fill the inter-bone space in the model with a solid body, followed by subtracting the bones and manually trimming the shape to simulate the joint [22, 23] or to manually segment cartilage slice-by-slice on both sides of the joint using magnetic resonance imaging scans [24, 25]. While the former method was anatomically-based, it was not physiologic (i.e., cartilage is not a plug between two bones). The latter method was much closer to the actual cartilaginous anatomy; however, it was time and labor intensive and subject to discontinuities between slices, leading to excessively rough surfaces. Additionally, by manually segmenting the cartilage from the data, unnecessary user-bias was instilled in the models.

The previous work on cartilage modeling most relevant to this study was presented by Marai et al. [36]. This method identifies contact surfaces by computing the distance field for each bone. Following distance field construction, Marai et al. describe the use of a distance threshold; distances are computed for all vertices on one bone to the adjacent bone and those vertices that

lie within the threshold are categorized as the articular surface. Cartilage maps in the form of "height-fields" were then developed by offsetting the vertices along their respective normal, e.g., along the average normal to the bone surface at that point, by half the minimum inter-joint distance. The minimum inter-joint distance is the minimum distance between two bones in a joint. Previous work by Carrigan et al. [108] revealed that this distance was suitable for estimating constant cartilage thickness.

Anderson et al. [98] used "meshing planes" that were projected onto the contact surfaces of the bone mesh. They used a tetrahedron with the apex located below the surface of the triangulated bone to create pathways for the meshing plane projection. Each point on the meshing plane was projected from the meshing plane in the direction of the apex, resulting in a quadrilateral surface superimposed on the articular face of the bone. Following this mapping, the quadrilateral surface and the bone surface were meshed together at common FE nodes. The volumetric cartilage mesh was then generated by extruding the surface mesh a user specified distance along the average point normals as determined by the neighboring quadrilaterals.

Cheung et al. [26, 27] created an FE model of the foot through MR image reconstruction. This method provided an anatomical model that was limited by imaging and computational technologies, but relied on segmentation of the data. The cartilage elements filled the interphalangeal gap, essentially creating a continuous structure with layered stiffness and mechanical properties. All other joints or bony articulations were modeled as elastic bodies with frictionless surfaces and pliable contact stiffness.

In summary, a number of viable approaches for producing volumetric articular cartilage elements have been presented in the literature. The most common methods include manual segmentation and reconstruction of MRI data [24, 25], filling the articular gap with a solid [22, 23, 26, 27],

articulation surface projections coupled with surface extrusions [98], and distance field mapping followed by vertex offsetting [36]. The method we present extends the approach of Marai et al. [36], by introducing joint-specific distance thresholds that are applied to the first ray of the foot. The intermediate purpose of this method is to improve joint kinematics of the model and in the long term, we aim to calculate accurate joint stresses.

Methods

Geometry for the computational foot model was derived from the left cadaveric foot of a 44-year-old male, with no gross deformities or significant degenerative changes based on anterior/posterior and lateral radiographs [24, 25]. CT imaging was performed using a Philips CT MX8000IDT (Philips Healthcare, 3000 Minuteman Road, Andover, MA 01810-1099), with 0.797 mm pixel spacing, 0.3 mm voxel depth, 1.255 pixel/mm resolution, 0.3 mm spacing between slices and a 0.6 mm slice thickness [25]. The cadaveric foot was loaded with a nominal force of 70 N applied normally to the sectioned tibia [25]. This loading was an artifact of another project, which required a 70N load to hold the cadaver foot in place during imaging; however, it has no bearing on this study, as under this small load, neither the cartilage or bones of the foot were altered. The bones in the model included the talus, calcaneus, navicular, three cuneiforms, cuboid, five metatarsals, 14 phalanges, and approximately the distal 1/3 of the tibia and fibula (Figure 9.12). Bone anatomy is represented as Standard Triangulation Language (STL) [109] files with vertex counts ranging from 911 vertices (1818 triangles) for the fourth distal phalanx to 35,940 vertices (71,876 triangles) for the calcaneus. For the purposes of this study, the first ray, including the medial cuneiform, is considered as demonstration context for our method.

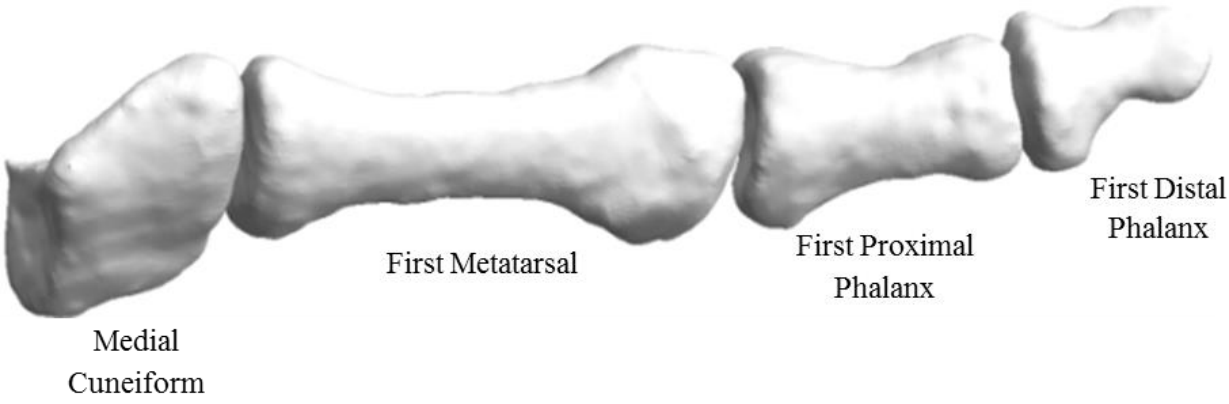


Figure 9.12. Dorsal view of the first ray used in this study.

A distance-based algorithm was developed and implemented on two adjacent STL bone models comprising a joint, e.g., the first metatarsal and the first proximal phalanx [18]. The algorithm was used to identify joint articulation surfaces for the two bones in question and employs a distance threshold specified by the user to isolate polygons in the bone mesh that represent adjacent bone contact areas. At this stage, the algorithm assumes that the bones are separated by a finite gap. In theory, the algorithm should function properly regardless of the presence of an articular gap, as long as there are two disjoint STL models. The algorithmically defined contact surfaces serve as potential locales for "growing" cartilage [36].

High resolution bone meshes were used, with each model storing up to 71,876 polygons. Our approach to identifying contact surfaces relies on a large number of distance computations per joint. In an attempt to increase the computational effectiveness of the algorithm, we limited computations to a specific region of interest. Bones were preprocessed to remove extraneous vertices that were outside of the selected region. An example of a truncated joint can be found in (Figure 9.13). This truncation resulted in runtime reductions of up to one order of magnitude. While this solution did require user interaction, it did not bias the model to the extent of manual segmentation techniques.

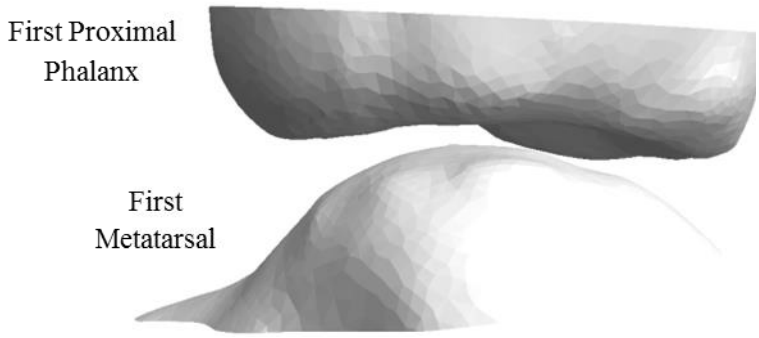


Figure 9.13. Truncated region of interest for the first metatarsophalangeal joint.

After preprocessing the pair of STL models to specify the regions of interest, a task that takes approximately five minutes, the two truncated bone meshes were read into the algorithm for contact surface identification. The algorithm computes distances from all of the vertices on one bone to a single vertex on the opposing bone. Vertices on one bone that are sufficiently close to the other bone are stored for further manipulation, e.g., *if(distance(*$vert_i$ *,* $vert_j$*) < threshold, keep, discard).* If two out of the three vertices that comprise a polygon are within the distance threshold, the polygon is included in the output contact surface. The algorithm traverses the vertex lists until all polygons have been classified as either contact surface or extraneous bone. The algorithm's output is a pair of distance-based joint articulation surface models; the algorithmically identified contact surface for the first metatarsal can be seen (Figure 9.14). It is worth noting that the output pair of contact surfaces is indeed a triangulated subset of the bone surface; subsequent operations must be performed to create volumetric cartilage elements.

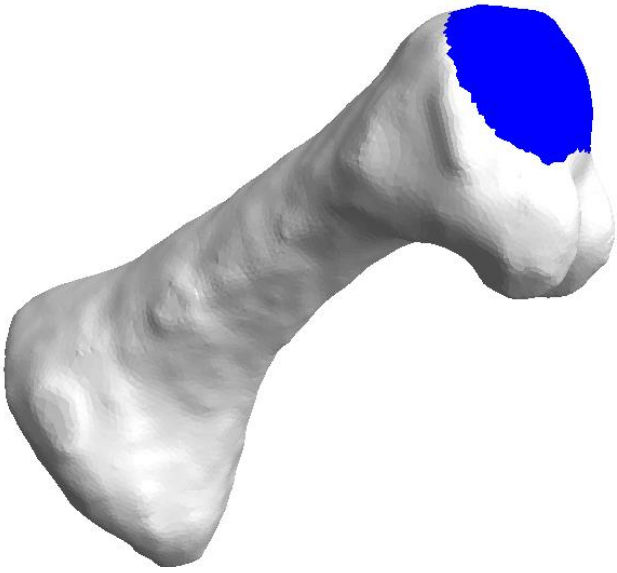


Figure 9.14. First metatarsal with algorithmically identified distal contact surface in blue.

Previous work by Marai et al. revealed that a two mm distance threshold was applicable for the many small bones in the hand. Initial attempts at identifying contact surfaces for the first ray of the foot utilized Marai's proposed distance thresholds of two mm. However, we discovered that this global distance threshold did not adequately describe the joint articulation surfaces for the first ray (Figure 9.15). This finding warranted the exploration of joint-specific distance thresholds.

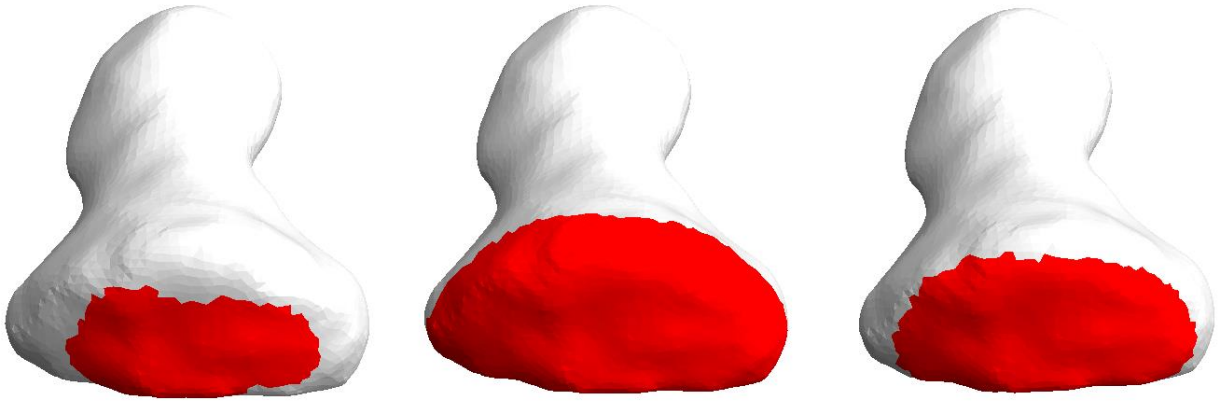


Figure 9.15. First distal phalanx with proximal contact surface highlighted in red. From left to right, contact surfaces determined using Marai et al.'s [36] two mm threshold (insufficient coverage), a 6.5 mm threshold (excessive coverage) and a 3.25 mm threshold (conservative coverage).

The extension of Marai et al.'s work via the inclusion of joint-specific distance thresholds is unique in that it allows for patient-specific customization. The spacing between two metatarsophalangeal joints may differ significantly between patient, even between the left and right foot of a single patient, and the application of a global threshold may result in excessive or insufficient cartilage coverage. Our threshold distances were heuristically determined and verified via comparison to depictions in the literature [31]. The minimum "inter-bone space" or "inter-joint distance" represents the smallest distance between the two bones comprising the joint. This distance, first used by Carrigan et al. [108] and later by Marai et al. [36] has shown to be effective in cartilage modeling. Using half of the inter-bone distance as the cartilage thickness guarantees that the pair of cartilage models will contact one another in at least one location. Joint-specific distance thresholds were found experimentally, using twice the minimum distance between vertices on polygonal models of adjacent bones, as the initial threshold. A contact surface was generated using the initial threshold and the threshold was adjusted so that the surface was visually similar to illustrations found in the literature [22].

The actual conversion from contact surface to volumetric cartilage element is relatively straightforward; the contact surface is duplicated and the copy is offset by half of the inter-bone distance described above. Each vertex is displaced along its normal by a constant distance; the vertex normal is found by computing the angle-weighted pseudo-normal [80, 110, 111]. The contact surface and offset contact surface are then splined together to create a constant thickness volumetric cartilage element. Splining takes place using a built in Rhinoceros® [112] modeling function (McNeel, 3670 Woodland Park Ave N, Seattle, WA 98103). It is worth noting that offsetting vertices by the triangle normal contained within the STL file can result in a very different outcome than if they are offset by the pseudo-normal [113]. While this method is

generally accepted, caution should be exercised when offsetting the articular surfaces; offsetting a concave surface a distance greater than its radius of curvature will result in wrinkled models that may prevent FE simulations from converging.

Athanasiou et al.'s [32] histological study of the first metatarsophalangeal joint (MTPJ) revealed a wide range of cartilage thicknesses. They presented complimentary thicknesses for the head of the first metatarsal and base of the first phalanx, with greater cartilage thickness located at the apex of the metatarsal head [11]. Cartilage thickness for the base of the first phalanx was shown to be thinnest at the center of the articulation cavity. However, for simplicity, a constant thickness cartilage was assumed.

Results

We present the first ray with algorithmically identified contact surfaces highlighted proximally in red and distally in blue (Figures 9.16a and 9.16b)[29]. Volumetric cartilage elements produced from the articular surface data are presented (Figure 9.16c); cartilage thickness was based on half of the minimum inter-bone distance. The current version of the algorithm can identify a pair of contact surfaces in less than two minutes for any joint in the first ray of the foot running on an HP® Pavilion dv6 laptop (Intel® Core™2 Duo P7450; 2.13 GHz CPU and 4GB RAM). A comparison of computational times with other methods in literature is not presented as such information was not available.

[29] Note that the boundary of the surface is "jagged" due to the nature of the algorithm; a polygon is automatically stored in the "retained" list if two or more of the vertices of that polygon lie within the specified threshold distance.

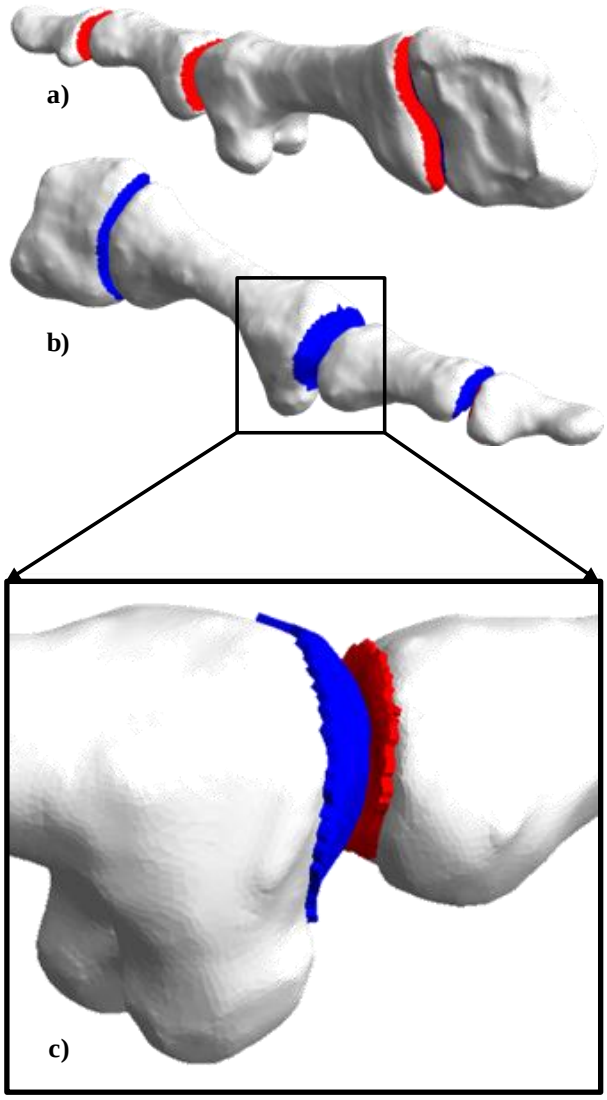


Figure 9.16. Cartilage models for the first metatarsophalangeal joint. a-b) The first ray with algorithmically identified contact surfaces highlighted proximally in red and distally in blue. c) First metatarsophalangeal joint with volumetric cartilage elements. Note that Figure 9.16c is slightly rotated from Figure 9.16b for clarity.

Distance thresholds for the first ray ranged from 3.0 to 4.25 mm depending on the joint and the desired cartilage coverage. Our algorithm identified cartilage thicknesses of 0.46, 0.72, and 0.51 mm for the first interphalangeal, metatarsophalangeal, and cuneometatarsal joints, respectively. Distance thresholds and cartilage thicknesses (half of the minimum inter-bone distance) for the first ray are presented in (Table 9.1).

Table 9.1. Distance thresholds and identified cartilage thicknesses for the first ray.

Joint	*Distance Threshold (mm)*	*Cartilage Thickness (mm)*
Cuneometatarsal (medial-first)	3.0	0.51
Metatarsophalangeal (first)	3.5 – 4.25	0.72
Interphalangeal (first: proximal-distal)	3.25 – 4.0	0.46

The cartilage thickness found using half of the minimum inter-bone distance was 0.72 mm for the distal head of the first metatarsal. This thickness is within one standard deviation of the histologically determined mean thickness presented by Athanasiou et al. (0.75 ± 0.21 mm) [32]. This result further validates the use of the inter-joint distance to predict cartilage thickness introduced by Carrigan et al. [108]. To the best of our knowledge, this is the first time this method has been successfully applied to the first ray of the foot.

Discussion

The current model extends the method presented by Marai et al. [36] to provide an algorithm that automatically identifies contact surfaces between adjacent bones; each surface is then offset, individually, by a constant distance and splined to the original to produce patient-specific volumetric cartilage elements. The current joint-specific distance thresholds go beyond what is presented by Marai et al.; we identified joint-specific distance thresholds for the first ray of the foot. Marai et al. applied a common two mm distance threshold to all of the joints in the hand. They employed the use of distance fields, whereas we generated distance maps between the two bones without computing distances to grid points. While Marai's global threshold approach may suffice for uniformly spaced joints, the joints of the foot tend to vary significantly in terms of minimum joint distance, and a global threshold was determined to be incompatible.

The use of an inadequate distance threshold, such as the two mm threshold presented by Marai, generated insufficient cartilage coverage, which resulted in cartilage edge collision during kinematic FE simulation. To prevent this collision, we implemented a conservative, that is, a slightly overcompensating distance threshold, which was shown previously in (Figure 9.4), in essence to trade anatomical accuracy for computational efficiency. This tradeoff is reflected in (Table 9.1), where the distance thresholds for the metatarsophalangeal and interphalangeal joints are given as ranges. The ranges represent minimum and maximum conservative thresholds for each of the joints; a model with a smaller threshold than the minimum will experience collision during kinematic FE simulation, whereas a model with a threshold greater than the maximum results in superfluous coverage (see Figure 9.15). The thresholds presented have not been validated against histologically determined mean thicknesses; the first ray used in this study is part of an intact cadaver foot that cannot be disarticulated due to other pending research. Future work is underway to validate the algorithmically determined distance thresholds using disarticulated first rays.

A variety of smoothing techniques including edge decimation and vertex culling could be performed on the outer boundary of the contact surfaces to decrease jaggedness and increase anatomical accuracy. However, the implementation of smoothing methods would negligibly impact the results during analysis; the purpose of the cartilage elements in these studies is to prevent bone-on-bone articulation during FE simulation. Given that the contact surface and subsequent cartilage element is an artifact of the polygonized bone model that is meshed for FE analysis, it is actually beneficial to maintain vertex coincidence between the bone and cartilage models. In doing so, it prevents superfluous gaps from forming between the bone mesh and

cartilage mesh, which leads to inaccurate results during simulation. Smoothing operations along the boundary of the contact surface would promote gap formation during meshing.

Some limitations are presented by our approach to modeling cartilage. The development of uniformly thick volumetric cartilage elements results in non-anatomical models. This is a direct consequence of the software used for manipulation of the articular contact surfaces (Rhinoceros). The volumetric cartilage elements were created by executing a built-in Rhinoceros function that displaces each vertex a constant, user-specified distance along its respective angle-weighted normal. Moreover, determination of joint-specific distance thresholds injects some bias into the models, however one can argue that any such model will have some degree of partiality imprinted on it by the user. Additionally, our approach assumes a conservative distance threshold which exposes sharp discontinuities at the edges that are not present in anatomical cartilage. These edge effects are found in other extruded cartilage models [36, 98], but their presence, in our model, does not interfere with full joint motion due to the conservative distance thresholds we applied in this study. Finally, the assumption of uniform cartilage thickness might lead to less than anatomical congruency and increased stress concentrations.

Conclusions

Developing patient-specific biological models from 3D imaging studies can provide investigators with the means to non-invasively study a number of biological and biomechanical phenomena, including joint kinematics and, if proper constitutive models are employed, joint contact stresses. We have presented a method based on extending previous methodologies in the literature to non-invasively model patient-specific volumetric cartilage elements. Our approach aims to reduce the time and effort required to generate articular cartilage models and allows for joint-specific customization. Including the user involved steps, e.g., model truncation and contact surface

offsets, we can generate volumetric cartilage elements in less than 10 minutes for any joint in the first ray using patient-specific bone models. Cartilage thickness for the first MTPJ as identified by our algorithm was in agreement to within one standard deviation of the histologically measured mean thickness in [32]. Future work is needed to further validate this technique and extend it to other joints in the foot. Moreover, the current approach is restricted to constant thickness cartilage models; successful mapping of a variable thickness function onto algorithmically identified contact surfaces will require an exhaustive study on cartilage thickness. Further studies will be necessary to validate the effectiveness of the algorithm via comparative means of surface geometry and volume. Our technique provides a computationally effective and user efficient method to generate volumetric, joint-specific articular cartilage elements that can be used in FE simulations.

9.2.4 Cartilage Modeling – Multi Position Study

Single-position CT scan data is quite useful for static modeling (Figure 9.17), however, the desire to study anatomical kinematics, contact stress and area requires full range of motion (ROM) data [36]. Acquiring full ROM CT data can be costly and timely; patients or volunteers are subjected to hours of scanning in an array of often uncomfortable positions, causing undue exposure to radiation.

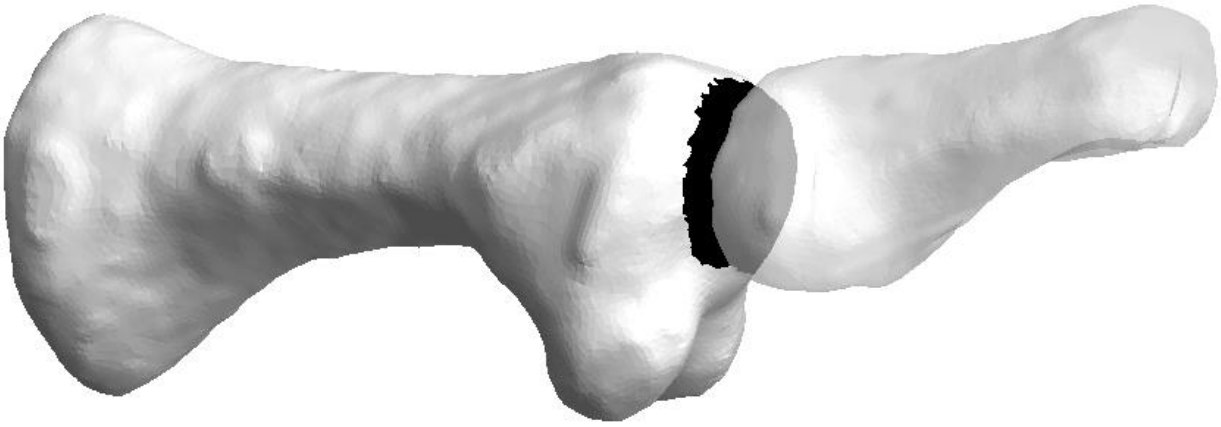


Figure 9.17: The first metatarsophalangeal joint. The models are based on single-position CT scan data with the contact surface (highlighted in black) as determined by our algorithm with a threshold value of 4.25mm.

In this initial study, we isolated the first MTPJ for ROM experimentation. It is worth noting that the first MTPJ was chosen for three main reasons: 1) the role it plays in load bearing during gait [32, 99], 2) the relatively large range of motion it exhibits during gait [114], and 3) the relative simplicity of the joint when compared to some of the highly complex joints in the midfoot and hindfoot. The first MTPJ represents an excellent site for testing the capabilities of the contact surface recognition algorithm and allows for swift model analysis via FE methods.

Shereff et al. [114] performed a kinematic study of the first MTPJ, that included an analysis of the articulation maxima. They found—for normal feet—an average total ROM of 111 degrees in the sagittal plane. Of the total 111 degrees, approximately 76 degrees was maximum dorsiflexion and approximately 34 degrees was maximum plantar flexion [114]. The articular motion for the first MTPJ was described as tangential sliding; essentially, for each position of the first proximal phalanx during articulation, there is a unique instant center of rotation that is—for normal feet—located within the first metatarsal head. Specimens with hallux rigidus revealed stochastic instant centers of rotation, with one being outside the first metatarsal head. A simplified representation

of the instant center of rotation was used in the subsequent approximation of tangential sliding for articulation of the first MTPJ.

The single-position CT scan data of the first MTPJ was converted into a multi-position, full range of motion data set by considering the pseudo-centroid for each bone. The pseudo-centroid is found by determining the average of the coordinates for each bone data set. A line segment can then be drawn through both the first metatarsal's and proximal phalanx's pseudo-centroid. The midpoint of this line segment closely approximated—visually—the average of the instant centers of rotation depicted in [114]. By rotating the first proximal phalanx about this point, we were able to emulate tangential sliding for the first MTPJ. Initially, rotation angles of 76 and 34 degrees were used for maximum dorsiflexion and maximum planter flexion, respectively (Figure 9.18) [114]. It is worth noting that by visually approximating the average instant centers of rotation, error may have been introduced during rotation of the first proximal phalanx. Due to the nature of this study, the only foreseeable consequence of this error was the necessity to either increase or decrease the distance threshold during contact surface identification.

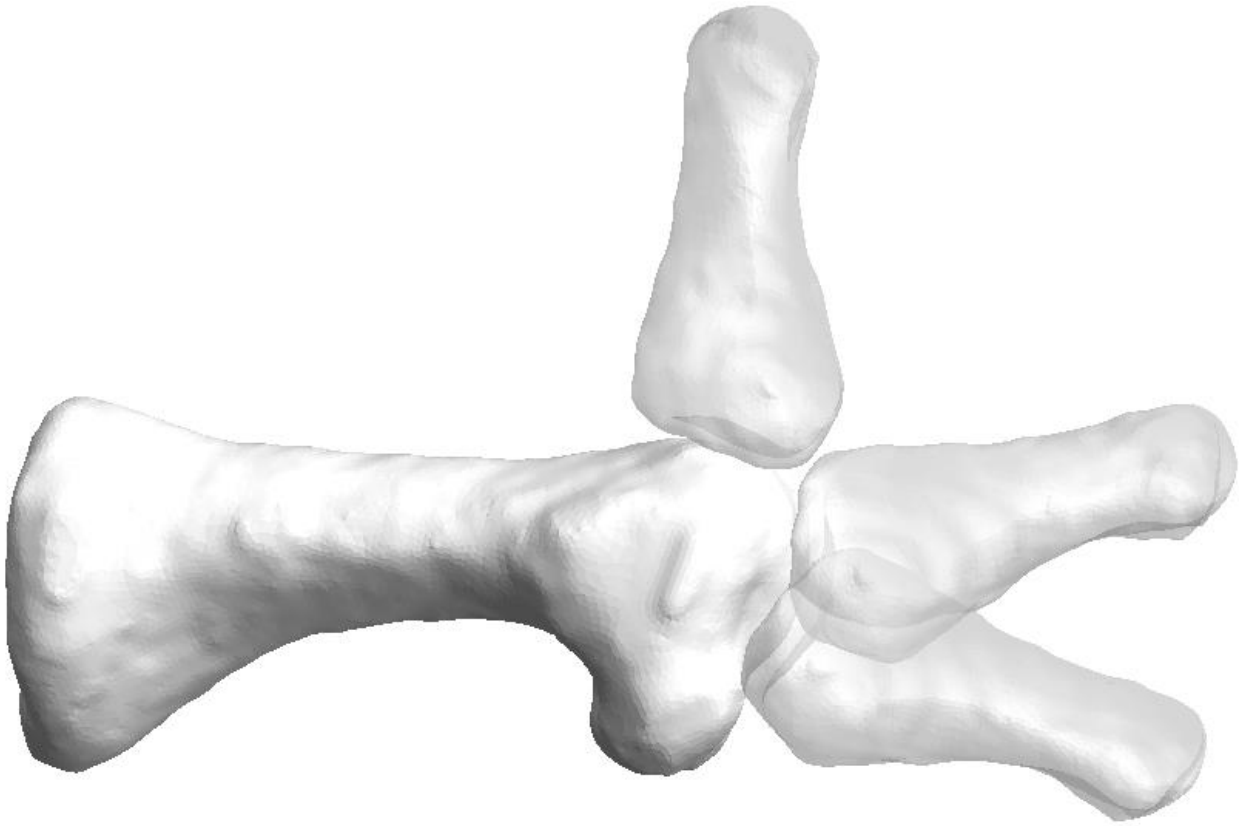


Figure 9.18: The first MTPJ with the proximal phalanx in three analyzed configurations. 1) Maximum plantar flexion (bottom), 2) neutral (middle), and 3) maximum dorsiflexion (top).

Figure 9.19 reveals the results of running our algorithm with the first proximal phalanx located in three distinct joint positions (neutral, maximum dorsiflexion, and maximum plantar flexion). The arrows ‘A’ and ‘B’ in Figure 9.19 indicate points of irregular sharpness on the overall contact surface. These sharp corners were the consequence of a union operation on the three contact surfaces. If left unchanged, the abrupt boundaries of the volumetric cartilage elements may impede articulation motion during simulation via cartilage-to-edge interference.

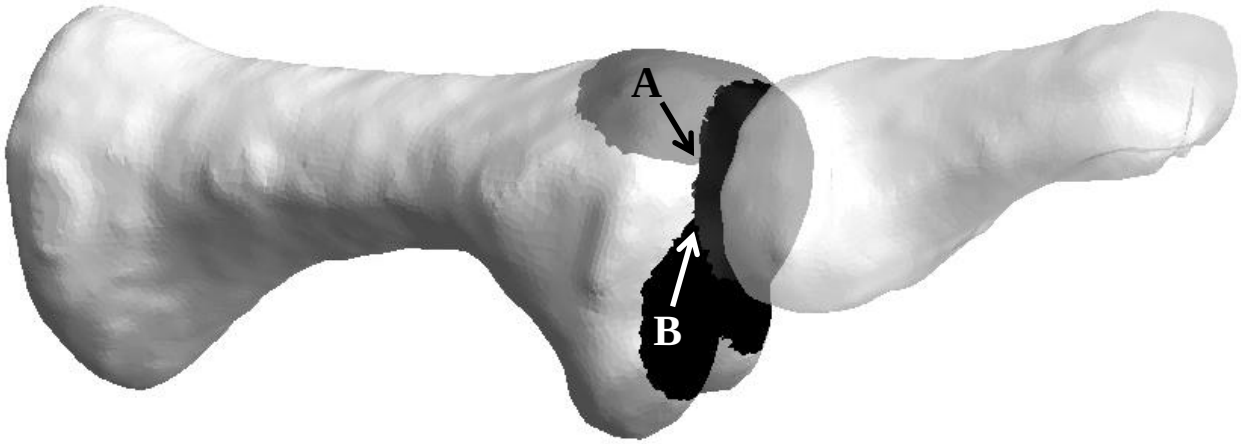


Figure 9.19. Results for the kinematic study of the first MTPJ. Contact surfaces are highlighted in black (maximum plantar flexion), dark gray (neutral), and light gray (maximum dorsiflexion).

To create a smoother and more anatomically reasonable full ROM contact surface for the first metatarsal head, we further discretized the range from dorsiflexion to plantar flexion. The need for a more anatomical model (i.e., a smooth, continuous patch of cartilage) was driven by the desire to eliminate bone-on-bone contact during articulation. This led to the addition of four proximal phalanx rotations at angles of: 19, 38, and 57 degrees for dorsiflexion, and 17 degrees for plantar flexion. The results of this discretization are shown in (Figure 9.20).

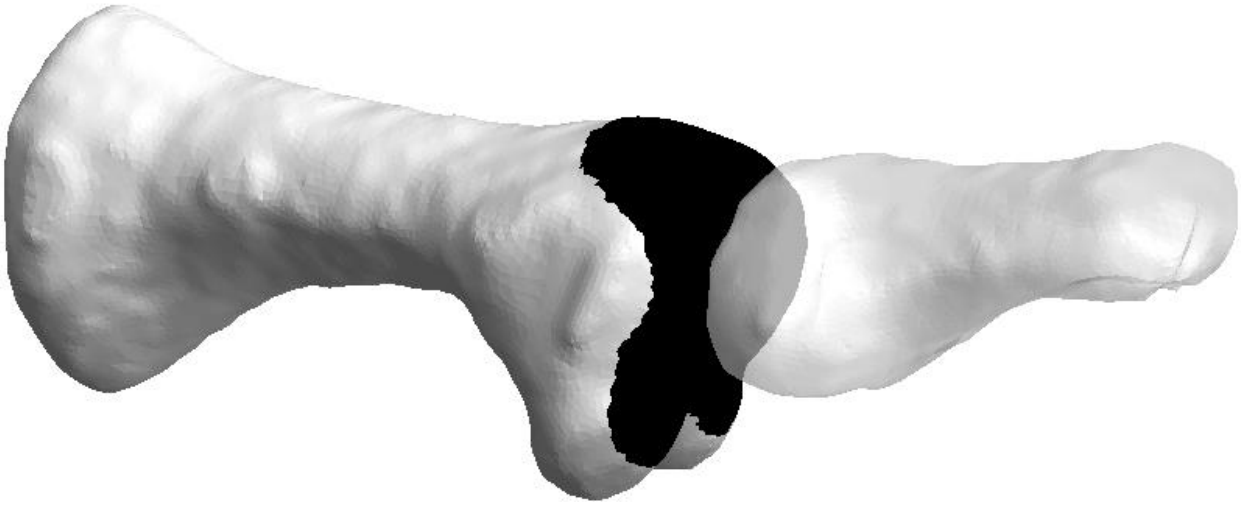


Figure 9.20. Results of the discretization study for the first MTPJ. Note the smoother transition from maximum dorsiflexion to plantar flexion when compared to Figure 9.19. A total of seven contact surfaces were combined using a union operation to create the final surface shown here.

Summary of Findings

A number of joints in the human foot can demonstrate large ranges of motion during gait, specifically the first MTPJ. As a direct consequence, single-position CT scans of the foot do not provide adequate data for complete contact surface identification and subsequent volumetric cartilage generation using our algorithm. We manipulated the first MTPJ data to emulate tangential sliding [114], a complicated motion exhibited by the first MTPJ during gait. Maximum dorsiflexion (76 degrees from neutral) and plantar flexion (34 degrees from neutral) [114], and four intermediate angles (19, 38, 57, and 17 degrees for dorsiflexion and plantar flexion, respectively) were employed in our algorithm. The outcome suggests that, given information about joint kinematics, multi-positional CT scans are not necessary for computing full ROM volumetric cartilage elements for the first MTPJ.

The cartilage thickness determined by our algorithm was 0.72 mm for the first MTPJ, which is within 0.03 mm (and within the standard deviation) of the thickness found histologically by Athanasiou et al. (0.75 ± 0.21 mm) [32]. A limitation to our method is the creation of uniformly

thick volumetric cartilage elements. This is a direct consequence of the software used for manipulation of the articular contact surfaces; Rhinoceros [112], a 3D modeling program, was used for contact surface manipulation. The volumetric cartilage elements were created by utilizing a built-in Rhinoceros® function that displaces each vertex a constant, user-specified distance. We are investigating methods to generate variable thickness cartilage elements.

Sharp discontinuities are exposed at the edges of the volumetric cartilage elements that are not present in anatomical hyaline cartilage. These edge effects are displayed in other extruded cartilage models [36, 98], but their presence does not interfere with the joint's kinematics during articulation. By choosing a conservative distance threshold, we produce contact surfaces that provide increased coverage on the bone. The slight overcompensation of cartilage coverage reduces the risk of potential edge effect interference during FE joint articulations, which could disrupt the computationally expensive simulations.

9.2.5 GPU-Based Cartilage Modeling

This section was adapted from a three month course designed to convert the CPU-based code discussed in the previous section into GPU-based code. As with the previous chapter, presentation style waivers from the bulk formatting of this dissertation.

To minimize the bias and time needed to generate cartilage models, our group implemented a serial algorithm similar to those described in the preceding sections, but written in Mathematica, that identified potential cartilage regions using inter-vertex distance. Vertices on one bone that were sufficiently close to the other bone, e.g., within a specified threshold, were stored and all polygons sharing a stored vertex were categorized as cartilage. This approach presented a computational complexity of $O(NM)$, where N and M represented the number of vertices in a pair of polygonized bone models. The large number of computations resulted in slow execution

times. To minimize computation time, bone models were manually truncated leading to a reduction in the total number of vertices. This approach enabled the generation of cartilage models in approximately 5-10 minutes (including pre/post-processing). However, identification of an appropriate distance threshold generally required several iterations, thus reducing computational efficiency. To address these issues, our approach will extend the work of Marai et al. through the implementation of distance fields computed on the GPU and displayed, in real-time, using a combination of OpenGL and CUDA.

Project objectives

The intended deliverable for the course was to create a user interface that, in real-time:

1. Rendered both bones,
2. Rendered both articular surfaces,
3. Provided independent kinematic control for each bone
4. Output a collection of triangles corresponding to the identified surfaces

Methods

The initial implementation of the articular surface identification algorithm was built upon the existing Mathematica (MMA) code briefly described above. However, we faced several challenges that prevented the final implementation from being executed using MMA and CUDALink. Each of the individual methods will be briefly discussed in the subsequent sections.

Method 0: Mathematica, CUDALink and vertex to vertex distance

The general idea was to convert existing MMA code that ran serially on the CPU into MMA code that ran in parallel on the GPU using MMA's CUDALink. CUDALink provides a bridge to parallel computing that enables the programmer to take advantage of MMA's many built-in

visualization and manipulation tools. In essence, this allows the programmer to interact with their results without having to write code in OpenGL, an API for interactive graphics that can be used in conjunction with CUDA.

To determine the inter-vertex distances we implemented a simple Euclidean distance from point to point in 3-space algorithm. Polygonized bone models were read into MMA and each bone's vertex list was stored in an array. The algorithm took as arguments the two vertex lists, a distance threshold and an output array that was initialized to zero prior to execution. Computational threads were launched for each vertex on each bone and the algorithm stored vertices that met the threshold criteria.

Visualization of the results was performed using built-in MMA functions that enable 3D interaction of polygonized models, in addition to a 3D point plot for viewing the vertices that were within the threshold. A manipulate function allows the user to vary plot parameters via an interface tool such as a slider and this function was used to adjust the distance threshold. As the threshold changed, vertices either appeared or disappeared on the bone of interest (see Results section).

Method 1: Visual Studio, CUDA, OpenGL and unsigned distance fields (UDFs)

The second method attempted was similar to *Method 0* in that existing MMA code was to be converted into code that could be executed on the GPU. However, as opposed to using MMA and CUDALink, the algorithm was written in Visual Studio which also has a CUDA plug-in. In this method, we implement a brute-force approach to computing unsigned distance fields. The two polygonized bones comprising a joint are read into the program, which then stores the vertices of each triangle in global arrays. We determine the bounding box of the two bones using the minimum and maximum coordinates of the vertices. By doing this, we ensure that each

bone's distance field is sampled on the same grid, which allows for simple visualization and computations in subsequent operations.

We launch a computational thread for each point on the grid and determine the closest point on the bone through primitive comparison. For each primitive on each triangle, e.g., 3 vertices, 3 edges, 1 face, we construct the implicit representations of the defining boundaries, determine where the grid point lies relative to the boundaries and use the appropriate distance formula to compute the distance to the surface of the bone. If the distance to the current triangle is less than the distance to the previous triangle, we update the distance. This protocol repeats until we have computed distances to all triangles in the models. The end result is a pair of UDFs for the two bones. Prior to outputting the data, we compute the union of the two fields by taking the minimum value at a grid point. This combined field is used in a later operation when isolating articular contact regions.

To visualize the bones and show the articular surfaces we modified the volume rendering example in the CUDA 5.0 SDK. The three unsigned distance fields, one for each bone and one for the union of the two fields, are stored in 3D textures. The example renders a volume by creating rays that originate at a focal point and end at a pixel. The algorithm samples the texture at points along the ray and if the point lies within the volume, the texture returns a value and a sum is updated. The resulting sum is rendered on screen at the appropriate pixel. Given that each ray is independent, computational threads can be launched for each pixel, making for an efficient rendering algorithm. Visualization of the UDFs does not require the algorithm to step through the volume because we are only interested in viewing the zero level set that approximates the bone's surface. As such, we evaluate the texture at a point on the ray and if the returned value is within some specified distance from the zero level set, we set the pixel value by compute the

normal at that point by approximating the gradient and taking the inner product of the normal with the ray. The product of this computation can be seen in the Results section.

Once we have rendered the bones, we also need to render the articular surfaces using a similar approach. However, rendering the articular surfaces requires that we employ the third texture that is the union of the two fields. We make use of the third texture by subtracting its sampled value from the active bone's sampled value. If the absolute value of this difference is less than some threshold, we render the surface as above but with two of the color channels turned off to distinguish bone from cartilage. Interactive controls from the SDK example were modified such that keystrokes increase or decrease the threshold.

Results

Method 0: Mathematica, CUDALink and vertex to vertex distance

Using *Method 0* we were able to identify vertices on one bone that were within the distance threshold in approximately 0.5 seconds. However, the data is output as an $N \times M$ matrix where N and M represent the number of vertices comprising each bone, respectively. The processing that was required to then view the saved vertices is computationally inefficient, and for a pair of bones with 4935 and 8036 vertices, respectively, processing takes approximately 55 seconds. Once processing is complete, a simple visualization function is executed that allows for interaction with the model (i.e., 3D rigid rotations). A representation of the bones with the potential articular surface vertices highlighted is shown in Figure 9.21a-b.

While using this method does yield interactive models that show both the bones and contact surfaces, it does not allow for fast interaction between model and distance threshold. Visualization of an updated distance threshold takes approximately one minute; given that

several iterations are generally needed to identify a single articular surface, the algorithm is ultimately time prohibitive.

a) b)

Figure 9.21. Articular surface maps. Method 0: a) Shows the first interphalangeal joint with articular surface vertices highlighted in red on the distal head of the first proximal phalanx. Note that the distance threshold was too large in this case resulting in excessive cartilage coverage. b) Shows the same joint with a smaller distance threshold and appropriate cartilage coverage.

Method 1: Visual Studio, CUDA, OpenGL and unsigned distance fields (UDFs)

This approach requires that we first generate unsigned distance fields for each of the two bones of interest. The algorithm is fairly efficient, generating UDFs for a 16,000-polygon bone on a 64^3 grid in ~6 seconds and on a 128^3 grid in ~30 seconds. However, once the UDFs have been stored in 3D textures, real-time visualization and interaction with both the bones and the articular surfaces is possible through the integration of CUDA with OpenGL. The renderer uses the distance threshold, which is controlled by a keystroke, to highlight the potential articular surface in real-time. Figures 9.22a-d show various screen captures of the algorithm when different distance thresholds were specified.

After identifying on-screen the region that is perceived to be most appropriate for cartilage, we must read the distance threshold from the command window. The distance threshold is then used

in the algorithm described in *Method 0*, to output the collection of polygons corresponding to the region identified using the real-time interface. As in *Method 0*, this step requires approximately one minute as complete. Nonetheless we are able to greatly reduce the time needed to identify articular surfaces by taking advantage of the real-time user interface, which enables instant visual determination of a single threshold rather than the more time consuming iterative determination used in *Method 0*.

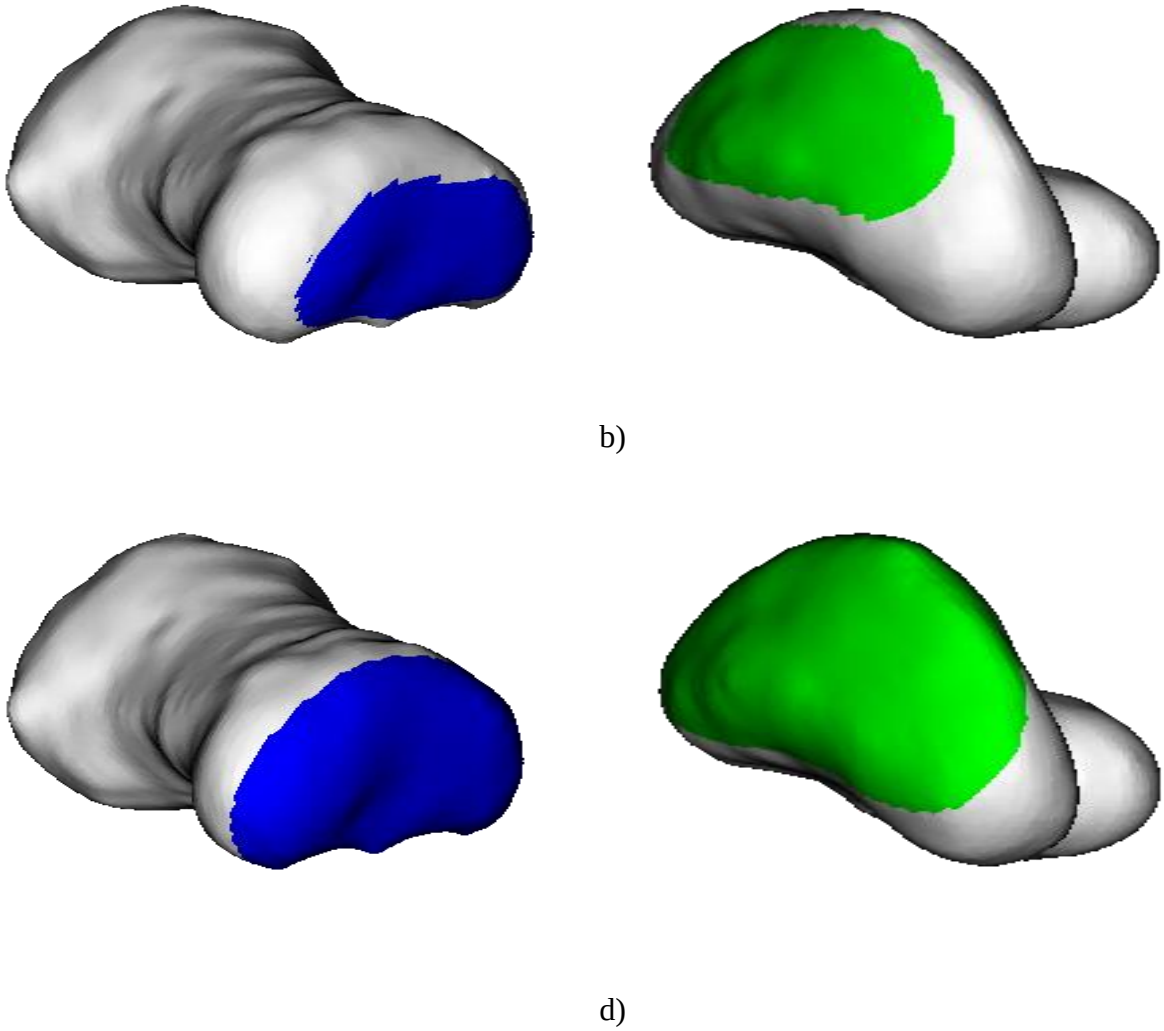
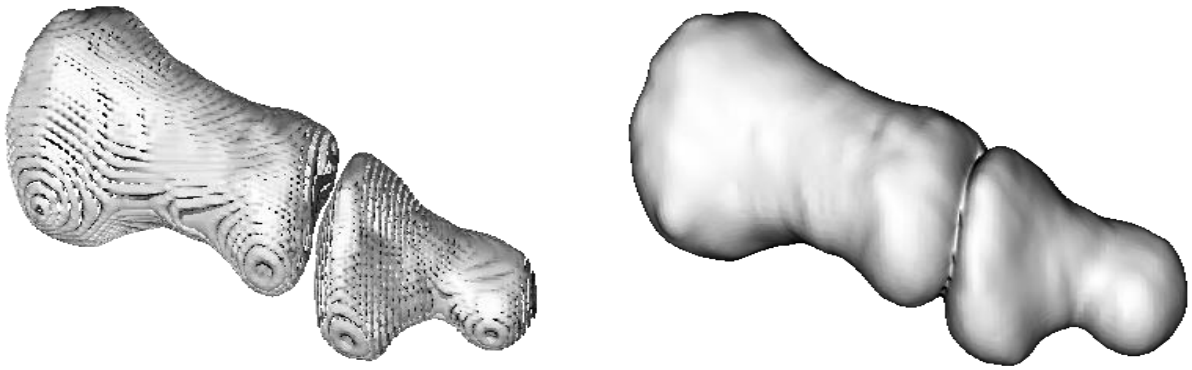


Figure 9.22. Articular surfaces found using a parallelized distance field algorithm. Method 1: a-b) Shows the first proximal and distal phalanxes with articular surfaces highlighted in blue and green, respectively. Note that the distance threshold was too small in this case resulting in insufficient cartilage coverage. c-d) Shows the same bones with a larger distance thresholds and appropriate cartilage coverage.

One problem we encountered with the level set viewing method was field resolution and field blending. A consequence of low grid resolution was renderings with visible artifacts at level sets near the true surface. As we approached the zero level set, the artifacts became more pronounced and the model was more difficult to view (Figure 9.23a). To ease viewing and minimize the appearance of artifacts, we moved further away from the surface (i.e., viewed larger level sets), but this resulted in blending of the two fields (Figure 9.23b). The blending became visible when each bone's level set was equal to or greater than half of the minimum articular gap distance, preventing simultaneous rendering of bone and articular surface.



a) b)

Figure 9.23. Various level sets of the first metatarsophalangeal joint. a) Shows the near-zero level set with artifacts present due to the grid spacing being too large to capture all detail on the bone. b) Shows an offset surface that exceeds half of the minimum articular gap resulting in field blending at the gap.

Conclusions

We present two approaches that minimize subjectivity when modeling articular surfaces using the GPU as the computational engine. Both methods provide a significant decrease in the time needed to identify articular surfaces compared to manual segmentation techniques previously used by our group, and offer a means to quickly visualize and manipulate the results. We show that existing serial Mathematica code used to identify vertices belonging to articular surfaces for

a given distance threshold in approximately 2 minutes can be streamlined using the CUDALink tools to facilitate GPU computing. By employing CUDALink, we were able to identify vertices belonging to an articular surface, for a single distance threshold, in approximately 0.5 seconds, which is 240x faster than the serial implementation.

Visualization of the findings proved to be the only negative ramification of utilizing MMA's CUDALink. Post-processing of the vertices took approximately 55 seconds and the iterative approach to finding the optimal distance threshold meant that identifying a single articular surface could take up to 30 minutes, making this method similar in overall runtime to the serial implementation. The second method converted existing MMA code for signed distance fields into VS/CUDA code for unsigned distance fields. UDFs for a pair of bones were computed in ~12 seconds (total) on a 64^3 grid and ~50 seconds (total) on a 128^3 grid, and a union operation was performed to create a third field. The three distance fields were stored in 3D textures and rendered in real-time using a volume rendering turned ray casting algorithm from the CUDA 5.0 SDK. A keystroke-modifiable distance threshold enabled real-time adjustment of articular surfaces. Once an articular surface was considered visually acceptable, the distance threshold was used in a MMA algorithm to export the polygonized version of the surface (~1 minute). Given that the only iterative operation for this method need happen during a real-time rendering step, the overall runtime to identify articular surfaces is greatly reduced over the MMA/CUDALink implementation.

We have shown that a pair of polygonized models can be rendered and articular surfaces identified in real-time using a combination of Visual Studio, CUDA, OpenGL and distance fields.

9.2.5 Full Foot Contact Surface Identification and Cartilage Modeling

Tables 9.2-9.4 provide the heuristically found distance thresholds, in addition to the cartilage thicknesses as determined by the algorithm, for most joints in the foot. As described above, the cartilage thickness is found by computing the minimum inter-joint spacing and then halving it. This provides constant thickness cartilage elements, and at minimum a single point of contact between adjacent cartilage elements within a joint [36, 108]. While the below tables do specify baselines for distance thresholds and cartilage thicknesses, it is worth noting that additional optimization should be performed on a joint-specific basis; some of the contact surfaces may have excessive or insufficient contact surface coverage due to current computational restrictions (refer back to Figure 9.18 for an examples of insufficient and excessive coverage). The values presented were based on a single patient's foot model; they are not intended to be absolute.

Table 9.2: Distance thresholds and cartilage thicknesses for the joints of the hindfoot and ankle.

Ankle/Hindfoot		
Joint	**Distance Threshold (mm)**	**Cartilage Thickness (mm)**
Tibiofibular	3.0-4.0	0.97
Tibiotalar	4.5	1.24
Fibulotalar	6.0	1.02
Talocalcaneal	6.0	0.72
Talonavicular	3.25 - 4.0	0.47
Calcaneocuboid	4.0	0.60

Table 9.3: Distance thresholds and cartilage thicknesses for the joints of the midfoot.

Midfoot		
Joint	**Distance Threshold (mm)**	**Cartilage Thickness (mm)**
Naviculocuboid	4.0	0.75
Naviculocuneiform (medial)	3.5	0.62
Naviculocuneiform (intermediate)	3.5	0.54
Naviculocuneiform (lateral)	2.0	0.52
Intercuneiform (medial-intermediate)	2.75-3.25	0.52
Intercuneiform (intermediate-lateral)	2.75	0.53
Cuneocuboid	3.5	0.48
Cuneometatarsal (medial-first)	3.0	0.51
Cuneometatarsal (intermediate-second)	2.5	0.47
Cuneometatarsal (lateral-third)	2.15 - 2.5	0.52
Cubometatarsal (fourth)	2.5	0.57
Cubometatarsal (fifth)	2.5 - 2.75	0.50

Table 9.4: Distance thresholds and cartilage thicknesses for the joints of the forefoot.

Forefoot		
Joint	**Distance Threshold (mm)**	**Cartilage Thickness (mm)**
Metatarsophalangeal (first)	3.5 - 4.25	0.72
Metatarsophalangeal (second)	3.25 - 4.0	0.95
Metatarsophalangeal (third)	3.0 - 4.0	0.62
Metatarsophalangeal (fourth)	3.0 - 4.0	0.67
Metatarsophalangeal (fifth)	3.75 - 4.5	0.72
Interphalangeal (first: proximal-distal)	3.25 - 4.0	0.46
Interphalangeal (second: proximal-int.)	3.25 - 4.25	0.61
Interphalangeal (third: proximal-int.)	3.75 - 4.25	0.63
Interphalangeal (fourth: proximal-int.)	2.25 - 3.5	0.51
Interphalangeal (fifth: proximal-int.)	---	3.08
Interphalangeal (second: int.-distal)	2.0 - 4.25	0.54
Interphalangeal (third: int.-distal)	4.0 - 4.5	0.80
Interphalangeal (fourth: int.-distal)	3.0 - 4.75	0.99
Intermetatarsal (first-second)	3.0 - 3.5	0.77
Intermetatarsal (second-third)	3.0	0.56
Intermetatarsal (third-fourth)	3.0	0.53
Intermetatarsal (fourth-fifth)	3.0	0.50

Also shown in Figure 9.24, are the results of reconstructing manually segmented cartilage mappings based on MRI data of the physical talus being modeled. As discussed, manual

segmentation is extremely user-biased, inefficient and monotonous. However, it does provide an excellent reference for comparing cartilage coverage generated by our algorithm.

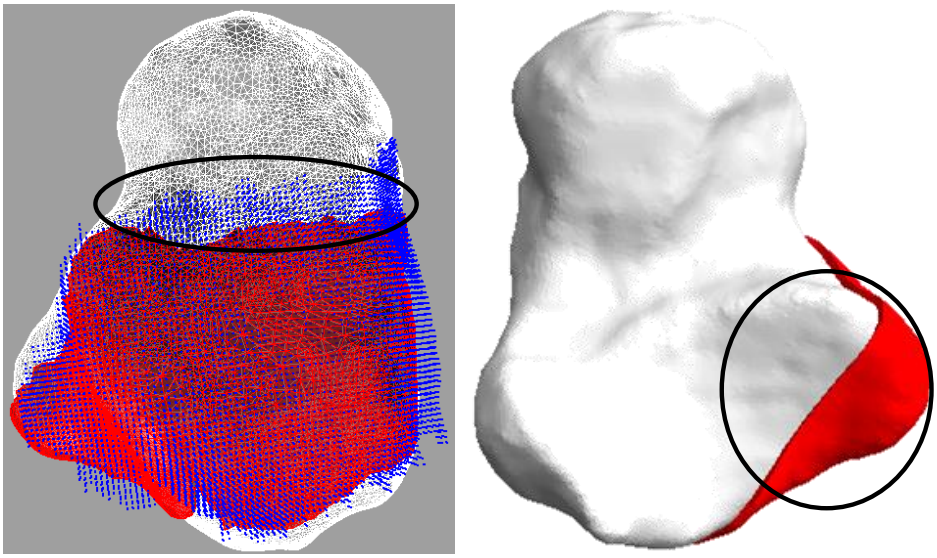


Figure 9.24. A superior view (left) of the talus (white), algorithmically generated volumetric cartilage (red), and a reconstructed cartilage map produced by manual segmentation of MRI slices (blue). Note the lack of algorithmically derived cartilage coverage (circled), which is related to the distance threshold and in this case kinematics. A posterior view (right) of the talus (white) and algorithmically generated cartilage (red). Note the excessive coverage of algorithmically derived cartilage (circled); tibiofibular cartilage is not present on the talocalcaneal articulation surface. This is a direct consequence of an excessive distance threshold.

The minimum distance threshold was 2.0 mm for the naviculocuneiform (lateral) joint. The maximum distance threshold was 6.0 mm for the fibulotalar and talocalcaneal joints. The minimum inter-bone half distances ranged from 0.93 to 2.49 mm for the first interphalangeal and tibiotalar joints, respectively.

Many of the joints in the midfoot and hindfoot are prime examples of the algorithm's computational limitation when executed on the CPU. The large data sets inhibit heuristic analysis of the joint distance threshold; a GPU implementation will eliminate this issue, indicating the need to further develop the technology.

9.2.6 Joint Capsule Modeling

The FE model of the foot developed by our group is comprised of bones, fat, cartilage, ligaments, tendons, plantar aponeurosis, and generic soft tissue. The investigation of joint capsules for use in the FE foot model was motivated by soft tissue interference that hindered physiological joint motion during kinematic simulation. Joint capsules were created to produce a "void" region (Figure 9.25) in the model where elements could be eliminated allowing for uninhibited joint articulation. Previous attempts at creating void spaces in the FE model required the researcher to manually remove elements in the neighborhood of the joint.

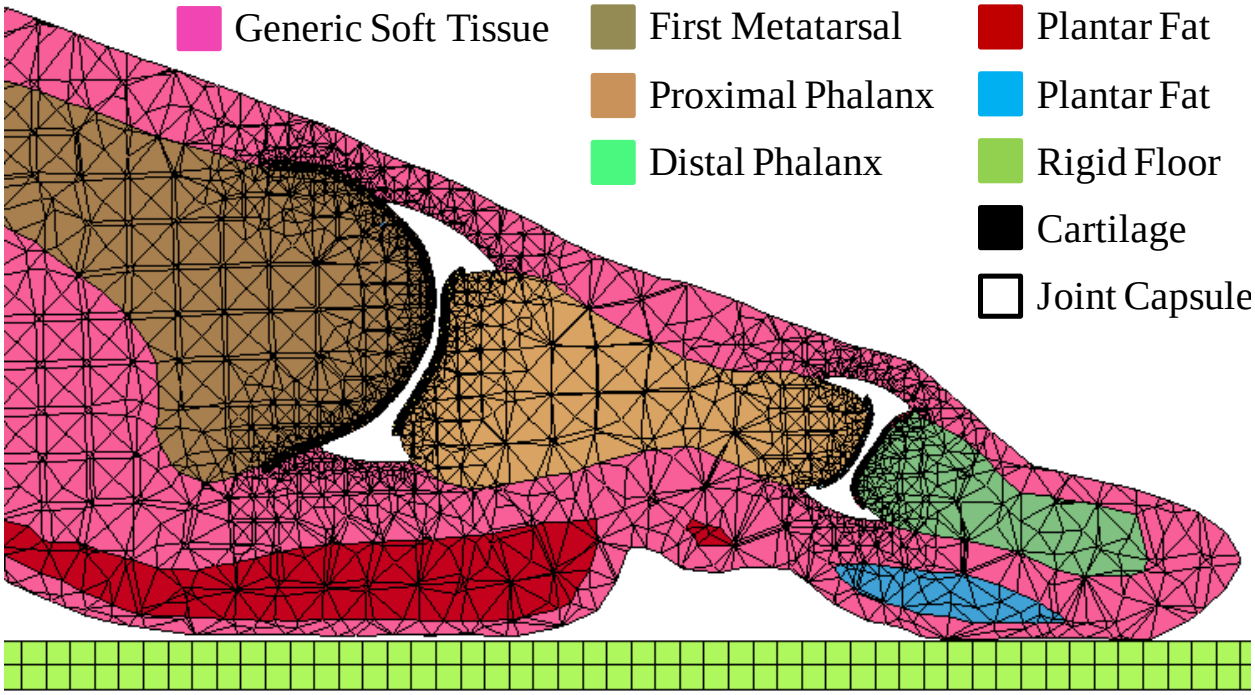


Figure 9.25. Sagittal view of the FE model of the first ray. Note the void regions at the joint locations (joint capsules) which prevents soft tissue interference during FE kinematic simulation. Printed with permission from a research colleague, Vara Isvilanonda.

We have completed joint-specific modeling of two articular capsules in the first ray, namely, the MTPJ and interphalangeal joints (IPJ). The joint capsules were generated heuristically in Rhinoceros® by strategically placing ellipsoids in the vicinity of the joint centers. The ellipsoids were then transformed into joint capsules by Boolean subtracting the pair of bone models,

including the volumetric cartilage elements, from the ellipsoid. The first MTPJ and IPJ articular capsules are illustrated in 3D in Figure 9.26.

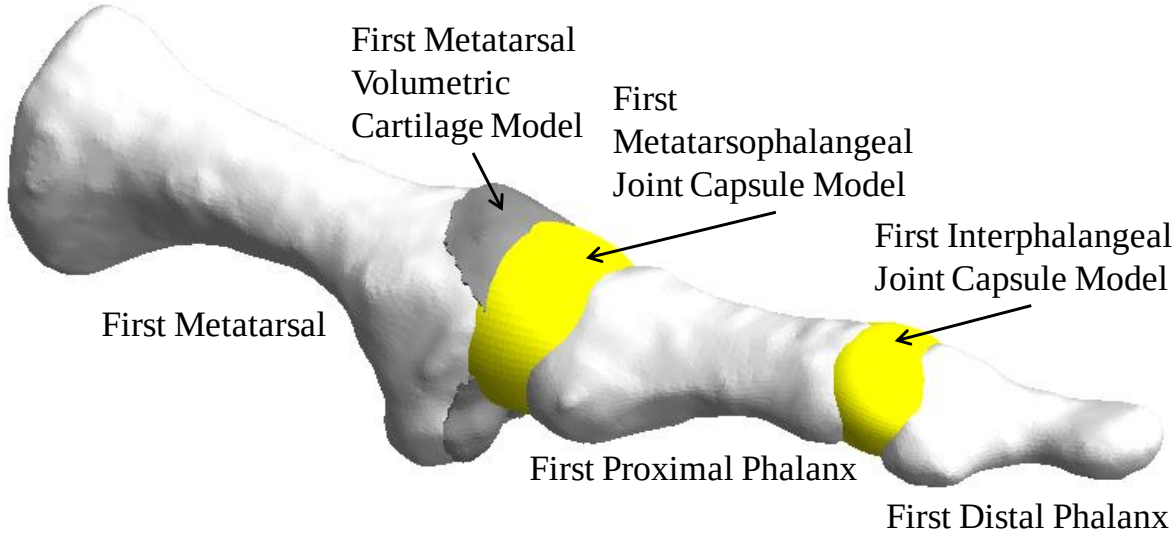


Figure 9.26. A 3D model of the first ray that includes algorithmically generated volumetric cartilage elements and systematically determined joint capsule models.

The approach that we have employed in the creation of these models is extremely user biased and highly user involved. As such, it would be optimal to automate this process in an effort to increase efficiency, model accuracy, and repeatability. The implementation of geometric skeletal models, a topic that we will introduce in the following section, has the potential to streamline patient- and joint-specific articular capsule modeling.

Chapter 10: Results, Conclusions and Recommended Works

The final chapter will be used to recapitulate the results presented in this dissertation and will provide a survey of future recommended works that are a direct consequence of this research. In Chapter 1, we introduced the underlying working hypothesis for this dissertation: *Can real-world computational problems in biomechanics – including soft tissue modeling and, in particular, 2D-3D markerless registration – be solved using a parallelized computational mindset to realize efficiency gains of several orders of magnitude over sequential processing without sacrificing the requisite sub-millimeter and sub-degree precision?* The short answer is YES!

Chapter 4: DRRACC and DRRACO were at the core of the work presented in this dissertation. We discussed in great detail the development of and methods used by the image registration toolkit that was the culmination of several years of dissertation research: DRRACO. DRRACO is a distilled version of DRRACC and was developed as a tool to radically accelerate the computationally intense challenge associated with 2D-3D medical image registration using a biplane fluoroscopy system as the 2D imaging modality and CT scan data as the 3D imaging modality. By far the most compelling feature of DRRACO is the interface that was put in place to insulate the user from the specifics of parallel processing on the GPU via CUDA. Another important feature that came out of the effort to restructure DRRACC into DRRACO was the development of completely independent GPU-based toolkit functions. This consequence of the DRRACC overhaul provides the user with more flexibility to target their specific computing needs, by promoting the evolution of custom-tailored toolkit process flow. Shielding the user from GPU-programming specifics, in conjunction with a malleable toolkit should help to expand the pool of investigators interested in parallel processing with ready access to GPU-computing capabilities, which in turn will lead to new scientific discoveries.

The list below summarizes the notable contributions of the DRRACO 2D-3D registration toolkit:

1. Complete separation at the interface level of CPU- and GPU-based programming
2. GPU-implementations of image processing tools
    a. Masking
    b. Cropping
    c. Dilation
    d. Erosion
    e. Convolution
    f. Finite difference gradient approximation
    g. Composition
    h. Summation
    i. NCC calculation
    j. DRR generation
3. Implementation of a real-time GUI using CUDA-OpenGL Interoperability Library
    a. Streaming quantitative feedback in the form of NCC
    b. Real-time manipulation and visualization of DRRs using intuitive commands
4. Integration of two optimization suites
    a. SNLP
    b. CONDOR

5. Consolidation of contributions 1 through 4 into an intuitive, stand-alone software package
6. Application of DRRACC/DRRACO to a real-world application
    a. Sub-millimeter and sub-degree precision achieved using GPU-based markerless registration
    b. Efficiency gains of 10x to 150x over typical CPU implementations

Chapter 8: Validation of DRRACC was at the pinnacle of this research, which manifested in the form of validating the DRRACC toolkit. Several methods were presented and implemented to establish the correctness, precision and computational performance of DRRACC:

1. Image formation
2. Pixel-by-pixel intensity comparison
3. Image linearity
4. Image geometry
5. Registration application routine to real imaging data:
    a. Pure translation
    b. Pure rotation
6. Computational performance

DRRACC passed all validation tests with results that closely matched or, in some cases, exceeded the CPU-based benchmarks in terms of precision. During stage translation validation, DRRACC achieved sub-millimeter precision on the order of the precision exhibited by the CPU benchmark. For stage rotation validation, DRRACC realized sub-degree precision that outperformed the CPU benchmark. In both cases, DRRACC was shown to speed up the

registration process by up to an exceptional 150x over the CPU benchmark (via MATLAB implementation of CPUDRR). Execution times for DRRACC were measured in **minutes**, while those for the corresponding CPU benchmark were measured in **hours**.

The implications of this massive boost in computational performance are significant, especially when considering the bigger picture: Quantification of 3D foot bone motion is not a problem that can be achieved by simply analyzing a single subject's gait record. Advancing our knowledge of foot bone kinematics may eventually lead to scientific breakthroughs and innovations in podiatry and related academic subjects; however, deepening our understanding of the problem domain will demand an expansive study that enlists a large pool of subjects exhibiting both normal and abnormal foot conditions. Attempting to tackle even a handful of subjects using the CPU benchmark code is computationally prohibitive. DRRACC has proven that it can find the balance that exists between highly precise, data-intense processing and computationally efficiency, by recruiting the GPU to execute commands in parallel.

In *Chapter 9*: *Parallelized Distance-Based Tools*, we introduced the concept of point-to-triangle distance and described how it could be mapped onto a collection of polygons to compute discrete distance fields of triangulated data. This concept was then extended to cartilage modeling by using finite distance fields in conjunction with a distance threshold to identify the vertices in a polygonized model that were good candidates for articulation contact surfaces. The findings presented in this chapter concluded the preliminary investigation of a distance-based algorithm that is capable of identifying a pair of adjacent articulation surfaces in <2 minutes for any joint in the foot. The previous approach that required a skilled investigator manually identify each of hundreds of pixels corresponding to a single slice of cartilage for a single joint for over a

hundred noisy CT slices was quite inefficient and warranted the development of the GPU-based cartilage modeling toolkit.

In the final months leading up to the transition away from distance-based tools towards image processing tools, the distance field strategy of mapping the point-to-triangle distance function onto a collection of polygons was implemented on the GPU. The parallel distance field algorithm is capable of generating unsigned distance fields for a 16,000-polygon bone model on a 64^3 grid in ~6 seconds and on a 128^3 grid in ~30 seconds. Once generated, the distance field can be utilized by the GPU-based cartilage modeling toolkit to achieve real-time modeling and manipulation of cartilage maps. This toolkit has the potential to positively impact the field of patient-specific biomechanical modeling.

Recommended Future Works

1. Code efficiency:

    a. Card-based optimization – The DRRACC toolkit was developed over the course of several years during a time when improvements to GPU hardware was blossoming. As such, we iterated through a handful of consumer GPUs as the CUDA hardware and software models improved. It was not until the final stages of this dissertation when we finally upgraded to a GPU that was specifically designed for massive parallelization (Tesla K20). Many of the kernel functions could benefit from a serious investigation of kernel parameters to identify an optimal configuration that enhances computation efficiency.

    b. GPU profiling – At the end of Chapter 8, we provided a cursory look at GPU analytics with Visual Profiler. The goal of this initial investigation was to determine the feasibility of profiling DRRACC; a more involved analysis should

be conducted in an effort to further improve the computational efficiency of DRRACC/DRRACO.

2. Gradient strategies:

    a. Combinations of components and magnitude – During the stage translation and stage rotation validation studies discussed in Chapter 8, we discovered that the form of the gradient used in subsequent NCC computations was found to affect registration quality. Optimal formulation of the gradient contribution to the objective function remains an open question worthy of further investigation.

3. Optimizers:

    a. Parameter sensitivity – Comprehensive analysis of the effects of the various optimization parameters available in both SNLP and CONDOR remains to be investigated especially in regards to reliability of registration of data from live subject trials.

    b. Alternative optimization suites – While SNLP and CONDOR performed adequately for the purposes of this dissertation, some resistance was encountered during integration with the DRRACC toolkit. One of the negative ramifications associated with the CONDOR suite is that it is no longer supported; this made debugging particularly challenging. Moreover, ongoing challenges in reliable registration of data from live subject trials support consideration of alternate optimization codes.

4. Multibone optimization:

a. Optimization strategy – The strategy presented in Chapter 8 for multibone optimization relied on registering all of the active bones simultaneously. We found that this strategy resulted in optimization times that scaled linearly with the number of bones for small numbers of bones (<=4). The scaling of registration times for larger numbers of bones remains an open question at this time. An alternative multibone strategy could be implemented that performs a pair of sequential single bone optimizations.

    i. Stage 0 – The user would manually register the bones with the help of the positioning GUI or use a starting configuration based on results from neighboring frames.

    ii. Stage 1 – The optimizer would be invoked sequentially for each active bone. This initial optimizer run would serve as a refinement step to increase the accuracy of the manual alignment.

    iii. Stage 2 – A second sequential optimization run would be used to improve the accuracy of the first run. If the registration quality met a user-defined threshold, the program would exit. In the event that a second run does not meet the minimum registration quality threshold, subsequent optimization passes would follow. This is purely speculative and research needs to be conducted to test this theory.

5. Automated cartilage modeling:

a. Simplify code/process – Due to a shift in the dissertation research path, the GPU-based cartilage modeling code was left in a "rough on the edges" state. The user is forced to jump between Mathematica and two Visual Studio CUDA projects to arrive at the final solution. Refinement of this code would make this toolkit much more amenable for use in real-world applications.

b. Streamline computation – The toolkit was the author's first attempt at creating a useful tool that took advantage of GPU-computing. A lot of knowledge and experience has been gained since the toolkit's introduction and many of the computations could benefit from a light overhaul.

6. Joint capsule modeling:

    a. Test geometric skeleton approach – The final and least discussed topic in this dissertation was the idea that geometric skeletons could be used to generate patient-specific joint capsule models. The strategy relies on the idea of using the exterior geometric skeletons from a pair of adjacent bones to identify the surface that is equidistant to each of the surfaces of the bone. Once identified, skeletal points could be identified as potential regions for "growing" the joint capsule model by increasing the radii of the skeletal balls located at those particular skeletal points. This alternative approach was not pursued to completion and remains as a suitable topic for future investigations.

We have presented two GPU-based software toolkits for markerless 2D-3D image registration and biomechanical modeling. Both toolkits were designed with the user in mind and provide a coding environment that shields the user from GPU-specific programming. The image registration software, DRRACC/DRRACO, was validated using methods ranging from

quantifying the image formation method to application of DRRACC/DRRACO to real imaging data, with the results being compared to the CPUDRR benchmark. DRRACC/DRRACO achieved sub-millimeter and sub-degree precision, which is required for successful bone tracking and matches, or in some cases, exceeds the precision of the CPU software.

The more significant result of this dissertation was the realization of up to a 150-fold increase in computational performance over CPUDRR, a very promising and exciting result for the future of real-time medical image registration and other imaging applications. One implication for this result is the ability to process gait records for studies that contain large numbers of subjects; completing such a study using CPUDRR is a nearly impossible task, with computation times being the limiting factor. Using DRRACC/DRRACO will allow the scientific community to gain a deeper understanding of foot bone kinematics as a direct result of increasing the quantity of processed gait records. Similarly, researchers will have the capacity to study a variety of foot disorders and compare those findings to normal foot mechanics, which would also not be possible without DRRACC/DRRACO.

Further advancements in GPU hardware and software will lead to major breakthroughs in the medical imaging and biomechanical industries and we hope that DRRACC/DRRACO will help pave the way for such innovations.

Bibliography

1. Shamos, M.I., *Computational Geometry*. 1978, Yale University. p. 246.
2. Preparata, F.P. and Shamos, M.I., *Computational geometry: an introduction*. 1985: Springer-Verlag.
3. Hatze, H., *What is Biomechanics.* Leibesiibungen-Leibeserziehung, 1971. **2**: p. 33-34.
4. Hatze, H., *Meaning of Term Biomechanics.* Journal of Biomechanics, 1974. **7**(2): p. 189-190.
5. Dirac, P.A.M., *The evolution of the physicist's picture of nature.* Scientific American, 1963. **208**: p. 45-53.
6. Khamene, A., Chisu, R., Wein, W., Navab, N. and Sauer, F., *A novel projection based approach for medical image registration.* Biomedical Image Registration, 2006: p. 247-256.
7. Weese, J., Goecke, R., Penney, G.P., Desmedt, P., Buzug, T.M. and Schumann, H. *Fast voxel-based 2D/3D registration algorithm using a volume rendering method based on the shear-warp factorization*. in *SPIE Medical Imaging*. 1999. International Society for Optics and Photonics.
8. Ledoux, W.R., Tsai, R., Sangeorzan, M.J.F.B.J. and Haynor, D.R. *Using Biplane Flouroscopy to Quantify Foot Bone Motion*. in *American Society of Biomechanics Annual Meeting*. 2010. Providence, RI.
9. Spoerk, J., Gendrin, C., Weber, C., Figl, M., Pawiro, S.A., Furtado, H., Fabri, D., Bloch, C., Bergmann, H. and Gröller, E., *High-performance GPU-based rendering for real-time, rigid 2D/3D-image registration and motion prediction in radiation oncology.* Zeitschrift für Medizinische Physik, 2012. **22**(1): p. 13-20.
10. Kubias, A., Deinzer, F., Feldmann, T., Paulus, D., Schreiber, B. and Brunner, T., *2D/3D image registration on the GPU.* Pattern Recognition and Image Analysis, 2008. **18**(3): p. 381-389.
11. Russakoff, D.B., Rohlfing, T., Mori, K., Rueckert, D., Ho, A., Adler Jr, J.R. and Maurer Jr, C.R., *Fast generation of digitally reconstructed radiographs using attenuation fields with application to 2D-3D image registration.* Medical Imaging, IEEE Transactions on, 2005. **24**(11): p. 1441-1454.
12. Ruijters, D., ter Haar Romeny, B.M. and Suetens, P., *GPU-accelerated digitally reconstructed radiographs.* BioMED, 2008. **8**: p. 431-435.
13. Spoerk, J., Bergmann, H., Wanschitz, F., Dong, S. and Birkfellner, W., *Fast DRR splat rendering using common consumer graphics hardware.* Medical physics, 2007. **34**: p. 4302.
14. LaRose, D.A., *Iterative X-ray/CT registration using accelerated volume rendering*. 2001, Carnegie Mellon University.
15. Sarrut, D. and Clippe, S., *Fast DRR generation for intensity-based 2D/3D image registration in radiotherapy.* LIRIS UMR, 2003. **5205**.
16. Kim, K., Park, S., Hong, H. and Shin, Y.G. *Fast 2D-3D registration using GPU-based preprocessing*. in *HEALTHCOM 2005*. 2005. IEEE.
17. Mori, S., Kobayashi, M., Kumagai, M. and Minohara, S., *Development of a GPU-based multithreaded software application to calculate digitally reconstructed radiographs for radiotherapy.* Radiological physics and technology, 2009. **2**(1): p. 40-45.
18. Ino, F., Gomita, J., Kawasaki, Y. and Hagihara, K., *A GPGPU Approach for Accelerating 2-D/3-D Rigid Registration of Medical Images.* Parallel and Distributed Processing and Applications, 2006. **4330**: p. 939-950.
19. GPGPU.org. *GPGPU*. 2012 [cited 2012 December]; Available from: www.GPGPU.org.
20. Tornai, G.J., Cserey, G. and Pappas, I., *Fast DRR generation for 2D to 3D registration on GPUs.* Medical Physics, 2012. **39**(8): p. 4795-4799.
21. Grosland, N.M. and Brown, T.D., *A voxel-based formulation for contact finite element analysis.* Computer Methods Biomechanics Biomedical Engineering, 2002. **5**(1): p. 21-32.
22. Camacho, D., Ledoux, W., Rohr, E., Sangeorzan, B., Ching, R., *A three-dimensional, anatomically detailed foot model: A foundation for a finite element simulation and means of quantifying foot-bone position.* Journal of Rehabilitation Research and Development, 2002. **39**(3): p. 401-410.
23. Ledoux, W.R., Camacho, D.L., Ching, R.P. and Sangeorzan, B.J., *The Development and Validation of a Computational Foot and Ankle Model*, in *The World Congress on Medical Physics and Biomedical Engineering*. 2000: Chicago.

24. Ledoux, W.R., Dengler, E.D.W. and Fassbind, M.J. *A Finite Element Foot Model for Simulating Muscle Imbalances*. in *Proceedings of the 1st International Foot and Ankle Biomechanics Congress*. 2008. Bologna, Italy.
25. Dengler, E.D.W., *A Finite Element Model of the Human Foot and Ankle*, in *Mechanical Engineering*. 2008, University of Washington: Seattle, WA.
26. Cheung, J.T., Zhang, M. and An, K.N., *Effect of Achilles tendon loading on plantar fascia tension in the standing foot.* Clin Biomech (Bristol, Avon), 2006. **21**(2): p. 194-203.
27. Cheung, J.T. and Zhang, M., *A 3-dimensional finite element model of the human foot and ankle for insole design.* Arch Phys Med Rehabil, 2005. **86**(2): p. 353-8.
28. Budhabhatti, S.P., Erdemir, A., Petre, M., Sferra, J., Donley, B. and Cavanagh, P.R., *Finite element modeling of the first ray of the foot: a tool for the design of interventions.* J Biomech Eng, 2007. **129**(5): p. 750-6.
29. Chen, W.M., Lee, T., Lee, P.V., Lee, J.W. and Lee, S.J., *Effects of internal stress concentrations in plantar soft-tissue-A preliminary three-dimensional finite element analysis.* Med Eng Phys, 2010. **32**(4): p. 324-31.
30. Harrysson, O.L., Hosni, Y.A. and Nayfeh, J.F., *Custom-designed orthopedic implants evaluated using finite element analysis of patient-specific computed tomography data: femoral-component case study.* BMC Musculoskelet Disord, 2007. **8**: p. 91.
31. Sarrafian, S.K., *Anatomy of the Foot and Ankle: Descriptive, Topographic, Functional*. 1993, Philadelphia, PA.: J.B. Lippincott.
32. Athanasiou, K.A., Liu, G.T., Lavery, L.A., Lanctot, D.R. and Schenck, R.C., Jr., *Biomechanical topography of human articular cartilage in the first metatarsophalangeal joint.* Clin Orthop Relat Res, 1998(348): p. 269-81.
33. Gilbert, S.L., Moore, D.C., Case, J.A., Crisco, J.J., *Quantification of Carpal Cartilage Facet Morphology using Micro-CT*, in *55th Annual Meeting of the Orthopaedic Research Society*. 2009: Las Vegas, NV. p. Poster No. 1157.
34. Han, S.K., Federico, S., Epstein, M. and Herzog, W., *An articular cartilage contact model based on real surface geometry.* J Biomech, 2005. **38**(1): p. 179-84.
35. Anderson, D.D., Goldsworthy, J.K., Li, W., James Rudert, M., Tochigi, Y. and Brown, T.D., *Physical validation of a patient-specific contact finite element model of the ankle.* J Biomech, 2007. **40**(8): p. 1662-9.
36. Marai, G.E., Crisco, J.J. and Laidlaw, D.H., *A kinematics-based method for generating cartilage maps and deformations in the multi-articulating wrist joint from CT images.* Conf Proc IEEE Eng Med Biol Soc, 2006. **1**: p. 2079-82.
37. Garcia-Aznar, J.M., Bayod, J., Rosas, A., Larrainzar, R., Garcia-Bogalo, R., Doblare, M. and Llanos, L.F., *Load transfer mechanism for different metatarsal geometries: a finite element study.* J Biomech Eng, 2009. **131**(2): p. 021011.
38. Marchelli, G.L.S., Ledoux, W.R., Isvilanonda, V., Ganter, M.A. and Storti, D.W. *An Automated Method for Creation of Patient-Specific Volumetric Articular Cartilage Elements in the Human Foot*. in *The 2011 Design of Medical Devices Conference*. 2011. Minneapolis, MN.
39. Sanders, J. and Kandrot, E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2010, Upper Saddle River, N.J.: Addison-Wesley.
40. OpenGL. 1997 [cited 2013 January 09]; Available from: www.opengl.org.
41. Stam, J. *What every CUDA programmer should know about OpenGL*. in *GPU Technology Conference, San Jose, CA*. 2009.
42. Microsoft. *DirectX Programming*. 2014 [cited 2014 August]; Available from: http://msdn.microsoft.com/en-us/library/windows/apps/bg182880.aspx.
43. Kirk, D. and Hwu, W.-m., *Programming massively parallel processors : a hands-on approach*. 2010, Burlington, MA: Morgan Kaufmann Publishers.
44. NVIDIA. *CUDA*. 2012 [cited 2011 January]; Available from: http://www.nvidia.com/object/cuda_home_new.html.
45. NVIDIA, *CUDA C Programming Guide*. 2014.
46. Wilt, N., *The cuda handbook: A comprehensive guide to gpu programming*. 2013: Pearson Education.
47. NVIDIA. *CUDA Programming Model Overview*. 2008 [cited 2010; Available from: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf.

48. Zeller, C. *CUDA C/C++ Supercomputing 2011 Tutorial*. 2011 [cited 2011; Available from: http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.
49. Ebersole, M. *Getting Started with CUDA C/C++*. in *GPU Technology Conference*. 2013. San Jose, CA: NVIDIA.
50. Bey, M.J., Zauel, R., Brock, S.K. and Tashman, S., *Validation of a new model-based tracking technique for measuring three-dimensional, in vivo glenohumeral joint kinematics.* Journal of Biomechanical Engineering, 2006. **128**(4): p. 604.
51. Tashman, S. and Anderst, W., *In-vivo measurement of dynamic joint motion using high speed biplane radiography and CT: application to canine ACL deficiency.* TRANSACTIONS-AMERICAN SOCIETY OF MECHANICAL ENGINEERS JOURNAL OF BIOMECHANICAL ENGINEERING, 2003. **125**(2): p. 238-245.
52. Tashman, S., Collon, D., Anderson, K., Kolowich, P. and Anderst, W., *Abnormal rotational knee motion during running after anterior cruciate ligament reconstruction.* The American journal of sports medicine, 2004. **32**(4): p. 975-983.
53. Bey, M.J., Kline, S.K., Tashman, S. and Zauel, R., *Accuracy of biplane x-ray imaging combined with model-based tracking for measuring in-vivo patellofemoral joint motion.* Journal of orthopaedic surgery and research, 2008. **3**.
54. Miranda, D.L., Schwartz, J.B., Loomis, A.C., Brainerd, E.L., Fleming, B.C. and Crisco, J.J., *Static and dynamic error of a biplanar videoradiography system using marker-based and markerless tracking techniques.* Journal of Biomechanical Engineering, 2011. **133**(12).
55. Iaquinto, J.M., Tsai, R., Haynor, D.R., Fassbind, M., Sangeorzan, B.J. and Ledoux, W.R., *Marker-Basede Validation of a Biplane Fluoroscopy System for Quantifying Foot Kinematics.* Medical Engineering & Physics, 2014. **36**(3): p. 391-396.
56. Iaquinto, J.M., Tsai, R., Vu, Q.-B., Haynor, D.R., Sangeorzan, B.J. and Ledoux, W.R. *Preliminary Model-Based Validation of a Biplane Fluoroscopy System*. in *Proceedings of the 4th International Foot and Ankle Biomechanics Congress*. 2014. Pusan, South Korea.
57. Hu, Y., Ledoux, W.R., Fassbind, M., Rohr, E.S., Sangeorzan, B.J. and Haynor, D., *Multi-rigid image segmentation and registration for the analysis of joint motion from three-dimensional magnetic resonance imaging.* Journal of Biomechanical Engineering, 2011. **133**(10): p. 101005.
58. Hsieh, J. *Computed tomography: principles, design, artifacts, and recent advances*. 2009. SPIE Bellingham, WA.
59. Vanden Berghen, F. and Bersini, H., *CONDOR, a new parallel, constrained extension of Powell's UOBYQA algorithm: Experimental results and comparison with the DFO algorithm.* Journal of computational and applied mathematics, 2005. **181**(1): p. 157-175.
60. Microsoft. *Visual Studio Developer Tools and Languages*. 2014 [cited 2014 August]; Available from: http://msdn.microsoft.com/en-us/library/vstudio.
61. Owen, G.S. *Ray-Box Intersection*. 1998 [cited 2012 May]; Available from: http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm.
62. Penney, G.P., Weese, J., Little, J.A., Desmedt, P. and Hill, D.L.G., *A comparison of similarity measures for use in 2-D-3-D medical image registration.* Medical Imaging, IEEE Transactions on, 1998. **17**(4): p. 586-595.
63. Wu, J., Kim, M., Peters, J., Chung, H. and Samant, S.S., *Evaluation of similarity measures for use in the intensity-based rigid 2D-3D registration for patient positioning in radiotherapy.* Medical physics, 2009. **36**(12): p. 5391-5403.
64. Haber, E. and Modersitzki, J., *Beyond mutual information: A simple and robust alternative*, in *Bildverarbeitung für die Medizin 2005*. 2005, Springer. p. 350-354.
65. Fabien, B.C., *Parameter optimization using the< i> L</i>< sub>∞</sub> exact penalty function and strictly convex quadratic programming problems.* Applied Mathematics and Computation, 2008. **198**(2): p. 833-848.
66. Swann, W., *A survey of non-linear optimization techniques.* FEBS letters, 1969. **2**: p. S39-S55.
67. Fabien, B.C., *linfsqp and l1sqp: The implementation of two algorithms*

for the solution of nonlinear programming problems. University of Washington: Seattle, WA.
68. Powell, M.J., *The BOBYQA algorithm for bound constrained optimization without derivatives.* Cambridge NA Report NA2009/06, University of Cambridge, Cambridge, 2009.

69. Vanden Berghen, F., *CONDOR User's Guide*, in *Applied Science - IRIDIA*. 2006, Université Libre de Bruxelles. p. 54.
70. Powell, M.J., *UOBYQA: Unconstrained Optimization by Quadratic Approximation.* Mathematical Programming, 2002. **92**(3): p. 555-582.
71. Wolfram Research, I. *Mathematica Edition: Version 8.0*. 2010 02.08.2011]; Available from: www.wolfram.com.
72. Wang, J. and Blackburn, T.J., *The AAPM/RSNA Physics Tutorial for Residents: X-ray Image Intensifiers for Fluoroscopy 1.* Radiographics, 2000. **20**(5): p. 1471-1477.
73. Pio, R.L., *Euler angle transformations.* Automatic Control, IEEE Transactions on, 1966. **11**(4): p. 707-715.
74. Eberly, D., *Distance Between Point and Triangle in 3D*. 2008, Geometric Tools, LLC.
75. Payne, B.A. and Toga, A.W., *Distance field manipulation of surface models.* Computer Graphics and Applications, IEEE, 1992. **12**(1): p. 65-71.
76. Jones, M.W., *3D Distance from a Point to a Triangle*. 1995, University of Wales Swansea.
77. Wang, S.W. and Kaufman, A.E. *Volume sampled voxelization of geometric primitives*. in *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*. 1993.
78. Aspert, N., Santa-Cruz, D. and Ebrahimi, T. *MESH: measuring errors between surfaces using the Hausdorff distance*. in *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*. 2002.
79. Rustamov, R.M., Lipman, Y. and Funkhouser, T., *Interior Distance Using Barycentric Coordinates.* Computer Graphics Forum, 2009. **28**(5): p. 1279-1288.
80. Bærentzen, J.A., Henrik, A., *Generating Signed Distance Fields from Triangle Meshes*. 2002, IMM Technical University of Denmark: Denmark.
81. Fuhrmann, S., *Volume Data Generation from Triangle Meshes Using the Signed Distance Function*, in *Department of Computer Science*. 2007, Darmstadt University of Technology, Germany: Darmstadt.
82. Mark, W.J., *3D Distance Fields: A Survey of Techniques and Applications.* IEEE Transactions on Visualization and Computer Graphics, 2006. **12**: p. 581-599.
83. Marai, G.E., Laidlaw, D.H., Demiralp, C., Andrews, S., Grimm, C.M. and Crisco, J.J., *Estimating joint contact areas and ligament lengths from bone kinematics and surfaces.* IEEE Trans Biomed Eng, 2004. **51**(5): p. 790-9.
84. Sud, A., Otaduy, M.A. and Manocha, D., *DiFi: Fast 3D distance field computation using graphics hardware.* Computer Graphics Forum, 2004. **23**(3): p. 557-566.
85. Sud, A., Govindaraju, N., Gayle, R. and Manocha, D., *Interactive 3D distance field computation using linear factorization*, in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006, ACM: Redwood City, California. p. 117-124.
86. Cignoni, P., Rocchini, C. and Scopigno, R., *Metro: Measuring Error on Simplified Surfaces.* Computer Graphics Forum, 1998. **17**(2): p. 167-174.
87. Sud, A., Andersen, E., Curtis, S., Lin, M. and Manocha, D., *Real-time path planning for virtual agents in dynamic environments*, in *ACM SIGGRAPH 2008 classes*. 2008, ACM: Los Angeles, California. p. 1-9.
88. Pineda, J. *A Parallel Algorithm for Polygon Rasterization*. in *In Proc. SIGGRAPH '88* 1988. Atlanta, Georgia, August 1-5, 1988: ACM SIGGRAPH, New York.
89. Laval, B.P., *Mathematics for Computer Graphics-Barycentric Coordinates*. 2003, Kennesaw State University.
90. Mauch, S., *A Fast Algorithm for Computing the Closest Point and Distance Transform*. 2000, CALTECH ASCI TECHNICAL REPORT 077. p. 1-17.
91. Sullivan, K., Perry, R. and Frisken, S., *Adaptively sampled distance fields.* Computer Graphics-Us, 2002. **36**(1): p. 3-3.
92. Erleben, K., Dohlmann, H., *Chapter 34. Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra*, in *GPU Gems 3*, H. Nguyen, Editor. 2007, Pearson Education, Inc.: Boston.
93. Kenneth E. Hoff, I., Keyser, J., Lin, M., Manocha, D. and Culver, T., *Fast computation of generalized Voronoi diagrams using graphics hardware*, in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, ACM Press/Addison-Wesley Publishing Co. p. 277-286.

94. Mauch, S.P., *Efficient algorithms for solving static Hamilton-Jacobi equations.* 2003, California Institute of Technology. p. 264.
95. Danielsson, P.E., *Euclidean Distance Mapping.* Computer Graphics and Image Processing, 1980. **14**(3): p. 227-248.
96. Gueziec, A., *"Meshsweeper": Dynamic point-to-polygonal-mesh distance and applications.* IEEE Transactions on Visualization and Computer Graphics, 2001. **7**(1): p. 47-61.
97. Sigg, C., Peikert, R. and Gross, M., *Signed distance transform using graphics hardware.* 2007: p. 83-90.
98. Anderson, D.D., Goldsworthy, J.K., Shivanna, K., Grosland, N.M., Pedersen, D.R., Thomas, T.P., Tochigi, Y., Marsh, J.L. and Brown, T.D., *Intra-articular contact stress distributions at the ankle throughout stance phase-patient-specific finite element analysis as a metric of degeneration propensity.* Biomech Model Mechanobiol, 2006. **5**(2-3): p. 82-9.
99. Muehleman, C. and Kuettner, K.E., *Distribution of cartilage thickness on the head of the human first metatarsal bone.* J Anat, 2000. **197 Pt 4**: p. 687-91.
100. El-Khoury, G.Y., Alliman, K. J., Lundberg, H. J., Rudert, M. J., Brown, T. D., Saltzman, C. L., *Cartilage thickness in cadaveric ankles: measurement with double-contrast multi-detector row CT arthrography versus MR imaging.* Radiology, 2004. **233**(3): p. 768-73.
101. Blum, H., *A Transformation for Extracting New Descriptors of Shape.*, in *Models for the Perception of Speech and Visual Forms*, Wathen-Dunn, Editor. 1967, MIT Press: Amsterdam. p. 362-380.
102. Storti, D.W., Turkiyyah, G.M., Ganter, M.A., Lim, C.T. and Stal, D.M., *Skeleton-based modeling operations on solids*, in *Proceedings of the fourth ACM symposium on Solid modeling and applications*. 1997, ACM: Atlanta, Georgia, United States. p. 141-154.
103. Attali, D. and Montanvert, A., *Computing and Simplifying 2D and 3D Continuous Skeletons.* Computer Vision and Image Understanding, 1997. **67**(3): p. 261-273.
104. Ju, T., Baker, M.L. and Chiu, W., *Computing a family of skeletons of volumetric models for shape description.* Computer-Aided Design, 2007. **39**(5): p. 352-360.
105. Dutta, D. and Hoffmann, C.M., *On the Skeleton of Simple CSG Objects.* Journal of Mechanical Design, 1993. **115**(1): p. 87-94.
106. Bonnassie, A., Peyrin, F. and Attali, D., *A new method for analyzing local shape in three-dimensional images based on medial axis transformation.* Ieee Transactions on Systems Man and Cybernetics Part B-Cybernetics, 2003. **33**(4): p. 700-705.
107. Marchelli, G., Ledoux, W., Isvilanonda, V., Ganter, M. and Storti, D., *Joint-specific distance thresholds for patient-specific approximations of articular cartilage modeling in the first ray of the foot.* Medical & biological engineering & computing, 2014. **52**(9): p. 773-779.
108. Carrigan, S.D., Whiteside, R.A., Pichora, D.R. and Small, C.F., *Development of a three-dimensional finite element model for carpal load transmission in a static neutral posture.* Annals of Biomedical Engineering, 2003. **31**(6): p. 718-725.
109. Gebhardt, A., *Rapid Prototyping*. 2003, Munich: Hanser Verlag. 379.
110. Séquin, C.H., *Procedural spline interpolation in UNICUBIX*. 1987, University of California, Computer Science Division: Berkeley, CA.
111. ThÜrmer, G. and WÜthrich, C.A., *Computing vertex normals from polygonal facets.* J. Graph. Tools, 1998. **3**(1): p. 43-46.
112. McNeel. 2010 12.24.2010]; Available from: http://www.rhino3d.com/.
113. Bærentzen, J.A. and Aanaes, H., *Signed Distance Computation Using the Angle Weighted Pseudonormal.* IEEE Transactions on Visualization and Computer Graphics, 2005. **11**(3): p. 243-253.
114. Shereff, M.J., Bejjani, F.J. and Kummer, F.J., *Kinematics of the first metatarsophalangeal joint.* J Bone Joint Surg Am, 1986. **68**(3): p. 392-8.

Appendices

Appendix A: CUDA Specifications and Compatibilities

Technical specifications	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535					2^{31}-1		
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum number of resident warps per multiprocessor	24		32		48	64		
Maximum number of resident threads per multiprocessor	768		1024		1536	2048		
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		
Maximum number of 32-bit registers per thread	128				63	255		
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Number of shared memory banks	16				32			
Amount of local memory per thread	16 KB				512 KB			
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB							10 KB
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				12 KB	Between 12 KB and 48 KB		24 KB
Maximum width for 1D texture reference bound to a CUDA array	8192				65536			
Maximum width for 1D texture reference bound to linear memory	2^{27}							
Maximum width and number of layers for a 1D layered texture reference	8192 × 512				16384 × 2048			
Maximum width and height for 2D texture reference bound to a CUDA array	65536 × 32768				65536 × 65535			
Maximum width and height for 2D texture reference bound to a linear memory	65000 × 65000				65000 × 65000			
Maximum width and height for 2D texture reference bound to a CUDA array supporting texture gather	N/A				16384 × 16384			
Maximum width, height, and number of layers for a 2D layered texture reference	8192 × 8192 × 512				16384 × 16384 × 2048			

Maximum width, height and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 × 2048 × 2048		4096 × 4096 × 4096
Maximum width (and height) for a cubemap texture reference	N/A		16384
Maximum width (and height) and number of layers for a cubemap layered texture reference	N/A		16384 × 2046
Maximum number of textures that can be bound to a kernel	128		256
Maximum width for a 1D surface reference bound to a CUDA array	Not supported		65536
Maximum width and number of layers for a 1D layered surface reference			65536 × 2048
Maximum width and height for a 2D surface reference bound to a CUDA array			65536 × 32768
Maximum width, height, and number of layers for a 2D layered surface reference			65536 × 32768 × 2048
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array			65536 × 32768 × 2048
Maximum width (and height) for a cubemap surface reference bound to a CUDA array			32768
Maximum width (and height) and number of layers for a cubemap layered surface reference			32768 × 2046
Maximum number of surfaces that can be bound to a kernel		8	16
Maximum number of instructions per kernel	2 million		512 million

Appendix B: DRRACO Functions Accessible by the User

```
// DRRACC Main Prototypes
#pragma once // Prevent multiple inclusions
#include "DRRACC_Includes.h"
#include "DRRACC_Structures.h"


/*************************MISCELLANEOUS Functions*************************/

// Print active device
//// Inputs:
/// int deviceID - Device identifier (0 = default; 0 or 1 for multi-GPU
/// machines)
//// Output:
/// Integer error code (TBD)
//// Comments:
/// This will print out the device name corresponding to deviceID
int checkGPU(int deviceID);

// Choose active device (ONLY for machines with multiple GPUs)
//// Inputs:
/// int deviceID - Device identifier (0 = default; 0 or 1 for multi-GPU
/// machines)
//// Output:
/// Integer error code (TBD)
//// Comments:
/// This will set the active device corresponding to deviceID
int setGPU(int deviceID);

// Find axis-aligned bounding boxes and compute corresponding centroids
//// Inputs:
/// int *label - Pointer to label file data
/// hostVoxelGeometry hVox - Host structure containing voxel geometry
/// boneData boneInfo[NROWS] - Array of host structures containing bone data
//// Output:
/// boneData* - Array of host structures containing bone data
//// Comments:
/// This function is intended for preprocessing only
boneData* findCentroids(int *label, hostVoxelGeometry hVox, boneData
boneInfo[NROWS]);

// Free memory (device pointers)
//// Inputs:
/// deviceImage d - Device image
//// Output:
/// Integer error code (TBD)
//// Comments:
/// Free pointers prior to the final return in main()
int Free(deviceImage d);
```

```
// Free memory (device pointers)
//// Inputs:
/// deviceImage *d - Pointer to device image
//// Output:
/// Integer error code (TBD)
//// Comments:
/// Free pointers prior to the final return in main()
int Free(deviceImage *d);

// Free memory (host pointers)
//// Inputs:
/// hostImage h - Host image
//// Output:
/// Integer error code (TBD)
//// Comments:
/// Free pointers prior to the final return in main()
int Free(hostImage h);

// Reset device
//// Inputs:
/// N/A
//// Output:
/// Integer error code (TBD)
//// Comments:
/// Reset device after freeing all device pointers
int reset();

/*****************************MAKE Functions*****************************/

// Allocate and initialize host image
//// Inputs:
/// int width - image width
/// int height - image height
/// float initVal - initialization value
//// Output:
/// hostImage structure
//// Function:
/// Allocate and initialize host image
//// Comments:
/// hostImage.ptr should be freed at end of program
hostImage makeHostImage(int width, int height, float initVal);

// Allocate and initialize host image
//// Inputs:
/// deviceImage dImg - device image structure
/// float initVal - initialization value
//// Output:
/// hostImage structure
//// Function:
```

```
/// Allocate and initialize host image based on device image parameters
//// Comments:
/// hostImage.ptr should be freed at end of program
hostImage makeHostImage(deviceImage dImg, float initVal);

// Allocate and initialize host image
//// Inputs:
/// deviceImage dImg - device image structure
/// float initVal - initialization value
/// int copy - flag that determines whether device image will be copied to
/// host (0 = no; 1 = yes)
//// Output:
/// hostImage structure
//// Function:
/// Allocate and initialize host image (optional: copy device image to host)
//// Comments:
/// hostImage.ptr should be freed at end of program
hostImage makeHostImage(deviceImage dImg, float initVal, int copy);

// Allocate and initialize device image
//// Inputs:
/// int width - image width
/// int height - image height
/// float initVal - initialization value
//// Output:
/// deviceImage structure
//// Function:
/// Allocate and initialize device image
//// Comments:
/// deviceImage.ptr should be CUDA freed at end of program
deviceImage makeDeviceImage(int width, int height, float initVal);

// Allocate and initialize device image
//// Inputs:
/// hostImage hImg - host image structure
/// float initVal - initialization value
//// Output:
/// deviceImage structure
//// Function:
/// Allocate and initialize device image based on host image parameters
//// Comments:
/// deviceImage.ptr should be CUDA freed at end of program
deviceImage makeDeviceImage(hostImage hImg, float initVal);

// Allocate and initialize device image
//// Inputs:
/// hostImage hImg - host image structure
/// float initVal - initialization value
/// int copy - flag that determines whether device image will be copied to
device (0 = no; 1 = yes)
```

```
//// Output:
/// deviceImage structure
//// Function:
///  Allocate and initialize device image (optional: copy host image to
/// device)
//// Comments:
/// deviceImage.ptr should be CUDA freed at end of program
deviceImage makeDeviceImage(hostImage hImg, float initVal, int copy);

// Create host structure to store voxel geometry
//// Inputs:
/// None
//// Output:
/// Host voxel geometry structure
//// Function:
/// Allocate/initialize host voxel geometry structure
//// Comments:
/// No data is passed to the function explicitly; the host structure is
/// declared (i.e., hostVoxelGeometry hVox = makeHostVoxelGeo();) and the
/// function allocates/initializes the structure
/// The user can then fill the structure
hostVoxelGeometry makeHostVoxelGeo();

// Create device structure to store voxel geometry
//// Inputs:
/// hostVoxelGeometry hVoxGeo - host structure containing voxel information
//// Output:
/// Device voxel geometry structure
//// Function:
/// Allocate/initialize device voxel geometry structure and copy host info
//// Comments:
/// Here we return a pointer to the structure which is necessary for use in
/// kernel functions
deviceVoxelGeometry* makeDeviceVoxelGeo(hostVoxelGeometry hVoxGeo);

// Create host structure to store fluoro geometry
//// Inputs:
/// None
//// Output:
/// Host fluoro geometry structure
//// Function:
/// Allocate/initialize host fluoro geometry structure
//// Comments:
/// No data is passed to the function explicitly; the host structure is
///  declared (i.e., hostFluoroGeometry hFluoro = makeHostFluoroGeo();) and
/// the function allocates/initializes the structure the user can then fill
/// the structure
hostFluoroGeometry makeHostFluoroGeo();

// Load fluoroscope geometry and store in device structure
```

```
//// Inputs:
/// fluoroGeometry *hGeometry - host structure containing Real32 fluoroscope
/// geometry
//// Output:
/// Device fluoro geometry structure
//// Function:
/// Copy fluoroscope geometry to device structure
//// Comments:
///  This function performs all of the necessary bookkeeping tasks except
/// freeing memory
deviceFluoroGeometry* makeDeviceFluoroGeo(hostFluoroGeometry hostStruct);

// Create host structure to store bone data
//// Inputs:
/// N/A
//// Output:
/// Host bone data structure
//// Comments:
/// This function initializes the host structure
boneData makeBoneData();

// Create device structure to store bone data
//// Inputs:
/// boneData *hBoneInfo - Array of host boneData structures (length = NROWS)
//// Output:
/// Pointer to bone data device structure
//// Comments:
///  This function performs all of the necessary bookkeeping tasks except
/// freeing memory
deviceBoneData* makeDeviceBoneData(boneData *hBoneInfo);

// Create Region of Interest (ROI)
//// Inputs:
/// deviceImage dImg - device image (DRR-based) used to find tightest ROI
//// Output:
/// Pointer to length = 4 array containing ROI coordinates {minX, maxX, minY,
/// maxY}
//// Comments:
/// This function requires that the input be a DRR or a binary mask
/// Inputs with background noise will produce unpredictable ROIs
int* makeROI(deviceImage dImg);

/**************************TRANSFER Functions**************************/

// Load image and store in device array
//// Inputs:
/// float *imagePTR - pointer to host image
/// float *devicePTR - name of device pointer that was declared by user (not
/// initialized/allocated)
/// int imageW - number of columns in image
```

```
/// int imageH - number of rows in image
//// Output:
/// Integer error code (value TBD)
//// Function:
/// Copy host image to device
//// Commments:
/// The user MUST declare the device pointer
/// There is no need to explicitly return the device pointer; the user will
/// simply declare/allocate a device pointer and will be able to use that
/// image until Free(devicePTR) is invoked
int loadImage(hostImage hImg, deviceImage dImg);

// Copy image from device to host memory
//// Inputs:
/// float *devicePTR - pointer to device image
/// float *imagePTR - pointer to host image
/// int imageW - number of columns in image
/// int imageH - number of rows in image
//// Output:
/// Integer error code (value TBD)
//// Function:
/// Copy device image to host
//// Comments:
/// The user will declare/malloc/initialize the host array that will store
/// the image copied from the device
int extractImage(deviceImage dImg, hostImage hImg);

// Transfer device image to device
//// Inputs:
/// deviceImage dImg - device image structure
/// deviceImage dImg2 - device image structure
//// Output:
/// deviceImage structure
//// Function:
/// Copy device image to another device image
//// Comments:
/// All deviceImage.ptr should be CUDA freed at end of program
int deviceImageTransfer(deviceImage dImg, deviceImage dImg2);

// Load CT data and store in texture memory
//// Inputs:
/// float *density - pointer to Real32 CT density data
/// int *label - pointer to Integer32 CT label data
/// voxelGeometry hVoxGeo - host structure containing {x, y, z} voxel count
/// (Integer32) & voxel spacing (Real32)
//// Output:
/// deviceVoxelGeometry structure
//// Function:
/// Copy density and label data into 3D texture memory
/// Copy voxel geometry to device structure
```

```
//// Comments:
/// Pointers to texture memory are not useful; this function performs
/// all of the necessary bookkeeping tasks associated with texture memory
deviceVoxelGeometry* loadCTData(float *density, int *label, hostVoxelGeometry
hVoxGeo);

/************************IMAGE PROCESSING Functions************************/

// A mask that is neither dilated nor eroded can be created by simply
// passing '0' for the dilationSz/erosionSz parameter

// Create a dilated mask by dilationSz = number of pixels
//// Inputs:
/// deviceImage dImg - device image (dilation source)
/// deviceImage dMask - masked device image (declared/allocated/initialized
/// by user)
/// int dilationSz - dilation parameter (number of pixels)
//// Output:
/// Integer error code (value TBD)
//// Function:
/// This function operates on the device source image and populates a device
/// mask image
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dMask.ptr) is invoked
int dilateMask(deviceImage dImg, deviceImage dMask, int dilationSz);

// Create an eroded mask by erosionSz = number of pixels
//// Inputs:
/// deviceImage dImg - device image (dilation source)
/// deviceImage dMask - device image mask (declared/allocated/initialized by
/// user)
/// int erosionSz - dilation parameter (number of pixels)
//// Output:
/// Integer error code (value TBD)
//// Function:
/// This function operates on the device source image and populates a device
/// mask image
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dMask.ptr) is invoked
int erodeMask(deviceImage dImg, deviceImage dMask, int erosionSz);

// Apply a mask to an image
//// Inputs:
/// deviceImage dImg - device image (dilation source)
/// deviceImage dMask - device image mask (declared/allocated/initialized by
/// user)
/// deviceImage dMaskedImage - masked device image (declared/allocated/
/// initialized by user)
```

```
//// Output:
/// Integer error code (value TBD)
//// Function:
/// This function applies a device mask to the device image and
/// populates a device masked image
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dMaskedImage.ptr) is invoked
int applyMask(deviceImage dImg, deviceImage dMask, deviceImage dMaskedImage);

// Convolve an image with a kernel
//// Inputs:
/// deviceImage dImg - device image (source)
/// deviceImage dConvolvedImage - convolved device image (declared/allocated/
/// initialized by user)
/// float *kernel - pointer to kernel array (host)
/// int halfWidth - kernel half width ((full width - 1) / 2)
//// Output:
/// Integer error code (value TBD)
//// Function:
/// This function convolves a device image with a device kernel and
/// populates a device convolved image
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dConvolvedImage.ptr) is invoked
int imageConvolve(deviceImage dImg, deviceImage dConvolvedImage, float
*kernel, int halfWidth);

// Use finite difference approximation to compute a gradient image
//// Inputs:
/// deviceImage dImg - device image (source)
/// deviceImage dGradImage - gradient device image (declared/allocated/
/// initialized by user)
/// int gradType - allows the user to choose between forward difference
/// (gradType = 0),
/// two-term (gradType = 1) and four-term (gradType = 2) central difference
/// int component - binary switch for i- or j-component
//// Function:
/// This function computes a directional image gradient based on a user-
/// specified finite difference method
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dGradImage.ptr) is invoked
int finiteDiffGradient(deviceImage dImg, deviceImage dGradImage, int
gradType, int component);

// Composite images
//// Inputs:
/// deviceImage dDRRs[MAXOBJECTS + 1] - array of device images (source)
/// deviceImage dCompositeImage - composite device image
```

```
/// int compositeList[MAXOBJECTS + 1] - list of flags for making bones
/// invisible (0) or visible (1)
/// int type = switch for adding or subtracting images
//// Output:
/// Integer error code (value TBD)
//// Function:
/// Composite up to MAXOBJECTS + 1 images on the device
//// Comments:
/// The user will simply create a deviceImage and will be able to use that
/// image until Free(dCompositeImage.ptr) is invoked
int compositeImages(deviceImage dDRRs[MAXOBJECTS + 1], deviceImage
dCompositeImage, int compositeList[MAXOBJECTS + 1], int type);

// Compute DRRs
//// Inputs:
/// deviceIMage bDRRs[MAXOBJECTS + 1] - array of device images (blue)
/// deviceIMage gDRRs[MAXOBJECTS + 1] - array of device images (green)
/// int visibleBones[MAXOBJECTS + 1] - binary (0/1) array that switches bone
/// visibility off/on
/// float tMatrices[MAXOBJECTS + 1][12] - one per bone transformation
/// matrices stored as flattened Real32
/// arrays of length(12) in the form: {#, #, #, tx, #, #, #, ty, #, #, #, tz}
/// deviceVoxelGeometry *dVox - pointer to device structure that contains
/// voxel geometry
/// deviceFluoroGeometry *dFluoro - pointer to device structure that contains
/// fluoroscope geometry
/// boneData boneInfo[MAXOBJECTS + 1] - array of boneData (centroid, bounding
/// box normals/displacements, optimization variables)
/// int copyDRRs - flag that allows individual DRRs to be saved (0 = no; 1 =
/// yes)
/// The default (copyDRRs = 0) only saves the composited DRR
/// outputData dataOut - output structure that will store DRRs, NCCs, etc.
//// Output:
/// Integer error code (value TBD)
//// Function:
/// This function computes and composites the visible DRRs
//// Comments:
/// The user MUST declare/allocate/initizialize the output structure
outputData computeDRR(deviceImage bDRRs[MAXOBJECTS + 1], deviceImage
gDRRs[MAXOBJECTS + 1], int visibleBones[MAXOBJECTS + 1], deviceVoxelGeometry
*dVox, deviceFluoroGeometry *dFluoro, boneData boneInfo[MAXOBJECTS + 1],
outputData dataOut);

// Compute NCC for two images
//// Inputs:
/// deviceImage drrPTR - device image1
/// deviceImage fluoroPTR - device image2
//// Output:
/// Real32 correlation value
//// Function:
```

```
/// Compute correlation between two device images
//// Comments:
/// All images MUST be in device memory
float calculateNCC(deviceImage drrPTR, deviceImage fluoroPTR);
float calculateNCC(deviceImage drrPTR, deviceImage fluoroPTR, int ROI[4]);

// Perform all necessary tasks for a single optimization call
//// Inputs:
/// deviceImage bDRRs[MAXOBJECTS + 1] - Individual DRRs (blue)
/// deviceImage gDRRs[MAXOBJECTS + 1] - Individual DRRs (green)
/// deviceImage bFluoro - Fluoro image (blue)
///   deviceImage gFluoro - Fluoro image (green)
///   deviceImage bEdgeFluoroi - i-component gradient image (blue fluoro)
///   deviceImage bEdgeFluoroj - j-component gradient image (blue fluoro)
///   deviceImage gEdgeFluoroi - i-component gradient image (green fluoro)
///   deviceImage gEdgeFluoroj - j-component gradient image (green fluoro)
///   int gradType - Forward = 0; 2-term central = 1; 4-term central = 2
///   deviceVoxelGeometry *dVox - Device voxel geometry
///   deviceFluoroGeometry *dFluoro - device fluoro geometry
///   boneData boneInfo[NROWS] - Bone centroids, axis-aligned bounding box
/// min/max, intial position
///   CONDORparams cParams - CONDOR optimization parameters
///   int optBones[MAXOBJECTS + 1] - Optimization bone list
/// (0 = do not optimize; 1 = do optimize)
///   int visibility[MAXOBJECTS + 1] - Visibility bone list (0 = invisible;
/// 1 = visible)
///   outputData dataOut - Output data structure
//// Output:
/// outputData - Output data structure
//// Comments:
/// Input data must be freed after use
///  All parameters must be defined prior to function execution for
/// predictable behavior
outputData exampleMeritFunction(deviceImage bDRRs[NROWS],
                                deviceImage gDRRs[NROWS],
                                deviceImage bFluoro,
                                deviceImage gFluoro,
                                deviceImage bEdgeFluoroi,
                                deviceImage bEdgeFluoroj,
                                deviceImage gEdgeFluoroi,
                                deviceImage gEdgeFluoroj,
                                int gradType,
                                deviceVoxelGeometry *dVox,
                                deviceFluoroGeometry *dFluoro,
                                boneData boneInfo[NROWS],
                                CONDORparams cParams,
                                int optBones[NROWS],
                                int visibility[NROWS],
                                outputData dataOut);
```

```
/**************************CONDOR Functions**************************/

// Perform all necessary tasks for a single optimization call
//// Inputs:
/// deviceImage bDRRs[MAXOBJECTS + 1] - Individual DRRs (blue)
/// deviceImage gDRRs[MAXOBJECTS + 1] - Individual DRRs (green)
/// deviceImage bFluoro - Fluoro image (blue)
///   deviceImage gFluoro - Fluoro image (green)
///   deviceImage bEdgeFluoroi - i-component gradient image (blue fluoro)
///   deviceImage bEdgeFluoroj - j-component gradient image (blue fluoro)
///   deviceImage gEdgeFluoroi - i-component gradient image (green fluoro)
///   deviceImage gEdgeFluoroj - j-component gradient image (green fluoro)
///   int gradType - Forward = 0; 2-term central = 1; 4-term central = 2
///   deviceVoxelGeometry *dVox - Device voxel geometry
///   deviceFluoroGeometry *dFluoro - device fluoro geometry
///   boneData boneInfo[NROWS] - Bone centroids, axis-aligned bounding box
/// min/max, intial position
///   CONDORparams cParams - CONDOR optimization parameters
///   int optBones[MAXOBJECTS + 1] - Optimization bone list
/// (0 = do not optimize; 1 = do optimize)
///   int visibility[MAXOBJECTS + 1] - Visibility bone list
/// (0 = invisible; 1 = visible)
///   outputData dataOut - Output data structure
//// Output:
/// outputData - Output data structure
//// Comments:
/// Input data must be freed after use
///  All parameters must be defined prior to function execution for
/// predictable behavior
typedef outputData (*meritFunc)(deviceImage bDRRs[NROWS],
                                deviceImage gDRRs[NROWS],
                                deviceImage bFluoro,
                                deviceImage gFluoro,
                                deviceImage bEdgeFluoroi,
                                deviceImage bEdgeFluoroj,
                                deviceImage gEdgeFluoroi,
                                deviceImage gEdgeFluoroj,
                                int gradType,
                                deviceVoxelGeometry *dVox,
                                deviceFluoroGeometry *dFluoro,
                                boneData boneInfo[NROWS],
                                CONDORparams cParams,
                                int optBones[NROWS],
                                int visibility[NROWS],
                                outputData dataOut);

// CONDOR class and sub-classes
class DRRACCObjFun : public CONDOR::ObjectiveFunction
{
      public:
```

```
        DRRACCObjFun(int _t = 0);
        ~DRRACCObjFun(){}; // destructor
        float xInit[NROWS][NVAR]; // this is initial value of variables
        float xCurr[NROWS][NVAR]; // this is current value of variables
        double eval(CONDOR::Vector v, int *nerror=NULL);
        int setVariables(boneData boneInfo[NROWS], const int OptBones[NROWS],
        const int Visibility[NROWS], CONDORparams cParams);
        int setBounds(const float lowBound[NVAR], const float upperBound[NVAR],
        const int flag); // set bounds constraints
        int setMerit(meritFunc mc); // set merit function
        // Copy other necessary inputs to DRRACCObjFun class
        int setDRRs(deviceImage bDRR[NROWS], deviceImage gDRR[NROWS]);
        int setFluoros(deviceImage bFluoro, deviceImage gFluoro);
        int setFluoroEdge(deviceImage bFluoroEdgei, deviceImage bFluoroEdgej,
                          deviceImage gFluoroEdgei, deviceImage gFluoroEdgej,
                          int gradType);
        int setBoneData(boneData boneInfo[NROWS]);
        int setGeometry(deviceVoxelGeometry *dVox, deviceFluoroGeometry
                        *dFluoro);
        int setOutput(outputData dataOut);
        int updateVariables(CONDOR::Vector X);

private:
        meritFunc DRRACCmerit;   // merit function pointer
        int nBones; // number of bones being optimized
        int error; // error code (0=success)
        deviceImage bDRR[NROWS], gDRR[NROWS];
        deviceImage bFluoro, gFluoro;
        deviceImage bFluoroEdgei, bFluoroEdgej;
        deviceImage gFluoroEdgei, gFluoroEdgej;
        int gradType;
        deviceVoxelGeometry *voxGeo;
        deviceFluoroGeometry *fluoroGeo;
        boneData boneParams[NROWS];
        CONDORparams conParams;
        int optBones[NROWS], visibility[NROWS];
        outputData output;
        Vector xStart; Vector bu; Vector bl; int isConstrained;
}; // end class definition

// Perform all necessary tasks for invoking CONDOR
//// Inputs:
///  meritFunc mc - Pointer to merit function that will be used during
/// optimization
///   int optBones[MAXOBJECTS + 1] - Optimization bone list
/// (0 = do not optimize; 1 = do optimize)
///   int visibility[MAXOBJECTS + 1] - Visibility bone list
/// (0 = invisible; 1 = visible)
///   deviceVoxelGeometry *dVox - Device voxel geometry
///   deviceFluoroGeometry *dFluoro - device fluoro geometry
```

```
///   CONDORparams cParams - CONDOR optimization parameters
///   int gradType - Forward = 0; 2-term central = 1; 4-term central = 2
/// deviceImage bDRRs[MAXOBJECTS + 1] - Individual DRRs (blue)
/// deviceImage gDRRs[MAXOBJECTS + 1] - Individual DRRs (green)
/// deviceImage bFluoro - Fluoro image (blue)
///   deviceImage gFluoro - Fluoro image (green)
///   deviceImage bEdgeFluoroi - i-component gradient image (blue fluoro)
///   deviceImage bEdgeFluoroj - j-component gradient image (blue fluoro)
///   deviceImage gEdgeFluoroi - i-component gradient image (green fluoro)
///   deviceImage gEdgeFluoroj - j-component gradient image (green fluoro)
///   boneData boneInfo[NROWS] - Bone centroids, axis-aligned bounding box
/// min/max, intial position
/// float lowerBounds[6] - Lower bounds for constrained optimization
/// float upperBounds[6] - Upper bounds for constrained optimization
///   outputData dataOut - Output data structure
//// Output:
/// outputData - Output data structure
//// Comments:
/// Input data must be freed after use
///  All parameters must be defined prior to function execution for
/// predictable behavior
outputData runCONDOR(meritFunc mc, int optBones[NROWS], int
visibility[NROWS], deviceVoxelGeometry *dVox, deviceFluoroGeometry *dFluoro,
CONDORparams cParams, int gradType, deviceImage bDRRs[NROWS], deviceImage
gDRRs[NROWS], deviceImage bFluoro, deviceImage gFluoro, deviceImage
bFluoroEdgei, deviceImage bFluoroEdgej, deviceImage gFluoroEdgei, deviceImage
gFluoroEdgej, boneData boneInfo[NROWS], float lowerBounds[6], float
upperBounds[6], outputData dataOut);
```

Appendix C: DRRACO Example Main

```
#include "../inc/DRRACC_MainPrototypes.h"
#include "../inc/DRRACC_CONDORCaller.h"

int setup_transformation(float *transformationData); // Simple function for assigning data from 1D to 2D array
float transformation[MAXOBJECTS + 1][12] = {}; // Declare/allocate/initialize transformation arrays
// This example main will introduce the user to the image processing tools
// available in DRRACO
// The purpose of this example is to provide actual examples of how data
// structures are created
// and utilized. This example demonstrates the various capabilities of
// DRRACO

int main()
{
	int deviceID = 0; // Choose device to query
	// (0 = default; 0 or 1 IFF multi-GPU machine)
	checkGPU(deviceID); // Check active device
	setGPU(deviceID); // Set active device
	// (ONLY for machines with multiple GPUs)

	int imageW = width; // Set image width
	int imageH = height; // Set image height
	// declare/malloc/init host image (blue fluoro)
	hostImage hBfluoro = makeHostImage(imageW, imageH, 0.0);
	// declare/malloc/init host image (green fluoro)
	hostImage hGfluoro = makeHostImage(imageW, imageH, 0.0);
	// declare/malloc/init device image (blue fluoro)
	deviceImage dBfluoro = makeDeviceImage(imageW, imageH, 0.0);
	// declare/malloc/init device image (green fluoro)
	deviceImage dGfluoro = makeDeviceImage(imageW, imageH, 0.0);

	//************************//FLUORO IMAGES//************************//

	FILE *staticImageFile = fopen("data.raw", "rb");
	if(staticImageFile == NULL)
	{
		perror("Error loading blue fluoro image\n");
	}
	fread(hBfluoro.ptr, sizeof(float), imageW * imageH, staticImageFile);
	fclose(staticImageFile);
	staticImageFile = fopen("data.raw", "rb");
	if(staticImageFile == NULL)
	{
		perror("Error loading green fluoro image\n");
	}
	fread(hGfluoro.ptr, sizeof(float), imageW * imageH, staticImageFile);
```

```
fclose(staticImageFile);

// Copy fluoro images to device images
loadImage(hBfluoro, dBfluoro);
loadImage(hGfluoro, dGfluoro);
// Free host memory when finished
free(hBfluoro.ptr);
free(hGfluoro.ptr);

//**********************//LABEL DATA//**********************//

// Declare/allocate/initialize host struct for voxel geometry
hostVoxelGeometry hVox1 = makeHostVoxelGeo();
hVox1.voxelCount = make_int3(input); // Set voxel count
hVox1.voxelSpacing = make_float3(input); // Set voxel spacing
size_t    size    =    hVox1.voxelCount.x    *    hVox1.voxelCount.y    *
hVox1.voxelCount.z;
int *labelData;
labelData = (int *)malloc(sizeof(int) * size);
FILE *labelDataFile = fopen("data.raw", "rb");
if(labelDataFile == NULL)
{
      perror("THE FOLLOWING ERROR OCCURRED WITH (labelFilename)");
      exit(EXIT_FAILURE);
}
size_t read = fread(labelData, 1, size*sizeof(int), labelDataFile);
fclose(labelDataFile);

//**********************//DENSITY DATA//**********************//

float *densityData;
densityData = (float *)malloc(sizeof(float) * size);
FILE *densityDataFile = fopen("data.raw", "rb");
if(densityDataFile == NULL)
{
      perror("THE FOLLOWING ERROR OCCURRED WITH (densityFilename)");
      exit(EXIT_FAILURE);
}
read = fread(densityData, 1, size*sizeof(float), densityDataFile);
fclose(densityDataFile);

//**********************//TRANSFORMATION DATA//**********************//

float transformationData[12 * (MAXOBJECTS + 1)] = {};
FILE *transformationDataFile = fopen("data.bin", "rb");
if(transformationDataFile == NULL)
{
      perror("THE   FOLLOWING   ERROR   OCCURRED   WITH   (TRANSFORMATION
      FILE)\n");
}
```

```
fread(transformationData, sizeof(float), (MAXOBJECTS + 1) * 12,
transformationDataFile);
fclose(transformationDataFile);
setup_transformation(transformationData);

//*********************//COPY VOXEL GEOMETRY//*********************//

deviceVoxelGeometry *dVox = loadCTData(densityData, labelData, hVox1);

//************************//FLUORO GEOMETRY//************************//

hostFluoroGeometry hostData = makeHostFluoroGeo();
hostData.blueFocus = make_float3(input);
hostData.blueUpperLeft = make_float3(input);
hostData.blueUpperRight = make_float3(input);
hostData.blueLowerLeft = make_float3(input);
hostData.greenFocus = make_float3(input);
hostData.greenUpperLeft = make_float3(input);
hostData.greenUpperRight = make_float3(input);
hostData.greenLowerLeft = make_float3(input);
hostData.width = 1152;
hostData.height = 896;

deviceFluoroGeometry *dFluoro = makeDeviceFluoroGeo(hostData);

//****//PREPARE IMAGE CONTAINERS & INITIALIZE INPUT PARAMETERS//****//

outputData drrOUT; // Declare data output struct
drrOUT.copyDRRs = 1; // Copies individual drrs to output structure
// (0 = no; 1 = yes) -- composite DRR image automatically is output
drrOUT.intensifier = 2; // Toggle DRR
// (0 = Blue ONLY; 1 = Green ONLY; 2 = Blue & Green)
drrOUT.blueDRR = makeHostImage(imageW, imageH, 0.0);
drrOUT.greenDRR = makeHostImage(imageW, imageH, 0.0);
boneData *boneInfo;
boneInfo = (boneData*)malloc(NROWS * sizeof(boneData));
deviceImage db_drrs[MAXOBJECTS + 1];
deviceImage dg_drrs[MAXOBJECTS + 1];
int visibility[MAXOBJECTS + 1] = {};
int optBones[MAXOBJECTS + 1] = {};
CONDORparams cParams; // Declare CONDOR optimization struct
cParams.constrained = 1; // Set constraint flag (0 = no; 1 = yes)
cParams.edgeWeight = 2.0/7.0; // Set NCC edge weighting
cParams.intensityWeight = 1. - cParams.edgeWeight; // Set weighting
cParams.dilationRadius = 15; // Set mask dilation radius
cParams.maxIters = 5000; // Max number of iterations
cParams.rhoStart = 22.0; // Starting search step size
cParams.rhoEnd = 0.015; // Ending search step size

for(int i = 0; i <= MAXOBJECTS; i++)
```

```
{
	visibility[i] = 1; // Set all bones to "visible"
	drrOUT.bDRRs[i] = makeHostImage(imageW, imageH, 0.0);
	drrOUT.gDRRs[i] = makeHostImage(imageW, imageH, 0.0);
	db_drrs[i] = makeDeviceImage(imageW, imageH, 0.0);
	dg_drrs[i] = makeDeviceImage(imageW, imageH, 0.0);

	boneInfo[i] = makeBoneData();
	boneInfo[i].variables[0] = transformation[i][3];
	boneInfo[i].variables[1] = transformation[i][7];
	boneInfo[i].variables[2] = transformation[i][11];
	boneInfo[i].variables[3] = 10.0*RAD;
	boneInfo[i].variables[4] = -10.0*RAD;
	boneInfo[i].variables[5] = 5.0*RAD;

	optBones[i] = 0; // Set all bones to "do not optimize"
}
int bone = 10;
optBones[bone] = 1; // We want at least one bone to be optimized or
// computeNCC will not return a correlation value

boneInfo = findCentroids(labelData, hVox1, boneInfo);

//************************//COMPUTE DRRs//************************//

drrOUT = computeDRR(	db_drrs, dg_drrs, visibility, dVox, dFluoro,
			boneInfo, drrOUT);

if(VERBOSE == 1)
{
	printf("\nWriting DRRs (computeDRR())\n");
	FILE *drrs; // output filename
	drrs = fopen("output.bin", "wb");
	fwrite(drrOUT.blueDRR.ptr, sizeof(float), imageW * imageH, drrs);
	fclose(drrs);
	drrs = fopen("output.bin", "wb");
	fwrite(drrOUT.greenDRR.ptr, sizeof(float), imageW * imageH,
		drrs);
	fclose(drrs);
}

//*****************//COMPUTE FLUORO EDGE IMAGES//*****************//

deviceImage bEdgeFluoroi = makeDeviceImage(imageW, imageH, 0.0);
deviceImage bEdgeFluoroj = makeDeviceImage(imageW, imageH, 0.0);
deviceImage gEdgeFluoroi = makeDeviceImage(imageW, imageH, 0.0);
deviceImage gEdgeFluoroj = makeDeviceImage(imageW, imageH, 0.0);

finiteDiffGradient(dBfluoro, bEdgeFluoroi, 2, 0); // Compute 4-term
// central difference i-component gradient
```

```
finiteDiffGradient(dBfluoro, bEdgeFluoroj, 2, 1);
finiteDiffGradient(dGfluoro, gEdgeFluoroi, 2, 0);
finiteDiffGradient(dGfluoro, gEdgeFluoroj, 2, 1);

if(VERBOSE == 1)
{
	hostImage temp = makeHostImage(bEdgeFluoroi, 0.0, 1);
	printf("Writing Fluoro Gradients\n");
	FILE *grads; // output filename
	grads = fopen("TestOutput//blueEdgei.bin", "wb");
	fwrite(temp.ptr, sizeof(float), imageW * imageH, grads);
	fclose(grads);
	temp = makeHostImage(bEdgeFluoroj, 0.0, 1);
	grads = fopen("TestOutput//blueEdgej.bin", "wb");
	fwrite(temp.ptr, sizeof(float), imageW * imageH, grads
	fclose(grads);
	temp = makeHostImage(gEdgeFluoroi, 0.0, 1);
	grads = fopen("TestOutput//greenEdgei.bin", "wb");
	fwrite(temp.ptr, sizeof(float), imageW * imageH, grads);
	fclose(grads);
	temp = makeHostImage(gEdgeFluoroj, 0.0, 1);
	grads = fopen("TestOutput//greenEdgej.bin", "wb");
	fwrite(temp.ptr, sizeof(float), imageW * imageH, grads);
	fclose(grads);
}

//**********************//COMPOSITE IMAGES//**********************//

deviceImage dBcomp = makeDeviceImage(imageW, imageH, 0.0);  deviceImage
dGcomp = makeDeviceImage(imageW, imageH, 0.0);

compositeImages(db_drrs, dBcomp, visibility, 0); // Composite blue DRRs
// (0 = add; 1 = subtract)
compositeImages(dg_drrs, dGcomp, visibility, 0);
hostImage hBcomp = makeHostImage(dBcomp, 0.0, 1);
hostImage hGcomp = makeHostImage(dGcomp, 0.0, 1);
// Note these images should identically match the blueDRR and greenDRR
images output above

if(VERBOSE == 1)
{
	printf("Writing Composite Images\n");
	FILE *comp; // output filename
	comp = fopen("TestOutput//blueComposite.bin", "wb");
	fwrite(hBcomp.ptr, sizeof(float), imageW * imageH, comp);
	fclose(comp);
	comp = fopen("TestOutput//greenComposite.bin", "wb");
	fwrite(hGcomp.ptr, sizeof(float), imageW * imageH, comp);
	fclose(comp);
```

```
	free(hBcomp.ptr);
	free(hGcomp.ptr);
}

//*********************//COMPUTE & APPLY MASKS//*********************//

deviceImage dBMask = makeDeviceImage(imageW, imageH, 0.0);
deviceImage dGMask = makeDeviceImage(imageW, imageH, 0.0);
deviceImage dBMasked = makeDeviceImage(imageW, imageH, 0.0);
deviceImage dGMasked = makeDeviceImage(imageW, imageH, 0.0);

dilateMask(db_drrs[bone], dBMask, 15); // Compute a dilated masks for
// optBones[bone] = 1 set above (15 pixel dilation radius)
	dilateMask(dg_drrs[bone], dGMask, 15);

applyMask(dBfluoro, dBMask, dBMasked); // Apply masks to fluoro images
applyMask(dGfluoro, dGMask, dGMasked);

hostImage hBMasked = makeHostImage(dBMasked, 0.0, 1); hostImage
hGMasked = makeHostImage(dGMasked, 0.0, 1);

if(VERBOSE == 1)
{
	printf("Writing Masked Images\n");
	FILE *masked; // output filename
	masked = fopen("TestOutput//blueMasked.bin", "wb");
	fwrite(hBMasked.ptr, sizeof(float), imageW * imageH, masked);
	fclose(masked);
	masked = fopen("TestOutput//greenMasked.bin", "wb");
	fwrite(hGMasked.ptr, sizeof(float), imageW * imageH, masked);
	fclose(masked);

	free(hBMasked.ptr); // Free host memory when finished
	free(hGMasked.ptr);
}

//*********************//IMAGE CONVOLUTION//*********************//

deviceImage conv = makeDeviceImage(imageW, imageH, 0.0);
float kern[121] = userDefined; // 11 x 11 Gaussian kernel (sigma = 2.0)
int halfWidth = 5; // Kernel half width = (full width - 1) / 2
imageConvolve(dBcomp, conv, kern, halfWidth); // Convolve

if(VERBOSE == 1)
{
	printf("Writing Convolved Image\n");
	hostImage check = makeHostImage(imageW, imageH, 0.0); // Reset
	extractImage(conv, check);
	FILE *convOUT;
	convOUT = fopen("TestOutput//convolution.bin", "wb");
```

```
	fwrite(check.ptr, sizeof(float), imageW * imageH, convOUT);
	fclose(convOUT);
	free(check.ptr); // Free host memory when finished
}

//************************//CALCULATE NCC//************************//

// The use of the same image for both arguments is just a simple test
// exampleMeritfunction() below will return real NCC between
// fluoros/DRRs
float blueNCC = calculateNCC(dBcomp, dBcomp);
float greenNCC = calculateNCC(dGcomp, dGcomp);

blueNCC = 0.0; greenNCC = 0.0;
int *bROI;
int *gROI;
bROI = (int*)malloc(4 * sizeof(int));
gROI = (int*)malloc(4 * sizeof(int));
bROI = makeROI(dBMask);
gROI = makeROI(dGMask);

blueNCC = calculateNCC(dBcomp, dBcomp, bROI);
greenNCC = calculateNCC(dGcomp, dGcomp, gROI);

//********************//EXAMPLE MERIT FUNCTION//********************//

// Reset device images
for(int i = 0; i <= MAXOBJECTS; i++)
{
	cudaMemset(db_drrs[i].ptr, 0, imageW * imageH);
	cudaMemset(dg_drrs[i].ptr, 0, imageW * imageH);
}

drrOUT = exampleMeritFunction(	db_drrs,
				dg_drrs,
				dBfluoro,
				dGfluoro,
				bEdgeFluoroi,
				bEdgeFluoroj,
				gEdgeFluoroi,
				gEdgeFluoroj,
				gradType,
				dVox,
				dFluoro,
				boneInfo,
				cParams,
				optBones,
				visibility,
				drrOUT);
```

```
if(VERBOSE == 1)
{
	printf("\nWriting DRRs (exampleMeritFunction())\n");
	// These images should match the images in files "blueDRR.bin"
	// and "greenDRR.bin" output above
	FILE *drrs; // output filename
	drrs = fopen("TestOutput//blueDRR_exampleMerit.bin", "wb");
	fwrite(drrOUT.blueDRR.ptr, sizeof(float), imageW * imageH, drrs);
	fclose(drrs);
	drrs = fopen("TestOutput//greenDRR_exampleMerit.bin", "wb");
	fwrite(drrOUT.greenDRR.ptr, sizeof(float), imageW * imageH,
		drrs);
	fclose(drrs);
}

//*************************//RUN CONDOR//*************************//

float tb = 153.0; // Set translation bounds
float rb = 65.0 * (PI/180.0); // Set rotation bounds
float lowBounds[6] = {-tb, -tb, -tb, -rb, -rb, -rb};
float hiBounds[6] = {tb, tb, tb, rb, rb, rb};

drrOUT = runCONDOR(	exampleMeritFunction,
			optBones,
			visibility,
			dVox,
			dFluoro,
			cParams,
			2,
			db_drrs,
			dg_drrs,
			dBfluoro,
			dGfluoro,
			bEdgeFluoroi,
			bEdgeFluoroj,
			gEdgeFluoroi,
			gEdgeFluoroj,
			boneInfo,
			lowBounds,
			hiBounds,
			drrOUT);

if(VERBOSE == 1)
{
	printf("\nWriting DRRs (runCONDOR())\n");
	// These images should match the images in files "blueDRR.bin"
	// and "greenDRR.bin" output above
	FILE *drrs;
	drrs = fopen("testoutput//blueDRR_CONDOR.bin", "wb");
```

```
		fwrite(drrOUT.blueDRR.ptr, sizeof(float), imageW * imageH, drrs);
		fclose(drrs);
		drrs = fopen("testoutput//greenDRR_CONDOR.bin", "wb");
		fwrite(drrOUT.greenDRR.ptr, sizeof(float), imageW * imageH,
			drrs);
		fclose(drrs);
	}

	//*****************//CLEANUP & RESET DEVICE MEMORY//*****************//

	// Free device images
	if(VERBOSE == 1) printf("Freeing Device Memory\n");
	Free(dBfluoro);
	Free(dGfluoro);
	Free(dBcomp);
	Free(dGcomp);
	Free(dBMask);
	Free(dGMask);
	Free(dBMasked);
	Free(dGMasked);
	Free(bEdgeFluoroi);
	Free(bEdgeFluoroj);
	Free(gEdgeFluoroi);
	Free(gEdgeFluoroj);
	for(int i = 0; i <= MAXOBJECTS; i++)
	{
		Free(db_drrs);//[i].ptr);
		Free(dg_drrs);//[i].ptr);
	}
	// Reset GPU
	if(VERBOSE == 1) printf("Device Reset\n");
	cudaDeviceReset();

	return 0;
}
```

Appendix D: Stage Translation Raw Data

Calcaneus 7451 (run 1)

CALCANEUS 7451		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.005	0.000	0.005	0.000	0.000	0.005	0.000
2	0.005	0.005	0.000	0.000	0.022	-0.017	0.000
3	0.005	0.004	0.001	0.000	0.032	-0.027	0.001
4	0.005	0.061	-0.056	0.003	0.033	-0.028	0.001
5	0.005	0.015	-0.010	0.000	0.021	-0.016	0.000
6	0.005	0.071	-0.066	0.004	0.023	-0.018	0.000
7	0.005	0.086	-0.081	0.007	0.055	-0.050	0.003
8	0.005	0.073	-0.068	0.005	0.053	-0.048	0.002
9	0.005	0.084	-0.079	0.006	0.045	-0.040	0.002
10	0.005	0.069	-0.064	0.004	0.028	-0.023	0.001
11	0.104	0.105	-0.001	0.000	0.221	-0.117	0.014
12	0.104	0.109	-0.005	0.000	0.142	-0.038	0.001
13	0.104	0.114	-0.010	0.000	0.143	-0.039	0.002
14	0.104	0.097	0.007	0.000	0.141	-0.037	0.001
15	0.104	0.049	0.055	0.003	0.141	-0.037	0.001
16	0.104	0.070	0.034	0.001	0.124	-0.020	0.000
17	0.104	0.099	0.005	0.000	0.168	-0.064	0.004
18	0.104	0.125	-0.021	0.000	0.130	-0.026	0.001
19	0.104	0.111	-0.007	0.000	0.148	-0.044	0.002
20	0.104	0.090	0.014	0.000	0.135	-0.031	0.001
21	0.202	0.198	0.004	0.000	0.301	-0.099	0.010
22	0.202	0.151	0.051	0.003	0.306	-0.104	0.011
23	0.202	0.164	0.038	0.001	0.332	-0.130	0.017
24	0.202	0.191	0.011	0.000	0.322	-0.120	0.014
25	0.202	0.181	0.021	0.000	0.326	-0.124	0.015
26	0.202	0.183	0.019	0.000	0.292	-0.090	0.008
27	0.202	0.182	0.020	0.000	0.317	-0.115	0.013

28	0.202	0.198	0.004	0.000	0.316	-0.114	0.013
29	0.202	0.208	-0.006	0.000	0.316	-0.114	0.013
30	0.202	0.139	0.063	0.004	0.316	-0.114	0.013
31	0.302	0.282	0.020	0.000	0.405	-0.103	0.011
32	0.302	0.269	0.033	0.001	0.408	-0.106	0.011
33	0.302	0.285	0.017	0.000	0.411	-0.109	0.012
34	0.302	0.270	0.032	0.001	0.437	-0.135	0.018
35	0.302	0.281	0.021	0.000	0.439	-0.137	0.019
36	0.302	0.284	0.018	0.000	0.437	-0.135	0.018
37	0.302	0.270	0.032	0.001	0.427	-0.125	0.016
38	0.302	0.303	-0.001	0.000	0.403	-0.101	0.010
39	0.302	0.267	0.035	0.001	0.395	-0.093	0.009
40	0.302	0.263	0.039	0.002	0.395	-0.093	0.009
41	0.399	0.362	0.037	0.001	0.472	-0.073	0.005
42	0.399	0.359	0.040	0.002	0.469	-0.070	0.005
43	0.399	0.367	0.032	0.001	0.487	-0.088	0.008
44	0.399	0.362	0.037	0.001	0.470	-0.071	0.005
45	0.399	0.390	0.009	0.000	0.469	-0.070	0.005
46	0.399	0.375	0.024	0.001	0.514	-0.115	0.013
47	0.399	0.376	0.023	0.001	0.490	-0.091	0.008
48	0.399	0.345	0.054	0.003	0.474	-0.075	0.006
49	0.399	0.388	0.011	0.000	0.476	-0.077	0.006
50	0.399	0.373	0.026	0.001	0.470	-0.071	0.005
51	0.498	0.490	0.008	0.000	0.593	-0.095	0.009
52	0.498	0.448	0.050	0.002	0.602	-0.104	0.011
53	0.498	0.456	0.042	0.002	0.561	-0.063	0.004
54	0.498	0.483	0.015	0.000	0.576	-0.078	0.006
55	0.498	0.472	0.026	0.001	0.590	-0.092	0.009
56	0.498	0.458	0.040	0.002	0.588	-0.090	0.008
57	0.498	0.452	0.046	0.002	0.584	-0.086	0.007
58	0.498	0.458	0.040	0.002	0.585	-0.087	0.008
59	0.498	0.476	0.022	0.000	0.582	-0.084	0.007

60	0.498	0.441	0.057	0.003	0.589	-0.091	0.008
61	1.003	0.957	0.046	0.002	1.089	-0.086	0.007
62	1.003	0.957	0.046	0.002	1.028	-0.025	0.001
63	1.003	0.957	0.046	0.002	1.115	-0.112	0.013
64	1.003	0.942	0.061	0.004	1.085	-0.082	0.007
65	1.003	0.963	0.040	0.002	1.058	-0.055	0.003
66	1.003	0.971	0.032	0.001	1.127	-0.124	0.015
67	1.003	0.935	0.068	0.005	1.064	-0.061	0.004
68	1.003	0.950	0.053	0.003	1.036	-0.033	0.001
69	1.003	0.968	0.035	0.001	1.037	-0.034	0.001
70	1.003	0.962	0.041	0.002	1.038	-0.035	0.001
71	2.002	1.955	0.047	0.002	2.069	-0.067	0.004
72	2.002	1.947	0.055	0.003	2.073	-0.071	0.005
73	2.002	2.008	-0.006	0.000	2.074	-0.072	0.005
74	2.002	1.979	0.023	0.001	2.075	-0.073	0.005
75	2.002	1.966	0.036	0.001	2.060	-0.058	0.003
76	2.002	1.980	0.022	0.000	2.058	-0.056	0.003
77	2.002	1.978	0.024	0.001	2.055	-0.053	0.003
78	2.002	2.009	-0.007	0.000	2.051	-0.049	0.002
79	2.002	1.959	0.043	0.002	2.090	-0.088	0.008
80	2.002	1.977	0.025	0.001	2.041	-0.039	0.002
81	3.000	2.940	0.060	0.004	2.972	0.028	0.001
82	3.000	2.932	0.068	0.005	2.956	0.044	0.002
83	3.000	2.942	0.058	0.003	2.961	0.039	0.002
84	3.000	2.987	0.013	0.000	3.001	-0.001	0.000
85	3.000	2.956	0.044	0.002	3.047	-0.047	0.002
86	3.000	2.965	0.035	0.001	2.983	0.017	0.000
87	3.000	2.956	0.044	0.002	2.921	0.079	0.006
88	3.000	2.960	0.040	0.002	2.999	0.001	0.000
89	3.000	2.946	0.054	0.003	2.969	0.031	0.001
90	3.000	2.947	0.053	0.003	2.983	0.017	0.000
91	3.999	3.979	0.020	0.000	3.999	0.000	0.000

92	3.999	3.909	0.090	0.008	3.904	0.095	0.009
93	3.999	3.935	0.064	0.004	3.886	0.113	0.013
94	3.999	3.941	0.058	0.003	3.914	0.085	0.007
95	3.999	3.962	0.037	0.001	3.984	0.015	0.000
96	3.999	3.989	0.010	0.000	3.963	0.036	0.001
97	3.999	3.951	0.048	0.002	3.953	0.046	0.002
98	3.999	3.952	0.047	0.002	3.965	0.034	0.001
99	3.999	3.936	0.063	0.004	3.945	0.054	0.003
100	3.999	3.929	0.070	0.005	4.010	-0.011	0.000
101	4.997	4.960	0.037	0.001	4.916	0.081	0.007
102	4.997	4.911	0.086	0.007	4.908	0.089	0.008
103	4.997	4.972	0.025	0.001	4.925	0.072	0.005
104	4.997	4.947	0.050	0.003	4.850	0.147	0.022
105	4.997	4.944	0.053	0.003	4.896	0.101	0.010
106	4.997	4.958	0.039	0.002	4.916	0.081	0.007
107	4.997	4.936	0.061	0.004	4.914	0.083	0.007
108	4.997	4.948	0.049	0.002	4.905	0.092	0.009
109	4.997	4.958	0.039	0.002	5.010	-0.013	0.000
110	4.997	4.985	0.012	0.000	4.893	0.104	0.011
111	9.998	9.903	0.095	0.009	9.887	0.111	0.012
112	9.998	9.918	0.080	0.006	9.855	0.143	0.020
113	9.998	9.937	0.061	0.004	9.861	0.137	0.019
114	9.998	9.877	0.121	0.015	9.797	0.201	0.040
115	9.998	9.920	0.078	0.006	9.832	0.166	0.027
116	9.998	9.920	0.078	0.006	9.874	0.124	0.015
117	9.998	9.941	0.057	0.003	9.833	0.165	0.027
118	9.998	9.892	0.106	0.011	9.818	0.180	0.033
119	9.998	9.940	0.058	0.003	9.866	0.132	0.017
120	9.998	9.930	0.068	0.005	9.909	0.089	0.008
121	15.004	14.923	0.081	0.007	14.891	0.113	0.013
122	15.004	14.911	0.093	0.009	14.767	0.237	0.056
123	15.004	14.905	0.099	0.010	14.844	0.160	0.026

124	15.004	14.921	0.083	0.007	14.771	0.233	0.055
125	15.004	14.908	0.096	0.009	14.786	0.218	0.048
126	15.004	14.916	0.088	0.008	14.813	0.191	0.036
127	15.004	14.929	0.075	0.006	14.846	0.158	0.025
128	15.004	14.943	0.061	0.004	14.819	0.185	0.034
129	15.004	14.898	0.106	0.011	14.816	0.188	0.035
130	15.004	14.890	0.114	0.013	14.773	0.231	0.053
	Max	**14.943**	**0.121**	**0.015**	**14.891**	**0.237**	**0.056**
	Min	**0.004**	**-0.081**	**2.30E-08**	**0.021**	**-0.137**	**2.43E-07**
	Average	**3.158**	**0.036**	**0.003**	**3.201**	**-0.008**	**0.010**
	Standard Deviation	**4.344**	**0.037**	**0.003**	**4.280**	**0.098**	**0.011**

CPU RMSE = 0.051085 mm **DRRACC RMSE = 0.098827 mm**

Calcaneus 7451 (run 2)

CALC 7451		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.005	0.000	0.000	0.000	0.000	0.000	0.000
2	0.005	0.005	-0.005	0.000	0.005	-0.005	0.000
3	0.005	0.004	-0.004	0.000	0.011	-0.011	0.000
4	0.005	0.061	-0.061	0.004	0.040	-0.040	0.002
5	0.005	0.015	-0.015	0.000	0.035	-0.035	0.001
6	0.005	0.071	-0.071	0.005	0.034	-0.034	0.001
7	0.005	0.086	-0.086	0.007	0.034	-0.034	0.001
8	0.005	0.073	-0.073	0.005	0.035	-0.035	0.001
9	0.005	0.084	-0.084	0.007	0.034	-0.034	0.001
10	0.005	0.069	-0.069	0.005	0.038	-0.038	0.001
11	0.104	0.105	-0.006	0.000	0.119	-0.020	0.000
12	0.104	0.109	-0.010	0.000	0.119	-0.020	0.000
13	0.104	0.114	-0.015	0.000	0.086	0.013	0.000
14	0.104	0.097	0.002	0.000	0.088	0.011	0.000
15	0.104	0.049	0.050	0.002	0.088	0.011	0.000
16	0.104	0.070	0.029	0.001	0.077	0.022	0.000
17	0.104	0.099	0.000	0.000	0.102	-0.003	0.000
18	0.104	0.125	-0.026	0.001	0.100	-0.001	0.000
19	0.104	0.111	-0.012	0.000	0.100	-0.001	0.000
20	0.104	0.090	0.009	0.000	0.169	-0.070	0.005
21	0.202	0.198	-0.001	0.000	0.283	-0.086	0.007
22	0.202	0.151	0.046	0.002	0.328	-0.131	0.017
23	0.202	0.164	0.033	0.001	0.257	-0.060	0.004
24	0.202	0.191	0.006	0.000	0.269	-0.072	0.005
25	0.202	0.181	0.016	0.000	0.274	-0.077	0.006
26	0.202	0.183	0.014	0.000	0.280	-0.083	0.007
27	0.202	0.182	0.015	0.000	0.280	-0.083	0.007
28	0.202	0.198	-0.001	0.000	0.294	-0.097	0.009

29	0.202	0.208	-0.011	0.000	0.240	-0.043	0.002
30	0.202	0.139	0.058	0.003	0.282	-0.085	0.007
31	0.302	0.282	0.015	0.000	0.322	-0.025	0.001
32	0.302	0.269	0.028	0.001	0.336	-0.039	0.002
33	0.302	0.285	0.012	0.000	0.332	-0.035	0.001
34	0.302	0.270	0.027	0.001	0.332	-0.035	0.001
35	0.302	0.281	0.016	0.000	0.332	-0.035	0.001
36	0.302	0.284	0.013	0.000	0.332	-0.035	0.001
37	0.302	0.270	0.027	0.001	0.297	0.000	0.000
38	0.302	0.303	-0.006	0.000	0.325	-0.028	0.001
39	0.302	0.267	0.030	0.001	0.325	-0.028	0.001
40	0.302	0.263	0.034	0.001	0.325	-0.028	0.001
41	0.399	0.362	0.032	0.001	0.402	-0.008	0.000
42	0.399	0.359	0.035	0.001	0.409	-0.015	0.000
43	0.399	0.367	0.027	0.001	0.405	-0.011	0.000
44	0.399	0.362	0.032	0.001	0.405	-0.011	0.000
45	0.399	0.390	0.004	0.000	0.405	-0.011	0.000
46	0.399	0.375	0.019	0.000	0.413	-0.019	0.000
47	0.399	0.376	0.018	0.000	0.449	-0.055	0.003
48	0.399	0.345	0.049	0.002	0.405	-0.011	0.000
49	0.399	0.388	0.006	0.000	0.408	-0.014	0.000
50	0.399	0.373	0.021	0.000	0.411	-0.017	0.000
51	0.498	0.490	0.003	0.000	0.522	-0.029	0.001
52	0.498	0.448	0.045	0.002	0.520	-0.027	0.001
53	0.498	0.456	0.037	0.001	0.447	0.046	0.002
54	0.498	0.483	0.010	0.000	0.497	-0.004	0.000
55	0.498	0.472	0.021	0.000	0.490	0.003	0.000
56	0.498	0.458	0.035	0.001	0.523	-0.030	0.001
57	0.498	0.452	0.041	0.002	0.524	-0.031	0.001
58	0.498	0.458	0.035	0.001	0.503	-0.010	0.000
59	0.498	0.476	0.017	0.000	0.491	0.002	0.000
60	0.498	0.441	0.052	0.003	0.491	0.002	0.000

61	1.003	0.957	0.041	0.002	1.048	-0.050	0.003
62	1.003	0.957	0.041	0.002	1.023	-0.025	0.001
63	1.003	0.957	0.041	0.002	1.023	-0.025	0.001
64	1.003	0.942	0.056	0.003	1.015	-0.017	0.000
65	1.003	0.963	0.035	0.001	0.984	0.014	0.000
66	1.003	0.971	0.027	0.001	1.004	-0.006	0.000
67	1.003	0.935	0.063	0.004	0.980	0.018	0.000
68	1.003	0.950	0.048	0.002	0.980	0.018	0.000
69	1.003	0.968	0.030	0.001	0.980	0.018	0.000
70	1.003	0.962	0.036	0.001	0.990	0.008	0.000
71	2.002	1.955	0.042	0.002	2.022	-0.025	0.001
72	2.002	1.947	0.050	0.002	2.020	-0.023	0.001
73	2.002	2.008	-0.011	0.000	2.024	-0.027	0.001
74	2.002	1.979	0.018	0.000	2.030	-0.033	0.001
75	2.002	1.966	0.031	0.001	2.023	-0.026	0.001
76	2.002	1.980	0.017	0.000	2.026	-0.029	0.001
77	2.002	1.978	0.019	0.000	2.029	-0.032	0.001
78	2.002	2.009	-0.012	0.000	1.999	-0.002	0.000
79	2.002	1.959	0.038	0.001	2.007	-0.010	0.000
80	2.002	1.977	0.020	0.000	2.000	-0.003	0.000
81	3.000	2.940	0.055	0.003	2.917	0.078	0.006
82	3.000	2.932	0.063	0.004	2.871	0.124	0.015
83	3.000	2.942	0.053	0.003	2.873	0.122	0.015
84	3.000	2.987	0.008	0.000	2.920	0.075	0.006
85	3.000	2.956	0.039	0.002	2.938	0.057	0.003
86	3.000	2.965	0.030	0.001	2.890	0.105	0.011
87	3.000	2.956	0.039	0.002	2.890	0.105	0.011
88	3.000	2.960	0.035	0.001	2.898	0.097	0.009
89	3.000	2.946	0.049	0.002	2.945	0.050	0.002
90	3.000	2.947	0.048	0.002	2.943	0.052	0.003
91	3.999	3.979	0.015	0.000	3.964	0.030	0.001
92	3.999	3.909	0.085	0.007	3.848	0.146	0.021

93	3.999	3.935	0.059	0.004	3.892	0.102	0.010
94	3.999	3.941	0.053	0.003	3.892	0.102	0.010
95	3.999	3.962	0.032	0.001	3.896	0.098	0.010
96	3.999	3.989	0.005	0.000	3.891	0.103	0.011
97	3.999	3.951	0.043	0.002	3.889	0.105	0.011
98	3.999	3.952	0.042	0.002	3.889	0.105	0.011
99	3.999	3.936	0.058	0.003	3.899	0.095	0.009
100	3.999	3.929	0.065	0.004	3.951	0.043	0.002
101	4.997	4.960	0.032	0.001	4.908	0.084	0.007
102	4.997	4.911	0.081	0.006	4.904	0.088	0.008
103	4.997	4.972	0.020	0.000	4.904	0.088	0.008
104	4.997	4.947	0.045	0.002	4.901	0.091	0.008
105	4.997	4.944	0.048	0.002	4.893	0.099	0.010
106	4.997	4.958	0.034	0.001	4.920	0.072	0.005
107	4.997	4.936	0.056	0.003	4.926	0.066	0.004
108	4.997	4.948	0.044	0.002	4.884	0.108	0.012
109	4.997	4.958	0.034	0.001	4.895	0.097	0.009
110	4.997	4.985	0.007	0.000	4.874	0.118	0.014
111	9.998	9.903	0.090	0.008	9.814	0.179	0.032
112	9.998	9.918	0.075	0.006	9.810	0.183	0.034
113	9.998	9.937	0.056	0.003	9.812	0.181	0.033
114	9.998	9.877	0.116	0.013	9.810	0.183	0.034
115	9.998	9.920	0.073	0.005	9.797	0.196	0.038
116	9.998	9.920	0.073	0.005	9.793	0.200	0.040
117	9.998	9.941	0.052	0.003	9.791	0.202	0.041
118	9.998	9.892	0.101	0.010	9.787	0.206	0.042
119	9.998	9.940	0.053	0.003	9.837	0.156	0.024
120	9.998	9.930	0.063	0.004	9.827	0.166	0.027
121	15.004	14.923	0.076	0.006	14.742	0.257	0.066
122	15.004	14.911	0.088	0.008	14.729	0.270	0.073
123	15.004	14.905	0.094	0.009	14.776	0.223	0.050
124	15.004	14.921	0.078	0.006	14.722	0.277	0.077

125	15.004	14.908	0.091	0.008	14.772	0.227	0.051
126	15.004	14.916	0.083	0.007	14.750	0.249	0.062
127	15.004	14.929	0.070	0.005	14.749	0.250	0.062
128	15.004	14.943	0.056	0.003	14.753	0.246	0.061
129	15.004	14.898	0.101	0.010	14.755	0.244	0.060
130	15.004	14.890	0.109	0.012	14.772	0.227	0.051
	Max	**14.943**	**0.116**	**0.013**	**14.776**	**0.277**	**0.077**
	Min	**0.004**	**-0.086**	**2.82E-08**	**0.005**	**-0.131**	**2.58E-08**
	Average	**3.158**	**0.031**	**0.002**	**3.149**	**0.039**	**0.010**
	Standard Deviation	**4.344**	**0.037**	**0.003**	**4.281**	**0.094**	**0.018**

CPU RMSE = 0.047742 mm **DRRACC RMSE = 0.047742 mm**

Calcaneus 7481

CALC 7481		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0	0.000	0.000	0.000	0.000	0.000	0.000
2	0	0.092	-0.092	0.009	0.031	-0.031	0.001
3	0	0.086	-0.086	0.007	0.114	-0.114	0.013
4	0	0.077	-0.077	0.006	0.016	-0.016	0.000
5	0	0.137	-0.137	0.019	0.016	-0.016	0.000
6	0	0.088	-0.088	0.008	0.167	-0.167	0.028
7	0	0.097	-0.097	0.009	0.135	-0.135	0.018
8	0	0.071	-0.071	0.005	0.034	-0.034	0.001
9	0	0.109	-0.109	0.012	0.049	-0.049	0.002
10	0	0.079	-0.079	0.006	0.066	-0.066	0.004
11	0.1	0.118	-0.018	0.000	0.266	-0.166	0.028
12	0.1	0.155	-0.055	0.003	0.284	-0.184	0.034
13	0.1	0.136	-0.036	0.001	0.218	-0.118	0.014
14	0.1	0.191	-0.091	0.008	0.267	-0.167	0.028
15	0.1	0.137	-0.037	0.001	0.275	-0.175	0.031
16	0.1	0.103	-0.003	0.000	0.283	-0.183	0.034
17	0.1	0.172	-0.072	0.005	0.207	-0.107	0.011
18	0.1	0.162	-0.062	0.004	0.189	-0.089	0.008
19	0.1	0.110	-0.010	0.000	0.272	-0.172	0.029
20	0.1	0.125	-0.025	0.001	0.256	-0.156	0.024
21	0.2	0.206	-0.006	0.000	0.210	-0.010	0.000
22	0.2	0.268	-0.068	0.005	0.192	0.008	0.000
23	0.2	0.220	-0.020	0.000	0.245	-0.045	0.002
24	0.2	0.186	0.014	0.000	0.271	-0.071	0.005
25	0.2	0.244	-0.044	0.002	0.266	-0.066	0.004
26	0.2	0.219	-0.019	0.000	0.218	-0.018	0.000
27	0.2	0.211	-0.011	0.000	0.230	-0.030	0.001
28	0.2	0.215	-0.015	0.000	0.229	-0.029	0.001

29	0.2	0.202	-0.002	0.000	0.346	-0.146	0.021
30	0.2	0.210	-0.010	0.000	0.265	-0.065	0.004
31	0.301	0.291	0.010	0.000	0.364	-0.063	0.004
32	0.301	0.303	-0.002	0.000	0.349	-0.048	0.002
33	0.301	0.259	0.042	0.002	0.342	-0.041	0.002
34	0.301	0.286	0.015	0.000	0.415	-0.114	0.013
35	0.301	0.281	0.020	0.000	0.328	-0.027	0.001
36	0.301	0.316	-0.015	0.000	0.371	-0.070	0.005
37	0.301	0.289	0.012	0.000	0.339	-0.038	0.001
38	0.301	0.299	0.002	0.000	0.332	-0.031	0.001
39	0.301	0.282	0.019	0.000	0.369	-0.068	0.005
40	0.301	0.309	-0.008	0.000	0.328	-0.027	0.001
41	0.401	0.383	0.018	0.000	0.354	0.047	0.002
42	0.401	0.366	0.035	0.001	0.398	0.003	0.000
43	0.401	0.377	0.024	0.001	0.452	-0.051	0.003
44	0.401	0.399	0.002	0.000	0.398	0.003	0.000
45	0.401	0.363	0.038	0.001	0.438	-0.037	0.001
46	0.401	0.361	0.040	0.002	0.456	-0.055	0.003
47	0.401	0.362	0.039	0.002	0.456	-0.055	0.003
48	0.401	0.372	0.029	0.001	0.354	0.047	0.002
49	0.401	0.351	0.050	0.002	0.452	-0.051	0.003
50	0.401	0.365	0.036	0.001	0.386	0.015	0.000
51	0.5	0.453	0.047	0.002	0.562	-0.062	0.004
52	0.5	0.464	0.036	0.001	0.539	-0.039	0.002
53	0.5	0.460	0.040	0.002	0.594	-0.094	0.009
54	0.5	0.495	0.005	0.000	0.531	-0.031	0.001
55	0.5	0.461	0.039	0.002	0.539	-0.039	0.002
56	0.5	0.479	0.021	0.000	0.556	-0.056	0.003
57	0.5	0.463	0.037	0.001	0.539	-0.039	0.002
58	0.5	0.469	0.031	0.001	0.539	-0.039	0.002
59	0.5	0.470	0.030	0.001	0.552	-0.052	0.003
60	0.5	0.434	0.066	0.004	0.623	-0.123	0.015

61	1.001	0.923	0.078	0.006	0.915	0.086	0.007
62	1.001	0.930	0.071	0.005	0.904	0.097	0.009
63	1.001	0.914	0.087	0.008	0.978	0.023	0.001
64	1.001	0.883	0.118	0.014	0.869	0.132	0.017
65	1.001	0.929	0.072	0.005	0.924	0.077	0.006
66	1.001	0.909	0.092	0.009	0.986	0.015	0.000
67	1.001	0.908	0.093	0.009	0.920	0.081	0.007
68	1.001	0.892	0.109	0.012	0.932	0.069	0.005
69	1.001	0.909	0.092	0.008	0.866	0.135	0.018
70	1.001	0.888	0.113	0.013	0.865	0.136	0.018
71	2.005	1.919	0.086	0.007	1.875	0.130	0.017
72	2.005	1.880	0.125	0.016	1.850	0.155	0.024
73	2.005	1.863	0.142	0.020	1.802	0.203	0.041
74	2.005	1.866	0.139	0.019	1.802	0.203	0.041
75	2.005	1.871	0.134	0.018	1.809	0.196	0.038
76	2.005	1.860	0.145	0.021	1.839	0.166	0.028
77	2.005	1.863	0.142	0.020	1.839	0.166	0.028
78	2.005	1.865	0.140	0.020	1.823	0.182	0.033
79	2.005	1.857	0.148	0.022	1.819	0.186	0.035
80	2.005	1.879	0.126	0.016	1.801	0.204	0.042
81	3.005	2.817	0.188	0.035	2.778	0.227	0.052
82	3.005	2.840	0.165	0.027	2.780	0.225	0.051
83	3.005	2.856	0.149	0.022	2.780	0.225	0.051
84	3.005	2.863	0.142	0.020	2.805	0.200	0.040
85	3.005	2.873	0.132	0.017	2.790	0.215	0.046
86	3.005	2.858	0.147	0.022	2.807	0.198	0.039
87	3.005	2.841	0.164	0.027	2.818	0.187	0.035
88	3.005	2.805	0.200	0.040	2.789	0.216	0.047
89	3.005	2.842	0.163	0.027	2.812	0.193	0.037
90	3.005	2.832	0.173	0.030	2.796	0.209	0.044
91	3.999	3.847	0.152	0.023	3.758	0.241	0.058
92	3.999	3.832	0.167	0.028	3.781	0.218	0.048

93	3.999	3.842	0.157	0.025	3.779	0.220	0.048
94	3.999	3.820	0.179	0.032	3.781	0.218	0.047
95	3.999	3.843	0.156	0.024	3.750	0.249	0.062
96	3.999	3.807	0.192	0.037	3.798	0.201	0.040
97	3.999	3.826	0.173	0.030	3.780	0.219	0.048
98	3.999	3.838	0.161	0.026	3.809	0.190	0.036
99	3.999	3.814	0.185	0.034	3.797	0.202	0.041
100	3.999	3.800	0.199	0.040	3.702	0.297	0.088
101	5	4.763	0.237	0.056	4.734	0.266	0.071
102	5	4.825	0.175	0.030	4.754	0.246	0.061
103	5	4.826	0.174	0.030	4.752	0.248	0.061
104	5	4.805	0.195	0.038	4.787	0.213	0.045
105	5	4.835	0.165	0.027	4.755	0.245	0.060
106	5	4.797	0.203	0.041	4.802	0.198	0.039
107	5	4.818	0.182	0.033	4.761	0.239	0.057
108	5	4.827	0.173	0.030	4.778	0.222	0.049
109	5	4.846	0.154	0.024	4.789	0.211	0.044
110	5	4.836	0.164	0.027	4.807	0.193	0.037
111	10	9.770	0.230	0.053	9.702	0.298	0.089
112	10	9.748	0.252	0.064	9.606	0.394	0.155
113	10	9.751	0.249	0.062	9.646	0.354	0.126
114	10	9.793	0.207	0.043	9.625	0.375	0.140
115	10	9.755	0.245	0.060	9.626	0.374	0.140
116	10	9.753	0.247	0.061	9.792	0.208	0.043
117	10	9.744	0.256	0.066	9.641	0.359	0.129
118	10	9.785	0.215	0.046	9.630	0.370	0.137
119	10	9.797	0.203	0.041	9.696	0.304	0.092
120	10	9.779	0.221	0.049	9.722	0.278	0.078
121	14.997	14.685	0.312	0.097	14.595	0.402	0.161
122	14.997	14.698	0.299	0.089	14.584	0.413	0.171
123	14.997	14.737	0.260	0.068	14.624	0.373	0.139
124	14.997	14.703	0.294	0.087	14.629	0.368	0.136

125	14.997	14.720	0.277	0.077	14.601	0.396	0.157
126	14.997	14.709	0.288	0.083	14.594	0.403	0.163
127	14.997	14.717	0.280	0.078	14.598	0.399	0.159
128	14.997	14.760	0.237	0.056	14.637	0.360	0.130
129	14.997	14.718	0.279	0.078	14.633	0.364	0.132
130	14.997	14.708	0.289	0.084	14.639	0.358	0.129
	Max	14.760	0.312	0.097	14.639	0.413	0.171
	Min	0.071	-0.137	3.40E-06	0.016	-0.184	7.02E-06
	Average	3.102	0.091	0.020	3.096	0.097	0.038
	Standard Deviation	4.276	0.110	0.024	4.226	0.169	0.046

CPU RMSE = 0.142880 mm DRRACC RMS Error = 0.194979 mm

First Metatarsal 6992R (run 1)

1st MET 6992R		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.002	0.000	0.002	0.000	0.000	0.002	0.000
2	0.002	0.054	-0.052	0.003	0.079	-0.077	0.006
3	0.002	0.013	-0.011	0.000	0.053	-0.051	0.003
4	0.002	0.059	-0.057	0.003	0.357	-0.355	0.126
5	0.002	0.023	-0.021	0.000	0.049	-0.047	0.002
6	0.002	0.037	-0.035	0.001	0.049	-0.047	0.002
7	0.002	0.031	-0.029	0.001	0.177	-0.175	0.031
8	0.002	0.009	-0.007	0.000	0.148	-0.146	0.021
9	0.002	0.039	-0.037	0.001	0.145	-0.143	0.021
10	0.002	0.021	-0.019	0.000	0.175	-0.173	0.030
11	0.102	0.159	-0.057	0.003	0.217	-0.115	0.013
12	0.102	0.106	-0.004	0.000	0.113	-0.011	0.000
13	0.102	0.139	-0.037	0.001	0.112	-0.010	0.000
14	0.102	0.138	-0.036	0.001	0.102	0.000	0.000
15	0.102	0.121	-0.019	0.000	0.187	-0.085	0.007
16	0.102	0.060	0.042	0.002	0.117	-0.015	0.000
17	0.102	0.128	-0.026	0.001	0.141	-0.039	0.002
18	0.102	0.090	0.012	0.000	0.185	-0.083	0.007
19	0.102	0.140	-0.038	0.001	0.089	0.013	0.000
20	0.102	0.090	0.012	0.000	0.182	-0.080	0.006
21	0.202	0.190	0.012	0.000	0.228	-0.026	0.001
22	0.202	0.219	-0.017	0.000	0.601	-0.399	0.159
23	0.202	0.195	0.007	0.000	0.192	0.010	0.000
24	0.202	0.198	0.004	0.000	0.341	-0.139	0.019
25	0.202	0.168	0.034	0.001	0.174	0.028	0.001
26	0.202	0.177	0.025	0.001	0.225	-0.023	0.001
27	0.202	0.181	0.021	0.000	0.204	-0.002	0.000
28	0.202	0.241	-0.039	0.002	0.175	0.027	0.001

29	0.202	0.245	-0.043	0.002	0.312	-0.110	0.012
30	0.202	0.233	-0.031	0.001	0.327	-0.125	0.016
31	0.302	0.315	-0.013	0.000	0.289	0.013	0.000
32	0.302	0.335	-0.033	0.001	0.376	-0.074	0.006
33	0.302	0.343	-0.041	0.002	0.374	-0.072	0.005
34	0.302	0.315	-0.013	0.000	0.382	-0.080	0.006
35	0.302	0.319	-0.017	0.000	0.407	-0.105	0.011
36	0.302	0.318	-0.016	0.000	0.431	-0.129	0.017
37	0.302	0.307	-0.005	0.000	0.265	0.037	0.001
38	0.302	0.324	-0.022	0.000	0.524	-0.222	0.049
39	0.302	0.305	-0.003	0.000	0.260	0.042	0.002
40	0.302	0.309	-0.007	0.000	0.338	-0.036	0.001
41	0.401	0.418	-0.017	0.000	0.430	-0.029	0.001
42	0.401	0.437	-0.036	0.001	0.417	-0.016	0.000
43	0.401	0.421	-0.020	0.000	0.411	-0.010	0.000
44	0.401	0.398	0.003	0.000	0.411	-0.010	0.000
45	0.401	0.405	-0.004	0.000	0.410	-0.009	0.000
46	0.401	0.373	0.028	0.001	0.411	-0.010	0.000
47	0.401	0.377	0.024	0.001	0.411	-0.010	0.000
48	0.401	0.383	0.018	0.000	0.403	-0.002	0.000
49	0.401	0.387	0.014	0.000	0.510	-0.109	0.012
50	0.401	0.426	-0.025	0.001	0.410	-0.009	0.000
51	0.500	0.488	0.012	0.000	0.515	-0.015	0.000
52	0.500	0.481	0.019	0.000	0.630	-0.130	0.017
53	0.500	0.480	0.020	0.000	0.566	-0.066	0.004
54	0.500	0.503	-0.003	0.000	0.435	0.065	0.004
55	0.500	0.499	0.001	0.000	0.461	0.039	0.002
56	0.500	0.522	-0.022	0.000	0.542	-0.042	0.002
57	0.500	0.495	0.005	0.000	0.540	-0.040	0.002
58	0.500	0.494	0.006	0.000	0.639	-0.139	0.019
59	0.500	0.485	0.015	0.000	0.510	-0.010	0.000
60	0.500	0.521	-0.021	0.000	0.498	0.002	0.000

61	0.995	1.006	-0.011	0.000	0.992	0.003	0.000
62	0.995	0.952	0.043	0.002	0.987	0.008	0.000
63	0.995	0.988	0.007	0.000	0.996	-0.001	0.000
64	0.995	0.989	0.006	0.000	0.995	0.000	0.000
65	0.995	0.980	0.015	0.000	0.994	0.001	0.000
66	0.995	0.966	0.029	0.001	0.991	0.004	0.000
67	0.995	0.979	0.016	0.000	1.002	-0.007	0.000
68	0.995	0.932	0.063	0.004	0.975	0.020	0.000
69	0.995	1.006	-0.011	0.000	1.040	-0.045	0.002
70	0.995	0.986	0.009	0.000	0.937	0.058	0.003
71	2.000	1.996	0.004	0.000	2.005	-0.005	0.000
72	2.000	2.031	-0.031	0.001	2.000	0.000	0.000
73	2.000	2.070	-0.070	0.005	1.985	0.015	0.000
74	2.000	2.029	-0.029	0.001	1.990	0.010	0.000
75	2.000	2.022	-0.022	0.000	2.003	-0.003	0.000
76	2.000	2.042	-0.042	0.002	1.977	0.023	0.001
77	2.000	2.029	-0.029	0.001	1.990	0.010	0.000
78	2.000	1.998	0.002	0.000	2.020	-0.020	0.000
79	2.000	2.017	-0.017	0.000	1.972	0.028	0.001
80	2.000	2.043	-0.043	0.002	1.972	0.028	0.001
81	3.003	2.996	0.007	0.000	2.960	0.043	0.002
82	3.003	3.000	0.003	0.000	2.877	0.126	0.016
83	3.003	3.013	-0.010	0.000	2.903	0.100	0.010
84	3.003	3.003	0.000	0.000	2.908	0.095	0.009
85	3.003	2.989	0.014	0.000	2.904	0.099	0.010
86	3.003	2.985	0.018	0.000	2.941	0.062	0.004
87	3.003	2.992	0.011	0.000	2.899	0.104	0.011
88	3.003	2.983	0.020	0.000	2.901	0.102	0.010
89	3.003	2.992	0.011	0.000	2.944	0.059	0.003
90	3.003	2.951	0.052	0.003	2.921	0.082	0.007
91	3.997	3.983	0.014	0.000	3.900	0.097	0.009
92	3.997	4.013	-0.016	0.000	3.897	0.100	0.010

93	3.997	3.958	0.039	0.002	3.879	0.118	0.014
94	3.997	3.982	0.015	0.000	3.880	0.117	0.014
95	3.997	3.950	0.047	0.002	3.839	0.158	0.025
96	3.997	3.999	-0.002	0.000	3.883	0.114	0.013
97	3.997	3.978	0.019	0.000	3.892	0.105	0.011
98	3.997	4.001	-0.004	0.000	3.903	0.094	0.009
99	3.997	3.984	0.013	0.000	3.910	0.087	0.008
100	3.997	3.955	0.042	0.002	3.905	0.092	0.008
101	4.999	4.991	0.008	0.000	4.889	0.110	0.012
102	4.999	5.024	-0.025	0.001	4.868	0.131	0.017
103	4.999	5.014	-0.015	0.000	4.882	0.117	0.014
104	4.999	4.991	0.008	0.000	4.918	0.081	0.007
105	4.999	5.016	-0.017	0.000	4.909	0.090	0.008
106	4.999	5.015	-0.016	0.000	4.893	0.106	0.011
107	4.999	5.001	-0.002	0.000	4.871	0.128	0.016
108	4.999	4.982	0.017	0.000	4.881	0.118	0.014
109	4.999	4.985	0.014	0.000	4.869	0.130	0.017
110	4.999	4.994	0.005	0.000	4.883	0.116	0.014
111	10.000	10.022	-0.022	0.000	9.827	0.173	0.030
112	10.000	9.963	0.037	0.001	9.776	0.224	0.050
113	10.000	9.961	0.039	0.002	9.811	0.189	0.036
114	10.000	9.958	0.042	0.002	9.859	0.141	0.020
115	10.000	9.961	0.039	0.002	9.863	0.137	0.019
116	10.000	9.958	0.042	0.002	9.788	0.212	0.045
117	10.000	9.974	0.026	0.001	9.784	0.216	0.047
118	10.000	9.990	0.010	0.000	9.816	0.184	0.034
119	10.000	9.991	0.009	0.000	9.775	0.225	0.050
120	10.000	9.992	0.008	0.000	9.774	0.226	0.051
121	15.003	14.980	0.023	0.001	14.822	0.181	0.033
122	15.003	14.943	0.060	0.004	14.809	0.194	0.038
123	15.003	14.970	0.033	0.001	14.743	0.260	0.068
124	15.003	14.939	0.064	0.004	14.787	0.216	0.047

125	15.003	14.967	0.036	0.001	14.846	0.157	0.025
126	15.003	14.967	0.036	0.001	14.826	0.177	0.031
127	15.003	14.947	0.056	0.003	14.799	0.204	0.042
128	15.003	14.968	0.035	0.001	14.816	0.187	0.035
129	15.003	14.938	0.065	0.004	14.802	0.201	0.040
130	15.003	14.970	0.033	0.001	14.794	0.209	0.044
	Max	**14.980**	**0.065**	**0.005**	**14.846**	**0.260**	**0.159**
	Min	**0.009**	**-0.070**	**1.26E-07**	**0.049**	**-0.399**	**1.99E-09**
	Average	**3.191**	**0.002**	**0.001**	**3.165**	**0.027**	**0.014**
	Standard Deviation	**4.353**	**0.028**	**0.001**	**4.281**	**0.113**	**0.022**

CPU RMSE = 0.028030 mm **DRRACC RMSE = 0.116735 mm**

First Metatarsal 6992R (run 2)

1st MET 6992R		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.002	0.000	0.000	0.000	0.000	0.000	0.000
2	0.002	0.054	-0.054	0.003	0.039	-0.039	0.002
3	0.002	0.013	-0.013	0.000	0.037	-0.037	0.001
4	0.002	0.059	-0.059	0.004	0.054	-0.054	0.003
5	0.002	0.023	-0.023	0.001	0.074	-0.074	0.006
6	0.002	0.037	-0.037	0.001	0.071	-0.071	0.005
7	0.002	0.031	-0.031	0.001	0.269	-0.269	0.072
8	0.002	0.009	-0.009	0.000	0.097	-0.097	0.009
9	0.002	0.039	-0.039	0.001	0.033	-0.033	0.001
10	0.002	0.021	-0.021	0.000	0.096	-0.096	0.009
11	0.102	0.159	-0.059	0.003	0.151	-0.051	0.003
12	0.102	0.106	-0.006	0.000	0.147	-0.047	0.002
13	0.102	0.139	-0.039	0.002	0.154	-0.054	0.003
14	0.102	0.138	-0.038	0.001	0.162	-0.062	0.004
15	0.102	0.121	-0.021	0.000	0.135	-0.035	0.001
16	0.102	0.060	0.040	0.002	0.104	-0.004	0.000
17	0.102	0.128	-0.028	0.001	0.119	-0.019	0.000
18	0.102	0.090	0.010	0.000	0.122	-0.022	0.000
19	0.102	0.140	-0.040	0.002	0.130	-0.030	0.001
20	0.102	0.090	0.010	0.000	0.137	-0.037	0.001
21	0.202	0.190	0.010	0.000	0.214	-0.014	0.000
22	0.202	0.219	-0.019	0.000	0.178	0.022	0.000
23	0.202	0.195	0.005	0.000	0.177	0.023	0.001
24	0.202	0.198	0.002	0.000	0.204	-0.004	0.000
25	0.202	0.168	0.032	0.001	0.285	-0.085	0.007
26	0.202	0.177	0.023	0.001	0.274	-0.074	0.006
27	0.202	0.181	0.019	0.000	0.222	-0.022	0.000
28	0.202	0.241	-0.041	0.002	0.209	-0.009	0.000

29	0.202	0.245	-0.045	0.002	0.252	-0.052	0.003
30	0.202	0.233	-0.033	0.001	0.217	-0.017	0.000
31	0.302	0.315	-0.015	0.000	0.312	-0.012	0.000
32	0.302	0.335	-0.035	0.001	0.356	-0.056	0.003
33	0.302	0.343	-0.043	0.002	0.300	0.000	0.000
34	0.302	0.315	-0.015	0.000	0.312	-0.012	0.000
35	0.302	0.319	-0.019	0.000	0.303	-0.003	0.000
36	0.302	0.318	-0.018	0.000	0.331	-0.031	0.001
37	0.302	0.307	-0.007	0.000	0.289	0.011	0.000
38	0.302	0.324	-0.024	0.001	0.290	0.010	0.000
39	0.302	0.305	-0.005	0.000	0.282	0.018	0.000
40	0.302	0.309	-0.009	0.000	0.289	0.011	0.000
41	0.401	0.418	-0.019	0.000	0.395	0.004	0.000
42	0.401	0.437	-0.038	0.001	0.385	0.014	0.000
43	0.401	0.421	-0.022	0.000	0.388	0.011	0.000
44	0.401	0.398	0.001	0.000	0.401	-0.002	0.000
45	0.401	0.405	-0.006	0.000	0.401	-0.002	0.000
46	0.401	0.373	0.026	0.001	0.400	-0.001	0.000
47	0.401	0.377	0.022	0.000	0.402	-0.003	0.000
48	0.401	0.383	0.016	0.000	0.385	0.014	0.000
49	0.401	0.387	0.012	0.000	0.429	-0.030	0.001
50	0.401	0.426	-0.027	0.001	0.427	-0.028	0.001
51	0.5	0.488	0.010	0.000	0.492	0.006	0.000
52	0.5	0.481	0.017	0.000	0.518	-0.020	0.000
53	0.5	0.480	0.018	0.000	0.505	-0.007	0.000
54	0.5	0.503	-0.005	0.000	0.510	-0.012	0.000
55	0.5	0.499	-0.001	0.000	0.538	-0.040	0.002
56	0.5	0.522	-0.024	0.001	0.469	0.029	0.001
57	0.5	0.495	0.003	0.000	0.466	0.032	0.001
58	0.5	0.494	0.004	0.000	0.488	0.010	0.000
59	0.5	0.485	0.013	0.000	0.528	-0.030	0.001
60	0.5	0.521	-0.023	0.001	0.513	-0.015	0.000

61	0.995	1.006	-0.013	0.000	1.034	-0.041	0.002
62	0.995	0.952	0.041	0.002	1.023	-0.030	0.001
63	0.995	0.988	0.005	0.000	1.005	-0.012	0.000
64	0.995	0.989	0.004	0.000	1.003	-0.010	0.000
65	0.995	0.980	0.013	0.000	1.017	-0.024	0.001
66	0.995	0.966	0.027	0.001	0.999	-0.006	0.000
67	0.995	0.979	0.014	0.000	0.965	0.028	0.001
68	0.995	0.932	0.061	0.004	0.981	0.012	0.000
69	0.995	1.006	-0.013	0.000	1.027	-0.034	0.001
70	0.995	0.986	0.007	0.000	1.010	-0.017	0.000
71	2	1.996	0.002	0.000	1.990	0.008	0.000
72	2	2.031	-0.033	0.001	1.995	0.003	0.000
73	2	2.070	-0.072	0.005	2.009	-0.011	0.000
74	2	2.029	-0.031	0.001	2.024	-0.026	0.001
75	2	2.022	-0.024	0.001	1.985	0.013	0.000
76	2	2.042	-0.044	0.002	2.009	-0.011	0.000
77	2	2.029	-0.031	0.001	2.011	-0.013	0.000
78	2	1.998	0.000	0.000	1.984	0.014	0.000
79	2	2.017	-0.019	0.000	2.003	-0.005	0.000
80	2	2.043	-0.045	0.002	2.003	-0.005	0.000
81	3.003	2.996	0.005	0.000	3.029	-0.028	0.001
82	3.003	3.000	0.001	0.000	2.951	0.050	0.003
83	3.003	3.013	-0.012	0.000	2.988	0.013	0.000
84	3.003	3.003	-0.002	0.000	2.990	0.011	0.000
85	3.003	2.989	0.012	0.000	2.989	0.012	0.000
86	3.003	2.985	0.016	0.000	2.989	0.012	0.000
87	3.003	2.992	0.009	0.000	2.971	0.030	0.001
88	3.003	2.983	0.018	0.000	2.990	0.011	0.000
89	3.003	2.992	0.009	0.000	3.010	-0.009	0.000
90	3.003	2.951	0.050	0.002	3.013	-0.012	0.000
91	3.997	3.983	0.012	0.000	3.975	0.020	0.000
92	3.997	4.013	-0.018	0.000	3.981	0.014	0.000

93	3.997	3.958	0.037	0.001	3.970	0.025	0.001
94	3.997	3.982	0.013	0.000	3.998	-0.003	0.000
95	3.997	3.950	0.045	0.002	3.971	0.024	0.001
96	3.997	3.999	-0.004	0.000	3.978	0.017	0.000
97	3.997	3.978	0.017	0.000	3.978	0.017	0.000
98	3.997	4.001	-0.006	0.000	3.978	0.017	0.000
99	3.997	3.984	0.011	0.000	4.012	-0.017	0.000
100	3.997	3.955	0.040	0.002	3.970	0.025	0.001
101	4.999	4.991	0.006	0.000	4.997	0.000	0.000
102	4.999	5.024	-0.027	0.001	4.976	0.021	0.000
103	4.999	5.014	-0.017	0.000	4.974	0.023	0.001
104	4.999	4.991	0.006	0.000	4.976	0.021	0.000
105	4.999	5.016	-0.019	0.000	5.010	-0.013	0.000
106	4.999	5.015	-0.018	0.000	5.012	-0.015	0.000
107	4.999	5.001	-0.004	0.000	4.996	0.001	0.000
108	4.999	4.982	0.015	0.000	4.993	0.004	0.000
109	4.999	4.985	0.012	0.000	4.976	0.021	0.000
110	4.999	4.994	0.003	0.000	4.974	0.023	0.001
111	10	10.022	-0.024	0.001	9.951	0.047	0.002
112	10	9.963	0.035	0.001	9.915	0.083	0.007
113	10	9.961	0.037	0.001	9.928	0.070	0.005
114	10	9.958	0.040	0.002	9.950	0.048	0.002
115	10	9.961	0.037	0.001	9.962	0.036	0.001
116	10	9.958	0.040	0.002	9.943	0.055	0.003
117	10	9.974	0.024	0.001	9.904	0.094	0.009
118	10	9.990	0.008	0.000	9.929	0.069	0.005
119	10	9.991	0.007	0.000	9.931	0.067	0.004
120	10	9.992	0.006	0.000	9.928	0.070	0.005
121	15.003	14.980	0.021	0.000	14.914	0.087	0.008
122	15.003	14.943	0.058	0.003	14.883	0.118	0.014
123	15.003	14.970	0.031	0.001	14.852	0.149	0.022
124	15.003	14.939	0.062	0.004	14.888	0.113	0.013

125	15.003	14.967	0.034	0.001	14.916	0.085	0.007
126	15.003	14.967	0.034	0.001	14.914	0.087	0.008
127	15.003	14.947	0.054	0.003	14.884	0.117	0.014
128	15.003	14.968	0.033	0.001	14.876	0.125	0.016
129	15.003	14.938	0.063	0.004	14.893	0.108	0.012
130	15.003	14.970	0.031	0.001	14.881	0.120	0.014
	Max	**14.980**	**0.063**	**0.005**	**14.916**	**0.149**	**0.072**
	Min	**0.009**	**-0.072**	**8.60E-08**	**0.033**	**-0.269**	**1.55E-08**
	Average	**3.191**	**0.000**	**0.001**	**3.188**	**0.003**	**0.003**
	Standard Deviation	**4.353**	**0.028**	**0.001**	**4.329**	**0.051**	**0.007**

CPU RMSE = 0.027973 mm **DRRACC RMSE = 0.0513721 mm**

1st Metatarsal 7451

1st MET 7451		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.004	0.000	0.000	0.000	0.000	0.000	0.000
2	0.004	0.000	0.000	0.000	0.019	-0.019	0.000
3	0.004	0.012	-0.012	0.000	0.048	-0.048	0.002
4	0.004	0.030	-0.030	0.001	0.090	-0.090	0.008
5	0.004	0.038	-0.038	0.001	0.044	-0.044	0.002
6	0.004	0.019	-0.019	0.000	0.042	-0.042	0.002
7	0.004	0.052	-0.052	0.003	0.044	-0.044	0.002
8	0.004	0.081	-0.081	0.007	0.054	-0.054	0.003
9	0.004	0.070	-0.070	0.005	0.107	-0.107	0.011
10	0.004	0.038	-0.038	0.001	0.064	-0.064	0.004
11	0.097	0.094	-0.001	0.000	0.094	-0.001	0.000
12	0.097	0.096	-0.003	0.000	0.081	0.012	0.000
13	0.097	0.150	-0.057	0.003	0.098	-0.005	0.000
14	0.097	0.144	-0.051	0.003	0.103	-0.010	0.000
15	0.097	0.130	-0.037	0.001	0.117	-0.024	0.001
16	0.097	0.095	-0.002	0.000	0.092	0.001	0.000
17	0.097	0.108	-0.015	0.000	0.120	-0.027	0.001
18	0.097	0.107	-0.014	0.000	0.129	-0.036	0.001
19	0.097	0.123	-0.030	0.001	0.122	-0.029	0.001
20	0.097	0.108	-0.015	0.000	0.064	0.029	0.001
21	0.197	0.196	-0.003	0.000	0.210	-0.017	0.000
22	0.197	0.224	-0.031	0.001	0.240	-0.047	0.002
23	0.197	0.219	-0.026	0.001	0.204	-0.011	0.000
24	0.197	0.221	-0.028	0.001	0.205	-0.012	0.000
25	0.197	0.225	-0.032	0.001	0.219	-0.026	0.001
26	0.197	0.231	-0.038	0.001	0.199	-0.006	0.000
27	0.197	0.216	-0.023	0.001	0.222	-0.029	0.001
28	0.197	0.229	-0.036	0.001	0.200	-0.007	0.000

29	0.197	0.253	-0.060	0.004	0.239	-0.046	0.002
30	0.197	0.235	-0.042	0.002	0.189	0.004	0.000
31	0.298	0.370	-0.076	0.006	0.389	-0.095	0.009
32	0.298	0.364	-0.070	0.005	0.378	-0.084	0.007
33	0.298	0.411	-0.117	0.014	0.369	-0.075	0.006
34	0.298	0.379	-0.085	0.007	0.372	-0.078	0.006
35	0.298	0.385	-0.091	0.008	0.372	-0.078	0.006
36	0.298	0.386	-0.092	0.009	0.392	-0.098	0.010
37	0.298	0.376	-0.082	0.007	0.379	-0.085	0.007
38	0.298	0.346	-0.052	0.003	0.368	-0.074	0.005
39	0.298	0.372	-0.078	0.006	0.381	-0.087	0.008
40	0.298	0.397	-0.103	0.011	0.371	-0.077	0.006
41	0.397	0.372	0.021	0.000	0.391	0.002	0.000
42	0.397	0.387	0.006	0.000	0.390	0.003	0.000
43	0.397	0.391	0.002	0.000	0.356	0.037	0.001
44	0.397	0.387	0.006	0.000	0.357	0.036	0.001
45	0.397	0.376	0.017	0.000	0.379	0.014	0.000
46	0.397	0.377	0.016	0.000	0.400	-0.007	0.000
47	0.397	0.394	-0.001	0.000	0.376	0.017	0.000
48	0.397	0.311	0.082	0.007	0.342	0.051	0.003
49	0.397	0.471	-0.078	0.006	0.361	0.032	0.001
50	0.397	0.406	-0.013	0.000	0.359	0.034	0.001
51	0.497	0.467	0.026	0.001	0.482	0.011	0.000
52	0.497	0.503	-0.010	0.000	0.498	-0.005	0.000
53	0.497	0.508	-0.015	0.000	0.489	0.004	0.000
54	0.497	0.517	-0.024	0.001	0.524	-0.031	0.001
55	0.497	0.510	-0.017	0.000	0.505	-0.012	0.000
56	0.497	0.495	-0.002	0.000	0.471	0.022	0.000
57	0.497	0.514	-0.021	0.000	0.478	0.015	0.000
58	0.497	0.478	0.015	0.000	0.500	-0.007	0.000
59	0.497	0.495	-0.002	0.000	0.487	0.006	0.000
60	0.497	0.557	-0.064	0.004	0.528	-0.035	0.001

61	0.996	0.954	0.038	0.001	0.968	0.024	0.001
62	0.996	0.960	0.032	0.001	0.964	0.028	0.001
63	0.996	0.963	0.029	0.001	0.985	0.007	0.000
64	0.996	0.984	0.008	0.000	0.984	0.008	0.000
65	0.996	0.968	0.024	0.001	0.976	0.016	0.000
66	0.996	0.947	0.045	0.002	0.955	0.037	0.001
67	0.996	0.953	0.039	0.002	0.979	0.013	0.000
68	0.996	0.983	0.009	0.000	0.987	0.005	0.000
69	0.996	0.963	0.029	0.001	0.962	0.030	0.001
70	0.996	0.992	0.000	0.000	0.987	0.005	0.000
71	2	1.989	0.007	0.000	2.045	-0.049	0.002
72	2	1.978	0.018	0.000	1.998	-0.002	0.000
73	2	2.008	-0.012	0.000	1.978	0.018	0.000
74	2	1.967	0.029	0.001	1.984	0.012	0.000
75	2	2.023	-0.027	0.001	1.984	0.012	0.000
76	2	2.004	-0.008	0.000	2.001	-0.005	0.000
77	2	2.001	-0.005	0.000	1.995	0.001	0.000
78	2	1.980	0.016	0.000	1.987	0.009	0.000
79	2	2.001	-0.005	0.000	1.967	0.029	0.001
80	2	1.987	0.009	0.000	1.988	0.008	0.000
81	3.000	2.989	0.007	0.000	2.978	0.018	0.000
82	3.000	3.069	-0.073	0.005	2.963	0.033	0.001
83	3.000	2.993	0.003	0.000	2.965	0.031	0.001
84	3.000	3.033	-0.037	0.001	2.983	0.013	0.000
85	3.000	3.021	-0.025	0.001	2.929	0.067	0.004
86	3.000	3.038	-0.042	0.002	2.974	0.022	0.000
87	3.000	2.984	0.012	0.000	2.951	0.045	0.002
88	3.000	3.006	-0.010	0.000	2.930	0.066	0.004
89	3.000	2.972	0.024	0.001	2.965	0.031	0.001
90	3.000	2.981	0.015	0.000	2.964	0.032	0.001
91	4.004	3.983	0.017	0.000	3.952	0.048	0.002
92	4.004	3.990	0.010	0.000	3.936	0.064	0.004

93	4.004	3.993	0.007	0.000	3.939	0.061	0.004
94	4.004	4.006	-0.006	0.000	3.955	0.045	0.002
95	4.004	4.025	-0.025	0.001	3.924	0.076	0.006
96	4.004	4.035	-0.035	0.001	3.983	0.017	0.000
97	4.004	4.008	-0.008	0.000	3.928	0.072	0.005
98	4.004	3.930	0.070	0.005	3.963	0.037	0.001
99	4.004	4.020	-0.020	0.000	3.944	0.056	0.003
100	4.004	4.083	-0.083	0.007	3.993	0.007	0.000
101	4.999	4.970	0.025	0.001	4.983	0.012	0.000
102	4.999	4.964	0.031	0.001	4.968	0.027	0.001
103	4.999	4.993	0.002	0.000	5.008	-0.013	0.000
104	4.999	4.965	0.030	0.001	4.971	0.024	0.001
105	4.999	4.974	0.021	0.000	4.969	0.026	0.001
106	4.999	4.964	0.031	0.001	4.955	0.040	0.002
107	4.999	4.939	0.056	0.003	4.956	0.039	0.001
108	4.999	4.992	0.003	0.000	4.954	0.041	0.002
109	4.999	5.002	-0.007	0.000	4.966	0.029	0.001
110	4.999	4.996	-0.001	0.000	4.978	0.017	0.000
111	9.997	10.010	-0.017	0.000	9.946	0.047	0.002
112	9.997	9.978	0.015	0.000	9.947	0.046	0.002
113	9.997	10.009	-0.016	0.000	9.939	0.054	0.003
114	9.997	9.977	0.016	0.000	9.964	0.029	0.001
115	9.997	9.965	0.028	0.001	9.900	0.093	0.009
116	9.997	9.964	0.029	0.001	9.911	0.082	0.007
117	9.997	9.955	0.038	0.001	9.924	0.069	0.005
118	9.997	9.955	0.038	0.001	9.981	0.012	0.000
119	9.997	9.963	0.030	0.001	9.949	0.044	0.002
120	9.997	9.963	0.030	0.001	9.924	0.069	0.005
121	14.997	15.027	-0.034	0.001	14.947	0.046	0.002
122	14.997	14.999	-0.006	0.000	14.956	0.037	0.001
123	14.997	14.960	0.033	0.001	14.947	0.046	0.002
124	14.997	14.980	0.013	0.000	14.958	0.035	0.001

125	14.997	14.979	0.014	0.000	14.905	0.088	0.008
126	14.997	14.967	0.026	0.001	14.905	0.088	0.008
127	14.997	14.996	-0.003	0.000	14.955	0.038	0.001
128	14.997	14.935	0.058	0.003	14.944	0.049	0.002
129	14.997	14.970	0.023	0.001	14.944	0.049	0.002
130	14.997	14.994	-0.001	0.000	14.960	0.033	0.001
	Max	**15.027**	**0.082**	**0.014**	**14.960**	**0.093**	**0.011**
	Min	**0.000**	**-0.117**	**1.97E-07**	**0.019**	**-0.107**	**2.67E-07**
	Average	**3.196**	**-0.009**	**0.001**	**3.180**	**0.007**	**0.002**
	Standard Deviation	**4.354**	**0.037**	**0.002**	**4.343**	**0.043**	**0.002**

CPU RMSE = 0.038422 mm **DRRACC RMSE = 0.044050 mm**

Talus 7451

TALUS 7451		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.003	0.000	0.000	0.000	0.000	0.000	0.000
2	0.003	0.078	-0.078	0.006	0.030	-0.030	0.001
3	0.003	0.058	-0.058	0.003	0.086	-0.086	0.007
4	0.003	0.055	-0.055	0.003	0.086	-0.086	0.007
5	0.003	0.070	-0.070	0.005	0.114	-0.114	0.013
6	0.003	0.031	-0.031	0.001	0.058	-0.058	0.003
7	0.003	0.057	-0.057	0.003	0.138	-0.138	0.019
8	0.003	0.021	-0.021	0.000	0.034	-0.034	0.001
9	0.003	0.062	-0.062	0.004	0.093	-0.093	0.009
10	0.003	0.063	-0.063	0.004	0.088	-0.088	0.008
11	0.099	0.113	-0.017	0.000	0.180	-0.084	0.007
12	0.099	0.123	-0.027	0.001	0.175	-0.079	0.006
13	0.099	0.115	-0.019	0.000	0.153	-0.057	0.003
14	0.099	0.102	-0.006	0.000	0.131	-0.035	0.001
15	0.099	0.118	-0.022	0.000	0.165	-0.069	0.005
16	0.099	0.092	0.004	0.000	0.104	-0.008	0.000
17	0.099	0.076	0.020	0.000	0.190	-0.094	0.009
18	0.099	0.099	-0.003	0.000	0.165	-0.069	0.005
19	0.099	0.128	-0.032	0.001	0.229	-0.133	0.018
20	0.099	0.087	0.009	0.000	0.125	-0.029	0.001
21	0.198	0.194	0.001	0.000	0.239	-0.044	0.002
22	0.198	0.195	0.000	0.000	0.231	-0.036	0.001
23	0.198	0.202	-0.007	0.000	0.293	-0.098	0.010
24	0.198	0.172	0.023	0.001	0.215	-0.020	0.000
25	0.198	0.178	0.017	0.000	0.276	-0.081	0.007
26	0.198	0.212	-0.017	0.000	0.317	-0.122	0.015
27	0.198	0.222	-0.027	0.001	0.227	-0.032	0.001
28	0.198	0.240	-0.045	0.002	0.230	-0.035	0.001

29	0.198	0.187	0.008	0.000	0.232	-0.037	0.001
30	0.198	0.190	0.005	0.000	0.243	-0.048	0.002
31	0.298	0.273	0.022	0.000	0.346	-0.051	0.003
32	0.298	0.286	0.009	0.000	0.376	-0.081	0.007
33	0.298	0.308	-0.013	0.000	0.327	-0.032	0.001
34	0.298	0.287	0.008	0.000	0.372	-0.077	0.006
35	0.298	0.286	0.009	0.000	0.324	-0.029	0.001
36	0.298	0.310	-0.015	0.000	0.346	-0.051	0.003
37	0.298	0.302	-0.007	0.000	0.346	-0.051	0.003
38	0.298	0.272	0.023	0.001	0.356	-0.061	0.004
39	0.298	0.297	-0.002	0.000	0.342	-0.047	0.002
40	0.298	0.299	-0.004	0.000	0.355	-0.060	0.004
41	0.394	0.393	-0.002	0.000	0.457	-0.066	0.004
42	0.394	0.429	-0.038	0.001	0.482	-0.091	0.008
43	0.394	0.382	0.009	0.000	0.480	-0.089	0.008
44	0.394	0.408	-0.017	0.000	0.462	-0.071	0.005
45	0.394	0.338	0.053	0.003	0.428	-0.037	0.001
46	0.394	0.398	-0.007	0.000	0.454	-0.063	0.004
47	0.394	0.376	0.015	0.000	0.478	-0.087	0.008
48	0.394	0.366	0.025	0.001	0.478	-0.087	0.008
49	0.394	0.396	-0.005	0.000	0.481	-0.090	0.008
50	0.394	0.368	0.023	0.001	0.386	0.005	0.000
51	0.503	0.508	-0.008	0.000	0.501	-0.001	0.000
52	0.503	0.482	0.018	0.000	0.520	-0.020	0.000
53	0.503	0.512	-0.012	0.000	0.492	0.008	0.000
54	0.503	0.500	0.000	0.000	0.495	0.005	0.000
55	0.503	0.495	0.005	0.000	0.502	-0.002	0.000
56	0.503	0.497	0.003	0.000	0.504	-0.004	0.000
57	0.503	0.476	0.024	0.001	0.505	-0.005	0.000
58	0.503	0.509	-0.009	0.000	0.499	0.001	0.000
59	0.503	0.472	0.028	0.001	0.615	-0.115	0.013
60	0.503	0.495	0.005	0.000	0.529	-0.029	0.001

61	1.005	0.985	0.017	0.000	0.960	0.042	0.002
62	1.005	1.027	-0.025	0.001	0.979	0.023	0.001
63	1.005	0.980	0.022	0.000	0.985	0.017	0.000
64	1.005	0.977	0.025	0.001	1.037	-0.035	0.001
65	1.005	1.018	-0.016	0.000	0.943	0.059	0.003
66	1.005	0.994	0.008	0.000	1.025	-0.023	0.001
67	1.005	0.996	0.006	0.000	0.939	0.063	0.004
68	1.005	0.999	0.003	0.000	1.046	-0.044	0.002
69	1.005	0.996	0.006	0.000	1.031	-0.029	0.001
70	1.005	0.991	0.011	0.000	1.046	-0.044	0.002
71	1.998	1.963	0.032	0.001	1.966	0.029	0.001
72	1.998	1.967	0.028	0.001	1.973	0.022	0.000
73	1.998	1.955	0.040	0.002	1.966	0.029	0.001
74	1.998	1.960	0.035	0.001	1.943	0.052	0.003
75	1.998	1.954	0.041	0.002	1.962	0.033	0.001
76	1.998	1.933	0.062	0.004	1.965	0.030	0.001
77	1.998	1.992	0.003	0.000	1.965	0.030	0.001
78	1.998	1.955	0.040	0.002	1.963	0.032	0.001
79	1.998	1.967	0.028	0.001	1.988	0.007	0.000
80	1.998	1.982	0.013	0.000	1.981	0.014	0.000
81	2.993	2.950	0.040	0.002	2.878	0.112	0.013
82	2.993	2.940	0.050	0.003	2.924	0.066	0.004
83	2.993	2.945	0.045	0.002	2.968	0.022	0.001
84	2.993	2.966	0.024	0.001	2.924	0.066	0.004
85	2.993	2.932	0.058	0.003	2.926	0.064	0.004
86	2.993	2.958	0.032	0.001	2.930	0.060	0.004
87	2.993	2.969	0.021	0.000	2.931	0.059	0.004
88	2.993	2.950	0.040	0.002	2.940	0.050	0.002
89	2.993	2.949	0.041	0.002	2.940	0.050	0.003
90	2.993	2.948	0.042	0.002	2.937	0.053	0.003
91	3.997	3.980	0.014	0.000	3.900	0.094	0.009
92	3.997	3.934	0.060	0.004	3.901	0.093	0.009

93	3.997	3.945	0.049	0.002	3.931	0.063	0.004
94	3.997	3.938	0.056	0.003	3.977	0.017	0.000
95	3.997	3.962	0.032	0.001	3.965	0.029	0.001
96	3.997	3.955	0.039	0.002	3.966	0.028	0.001
97	3.997	3.962	0.032	0.001	3.962	0.032	0.001
98	3.997	3.943	0.051	0.003	3.990	0.004	0.000
99	3.997	3.969	0.025	0.001	3.977	0.017	0.000
100	3.997	3.951	0.043	0.002	3.976	0.018	0.000
101	4.993	4.956	0.034	0.001	4.891	0.099	0.010
102	4.993	4.961	0.029	0.001	4.951	0.039	0.001
103	4.993	4.945	0.045	0.002	4.923	0.067	0.005
104	4.993	4.957	0.033	0.001	4.918	0.072	0.005
105	4.993	4.959	0.031	0.001	4.916	0.074	0.006
106	4.993	4.964	0.026	0.001	4.943	0.047	0.002
107	4.993	4.977	0.013	0.000	4.959	0.031	0.001
108	4.993	4.977	0.013	0.000	4.895	0.095	0.009
109	4.993	4.958	0.032	0.001	4.964	0.026	0.001
110	4.993	4.935	0.055	0.003	4.942	0.048	0.002
111	10.003	9.934	0.066	0.004	9.891	0.109	0.012
112	10.003	9.937	0.063	0.004	9.865	0.135	0.018
113	10.003	9.959	0.041	0.002	9.966	0.034	0.001
114	10.003	9.973	0.027	0.001	9.900	0.100	0.010
115	10.003	9.963	0.037	0.001	9.967	0.033	0.001
116	10.003	9.938	0.062	0.004	10.002	-0.002	0.000
117	10.003	9.925	0.075	0.006	10.001	-0.001	0.000
118	10.003	9.954	0.046	0.002	9.898	0.102	0.010
119	10.003	9.959	0.041	0.002	9.984	0.016	0.000
120	10.003	9.955	0.045	0.002	9.912	0.088	0.008
121	15.003	14.884	0.116	0.013	14.854	0.146	0.021
122	15.003	14.819	0.181	0.033	14.831	0.169	0.029
123	15.003	14.884	0.116	0.013	14.849	0.151	0.023
124	15.003	14.909	0.091	0.008	14.862	0.138	0.019

125	15.003	14.900	0.100	0.010	14.862	0.138	0.019
126	15.003	14.913	0.087	0.008	14.867	0.133	0.018
127	15.003	14.897	0.103	0.011	14.835	0.165	0.027
128	15.003	14.887	0.113	0.013	14.867	0.133	0.018
129	15.003	14.897	0.103	0.011	14.867	0.133	0.018
130	15.003	14.864	0.136	0.019	14.874	0.126	0.016
	Max	**14.913**	**0.181**	**0.033**	**14.874**	**0.169**	**0.029**
	Min	**0.021**	**-0.078**	**3.30E-09**	**0.030**	**-0.138**	**6.25E-07**
	Average	**3.168**	**0.020**	**0.002**	**3.183**	**0.005**	**0.005**
	Standard Deviation	**4.336**	**0.041**	**0.004**	**4.313**	**0.071**	**0.006**

CPU RMSE = 0.045905 mm **DRRACC RMSE = 0.071553 mm**

Talus 7481

TALUS 7481		***CPU***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.003	0.000	0.000	0.000	0.000	0.000	0.000
2	0.003	0.079	-0.079	0.006	0.115	-0.115	0.013
3	0.003	0.044	-0.044	0.002	0.107	-0.107	0.011
4	0.003	0.091	-0.091	0.008	0.060	-0.060	0.004
5	0.003	0.066	-0.066	0.004	0.025	-0.025	0.001
6	0.003	0.064	-0.064	0.004	0.144	-0.144	0.021
7	0.003	0.032	-0.032	0.001	0.066	-0.066	0.004
8	0.003	0.026	-0.026	0.001	0.037	-0.037	0.001
9	0.003	0.042	-0.042	0.002	0.114	-0.114	0.013
10	0.003	0.037	-0.037	0.001	0.184	-0.184	0.034
11	0.102	0.156	-0.057	0.003	0.327	-0.228	0.052
12	0.102	0.090	0.009	0.000	0.111	-0.012	0.000
13	0.102	0.093	0.006	0.000	0.113	-0.014	0.000
14	0.102	0.091	0.008	0.000	0.251	-0.152	0.023
15	0.102	0.081	0.018	0.000	0.102	-0.003	0.000
16	0.102	0.119	-0.020	0.000	0.103	-0.004	0.000
17	0.102	0.072	0.027	0.001	0.109	-0.010	0.000
18	0.102	0.080	0.019	0.000	0.114	-0.015	0.000
19	0.102	0.083	0.016	0.000	0.087	0.012	0.000
20	0.102	0.074	0.025	0.001	0.082	0.017	0.000
21	0.202	0.185	0.014	0.000	0.208	-0.009	0.000
22	0.202	0.201	-0.002	0.000	0.223	-0.024	0.001
23	0.202	0.194	0.005	0.000	0.221	-0.022	0.000
24	0.202	0.194	0.005	0.000	0.226	-0.027	0.001
25	0.202	0.177	0.022	0.000	0.197	0.002	0.000
26	0.202	0.196	0.003	0.000	0.197	0.002	0.000
27	0.202	0.191	0.008	0.000	0.197	0.002	0.000
28	0.202	0.187	0.012	0.000	0.197	0.002	0.000

29	0.202	0.178	0.021	0.000	0.244	-0.045	0.002
30	0.202	0.197	0.002	0.000	0.289	-0.090	0.008
31	0.301	0.263	0.035	0.001	0.221	0.077	0.006
32	0.301	0.295	0.003	0.000	0.239	0.059	0.003
33	0.301	0.265	0.033	0.001	0.276	0.022	0.000
34	0.301	0.257	0.041	0.002	0.278	0.020	0.000
35	0.301	0.287	0.011	0.000	0.276	0.022	0.000
36	0.301	0.279	0.019	0.000	0.314	-0.016	0.000
37	0.301	0.294	0.004	0.000	0.331	-0.033	0.001
38	0.301	0.250	0.048	0.002	0.286	0.012	0.000
39	0.301	0.267	0.031	0.001	0.272	0.026	0.001
40	0.301	0.249	0.049	0.002	0.289	0.009	0.000
41	0.4	0.362	0.035	0.001	0.445	-0.048	0.002
42	0.4	0.377	0.020	0.000	0.384	0.013	0.000
43	0.4	0.380	0.017	0.000	0.494	-0.097	0.009
44	0.4	0.361	0.036	0.001	0.483	-0.086	0.007
45	0.4	0.396	0.001	0.000	0.412	-0.015	0.000
46	0.4	0.390	0.007	0.000	0.603	-0.206	0.042
47	0.4	0.351	0.046	0.002	0.330	0.067	0.004
48	0.4	0.364	0.033	0.001	0.478	-0.081	0.006
49	0.4	0.367	0.030	0.001	0.455	-0.058	0.003
50	0.4	0.365	0.032	0.001	0.436	-0.039	0.002
51	0.504	0.471	0.030	0.001	0.640	-0.139	0.019
52	0.504	0.469	0.032	0.001	0.468	0.033	0.001
53	0.504	0.486	0.015	0.000	0.544	-0.043	0.002
54	0.504	0.492	0.009	0.000	0.603	-0.102	0.011
55	0.504	0.494	0.007	0.000	0.502	-0.001	0.000
56	0.504	0.500	0.001	0.000	0.521	-0.020	0.000
57	0.504	0.459	0.042	0.002	0.514	-0.013	0.000
58	0.504	0.483	0.018	0.000	0.510	-0.009	0.000
59	0.504	0.488	0.013	0.000	0.547	-0.046	0.002
60	0.504	0.513	-0.012	0.000	0.549	-0.048	0.002

61	1.003	0.950	0.050	0.003	1.000	0.000	0.000
62	1.003	0.986	0.014	0.000	0.977	0.023	0.001
63	1.003	0.987	0.013	0.000	1.097	-0.097	0.009
64	1.003	0.995	0.005	0.000	1.093	-0.093	0.009
65	1.003	1.011	-0.011	0.000	0.963	0.037	0.001
66	1.003	0.990	0.010	0.000	1.020	-0.020	0.000
67	1.003	1.017	-0.017	0.000	1.053	-0.053	0.003
68	1.003	1.012	-0.012	0.000	1.051	-0.051	0.003
69	1.003	1.013	-0.013	0.000	1.053	-0.053	0.003
70	1.003	0.993	0.007	0.000	1.054	-0.054	0.003
71	2	1.955	0.042	0.002	2.024	-0.027	0.001
72	2	1.994	0.003	0.000	1.984	0.013	0.000
73	2	1.998	-0.001	0.000	2.076	-0.079	0.006
74	2	1.943	0.054	0.003	2.072	-0.075	0.006
75	2	1.984	0.013	0.000	2.102	-0.105	0.011
76	2	1.987	0.010	0.000	2.103	-0.106	0.011
77	2	1.957	0.040	0.002	2.064	-0.067	0.005
78	2	1.944	0.053	0.003	2.062	-0.065	0.004
79	2	1.945	0.052	0.003	2.042	-0.045	0.002
80	2	1.996	0.001	0.000	2.041	-0.044	0.002
81	3.008	2.958	0.047	0.002	3.106	-0.101	0.010
82	3.008	2.944	0.061	0.004	2.915	0.090	0.008
83	3.008	2.928	0.077	0.006	2.941	0.064	0.004
84	3.008	2.917	0.088	0.008	3.016	-0.011	0.000
85	3.008	2.930	0.075	0.006	3.008	-0.003	0.000
86	3.008	2.955	0.050	0.003	3.057	-0.052	0.003
87	3.008	2.952	0.053	0.003	3.122	-0.117	0.014
88	3.008	2.937	0.068	0.005	3.090	-0.085	0.007
89	3.008	2.939	0.066	0.004	3.002	0.003	0.000
90	3.008	2.940	0.065	0.004	2.962	0.043	0.002
91	4.003	3.963	0.037	0.001	4.062	-0.062	0.004
92	4.003	3.929	0.071	0.005	4.066	-0.066	0.004

93	4.003	3.944	0.056	0.003	4.067	-0.067	0.005
94	4.003	3.962	0.038	0.001	4.067	-0.067	0.005
95	4.003	3.949	0.051	0.003	4.068	-0.068	0.005
96	4.003	3.942	0.058	0.003	3.979	0.021	0.000
97	4.003	3.911	0.089	0.008	4.047	-0.047	0.002
98	4.003	3.956	0.044	0.002	3.995	0.005	0.000
99	4.003	3.925	0.075	0.006	4.072	-0.072	0.005
100	4.003	3.963	0.037	0.001	4.074	-0.074	0.005
101	5.003	4.934	0.066	0.004	5.036	-0.036	0.001
102	5.003	4.944	0.056	0.003	5.015	-0.015	0.000
103	5.003	4.927	0.073	0.005	5.015	-0.015	0.000
104	5.003	4.957	0.043	0.002	5.015	-0.015	0.000
105	5.003	4.969	0.031	0.001	5.113	-0.113	0.013
106	5.003	4.964	0.036	0.001	5.045	-0.045	0.002
107	5.003	4.972	0.028	0.001	5.034	-0.034	0.001
108	5.003	4.959	0.041	0.002	4.974	0.026	0.001
109	5.003	4.971	0.029	0.001	5.123	-0.123	0.015
110	5.003	4.960	0.040	0.002	5.058	-0.058	0.003
111	10.001	9.953	0.045	0.002	10.038	-0.040	0.002
112	10.001	9.942	0.056	0.003	10.074	-0.076	0.006
113	10.001	9.926	0.072	0.005	10.101	-0.103	0.011
114	10.001	9.907	0.091	0.008	10.108	-0.110	0.012
115	10.001	9.928	0.070	0.005	10.092	-0.094	0.009
116	10.001	9.945	0.053	0.003	10.077	-0.079	0.006
117	10.001	9.895	0.103	0.011	10.125	-0.127	0.016
118	10.001	9.924	0.074	0.005	10.123	-0.125	0.016
119	10.001	9.980	0.018	0.000	10.161	-0.163	0.027
120	10.001	9.969	0.029	0.001	10.061	-0.063	0.004
121	14.993	14.931	0.059	0.004	15.128	-0.138	0.019
122	14.993	14.924	0.066	0.004	15.101	-0.111	0.012
123	14.993	14.904	0.086	0.007	15.259	-0.269	0.072
124	14.993	14.872	0.118	0.014	15.226	-0.236	0.056

125	14.993	14.907	0.083	0.007	15.100	-0.110	0.012
126	14.993	14.917	0.073	0.005	15.134	-0.144	0.021
127	14.993	14.895	0.095	0.009	15.120	-0.130	0.017
128	14.993	14.897	0.093	0.009	15.226	-0.236	0.056
129	14.993	14.965	0.025	0.001	15.087	-0.097	0.009
130	14.993	14.913	0.077	0.006	15.134	-0.144	0.021
	Max	**14.965**	**0.118**	**0.014**	**15.259**	**0.090**	**0.072**
	Min	**0.026**	**-0.091**	**6.34E-07**	**0.025**	**-0.269**	**2.01E-07**
	Average	**3.163**	**0.028**	**0.002**	**3.244**	**-0.053**	**0.007**
	Standard Deviation	**4.343**	**0.037**	**0.003**	**4.400**	**0.066**	**0.012**

CPU RMSE = 0.046832 mm **DRRACC RMSE = 0.084175 mm**

Appendix E: Stage Rotation Raw Data

Calcaneus 7451 (run 1)

CALC 7451		***CPU (Run 1)***			***CPU (Run 2)***			***DRRACC***		
Frame Number	**Expected Position**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**	**Calculated Position**	**Difference**	**Squared Difference**
1	0.01	0.293	-0.283	0.080	0.226	-0.216	0.047	0.042	-0.032	0.001
2	0.01	0.354	-0.344	0.118	0.195	-0.185	0.034	0.088	-0.078	0.006
3	0.01	0.221	-0.211	0.044	0.226	-0.216	0.047	0.000	0.010	0.000
4	0.01	0.078	-0.068	0.005	0.138	-0.128	0.016	0.092	-0.082	0.007
5	0.01	0.139	-0.129	0.017	0.165	-0.155	0.024	0.178	-0.168	0.028
6	0.01	0.237	-0.227	0.051	0.169	-0.159	0.025	0.000	0.010	0.000
7	0.01	0.240	-0.230	0.053	0.164	-0.154	0.024	0.210	-0.200	0.040
8	0.01	0.300	-0.290	0.084	0.164	-0.154	0.024	0.074	-0.064	0.004
9	0.01	0.222	-0.212	0.045	0.304	-0.294	0.087	0.034	-0.024	0.001
10	0.01	0.165	-0.155	0.024	0.050	-0.040	0.002	0.103	-0.093	0.009
11	0.02	0.250	-0.230	0.053	0.134	-0.114	0.013	0.024	-0.004	0.000
12	0.02	0.168	-0.148	0.022	0.150	-0.130	0.017	0.000	0.020	0.000
13	0.02	0.152	-0.132	0.017	0.273	-0.253	0.064	0.000	0.020	0.000
14	0.02	0.251	-0.231	0.053	0.195	-0.175	0.030	0.000	0.020	0.000
15	0.02	0.220	-0.200	0.040	0.260	-0.240	0.057	0.109	-0.089	0.008
16	0.02	0.181	-0.161	0.026	0.285	-0.265	0.070	0.135	-0.115	0.013
17	0.02	0.236	-0.216	0.047	0.330	-0.310	0.096	0.000	0.020	0.000
18	0.02	0.195	-0.175	0.031	0.228	-0.208	0.043	0.000	0.020	0.000
19	0.02	0.305	-0.285	0.081	0.170	-0.150	0.022	0.063	-0.043	0.002
20	0.02	0.291	-0.271	0.073	0.155	-0.135	0.018	0.103	-0.083	0.007
21	0.03	0.091	-0.061	0.004	0.122	-0.092	0.008	0.054	-0.024	0.001
22	0.03	0.090	-0.060	0.004	0.184	-0.154	0.024	0.089	-0.059	0.003
23	0.03	0.060	-0.030	0.001	0.194	-0.164	0.027	0.082	-0.052	0.003
24	0.03	0.028	0.002	0.000	0.139	-0.109	0.012	0.000	0.030	0.001
25	0.03	0.131	-0.101	0.010	0.257	-0.227	0.051	0.000	0.030	0.001
26	0.03	0.114	-0.084	0.007	0.319	-0.289	0.083	0.000	0.030	0.001
27	0.03	0.155	-0.125	0.016	0.291	-0.261	0.068	0.075	-0.045	0.002

28	0.03	0.113	-0.083	0.007	0.193	-0.163	0.027	0.051	-0.021	0.000
29	0.03	0.107	-0.077	0.006	0.200	-0.170	0.029	0.078	-0.048	0.002
30	0.03	0.244	-0.214	0.046	0.268	-0.238	0.057	0.066	-0.036	0.001
31	0.04	0.348	-0.308	0.095	0.343	-0.303	0.092	0.073	-0.033	0.001
32	0.04	0.169	-0.129	0.017	0.434	-0.394	0.155	0.034	0.006	0.000
33	0.04	0.253	-0.213	0.045	0.342	-0.302	0.091	0.000	0.040	0.002
34	0.04	0.169	-0.129	0.017	0.222	-0.182	0.033	0.098	-0.058	0.003
35	0.04	0.162	-0.122	0.015	0.175	-0.135	0.018	0.104	-0.064	0.004
36	0.04	0.132	-0.092	0.009	0.072	-0.032	0.001	0.112	-0.072	0.005
37	0.04	0.115	-0.075	0.006	0.129	-0.089	0.008	0.155	-0.115	0.013
38	0.04	0.116	-0.076	0.006	0.179	-0.139	0.019	0.097	-0.057	0.003
39	0.04	0.175	-0.135	0.018	0.129	-0.089	0.008	0.066	-0.026	0.001
40	0.04	0.206	-0.166	0.028	0.161	-0.121	0.015	0.072	-0.032	0.001
41	0.05	0.213	-0.163	0.027	0.181	-0.131	0.017	0.025	0.025	0.001
42	0.05	0.261	-0.211	0.044	0.223	-0.173	0.030	0.066	-0.016	0.000
43	0.05	0.242	-0.192	0.037	0.211	-0.161	0.026	0.000	0.050	0.003
44	0.05	0.201	-0.151	0.023	0.189	-0.139	0.019	0.000	0.050	0.003
45	0.05	0.108	-0.058	0.003	0.183	-0.133	0.018	0.119	-0.069	0.005
46	0.05	0.221	-0.171	0.029	0.309	-0.259	0.067	0.119	-0.069	0.005
47	0.05	0.151	-0.101	0.010	0.040	0.010	0.000	0.000	0.050	0.003
48	0.05	0.174	-0.124	0.015	0.057	-0.007	0.000	0.135	-0.085	0.007
49	0.05	0.058	-0.008	0.000	0.179	-0.129	0.017	0.148	-0.098	0.010
50	0.05	0.113	-0.063	0.004	0.167	-0.117	0.014	0.128	-0.078	0.006
51	0.1	0.176	-0.076	0.006	0.122	-0.022	0.000	0.110	-0.010	0.000
52	0.1	0.150	-0.050	0.003	0.127	-0.027	0.001	0.085	0.015	0.000
53	0.1	0.062	0.038	0.001	0.208	-0.108	0.012	0.139	-0.039	0.002
54	0.1	0.101	-0.001	0.000	0.226	-0.126	0.016	0.154	-0.054	0.003
55	0.1	0.139	-0.039	0.002	0.234	-0.134	0.018	0.103	-0.003	0.000
56	0.1	0.187	-0.087	0.008	0.192	-0.092	0.009	0.032	0.068	0.005
57	0.1	0.180	-0.080	0.006	0.222	-0.122	0.015	0.165	-0.065	0.004
58	0.1	0.182	-0.082	0.007	0.236	-0.136	0.019	0.090	0.010	0.000
59	0.1	0.180	-0.080	0.006	0.152	-0.052	0.003	0.159	-0.059	0.004

60	0.1	0.142	-0.042	0.002	0.195	-0.095	0.009	0.079	0.021	0.000
61	0.2	0.251	-0.051	0.003	0.353	-0.153	0.023	0.225	-0.025	0.001
62	0.2	0.303	-0.103	0.011	0.260	-0.060	0.004	0.194	0.006	0.000
63	0.2	0.413	-0.213	0.046	0.189	0.011	0.000	0.262	-0.062	0.004
64	0.2	0.127	0.073	0.005	0.232	-0.032	0.001	0.221	-0.021	0.000
65	0.2	0.196	0.004	0.000	0.440	-0.240	0.058	0.289	-0.089	0.008
66	0.2	0.201	-0.001	0.000	0.303	-0.103	0.011	0.154	0.046	0.002
67	0.2	0.219	-0.019	0.000	0.205	-0.005	0.000	0.243	-0.043	0.002
68	0.2	0.274	-0.074	0.005	0.269	-0.069	0.005	0.172	0.028	0.001
69	0.2	0.256	-0.056	0.003	0.242	-0.042	0.002	0.297	-0.097	0.009
70	0.2	0.251	-0.051	0.003	0.266	-0.066	0.004	0.262	-0.062	0.004
71	0.3	0.294	0.006	0.000	0.327	-0.027	0.001	0.266	0.034	0.001
72	0.3	0.317	-0.017	0.000	0.315	-0.015	0.000	0.258	0.042	0.002
73	0.3	0.369	-0.069	0.005	0.262	0.038	0.001	0.306	-0.006	0.000
74	0.3	0.288	0.012	0.000	0.322	-0.022	0.000	0.358	-0.058	0.003
75	0.3	0.243	0.057	0.003	0.407	-0.107	0.012	0.234	0.066	0.004
76	0.3	0.317	-0.017	0.000	0.343	-0.043	0.002	0.235	0.065	0.004
77	0.3	0.198	0.102	0.010	0.412	-0.112	0.013	0.241	0.059	0.003
78	0.3	0.241	0.059	0.003	0.430	-0.130	0.017	0.241	0.059	0.003
79	0.3	0.240	0.060	0.004	0.414	-0.114	0.013	0.294	0.006	0.000
80	0.3	0.359	-0.059	0.003	0.248	0.052	0.003	0.247	0.053	0.003
81	0.400	0.427	-0.027	0.001	0.491	-0.091	0.008	0.384	0.016	0.000
82	0.400	0.352	0.048	0.002	0.814	-0.414	0.171	0.389	0.011	0.000
83	0.400	0.222	0.178	0.032	0.378	0.022	0.000	0.384	0.016	0.000
84	0.400	0.424	-0.024	0.001	0.420	-0.020	0.000	0.396	0.004	0.000
85	0.400	0.497	-0.097	0.009	0.443	-0.043	0.002	0.326	0.074	0.005
86	0.400	0.549	-0.149	0.022	0.444	-0.044	0.002	0.349	0.051	0.003
87	0.400	0.410	-0.010	0.000	0.405	-0.005	0.000	0.435	-0.035	0.001
88	0.400	0.379	0.021	0.000	0.402	-0.002	0.000	0.449	-0.049	0.002
89	0.400	0.426	-0.026	0.001	0.396	0.004	0.000	0.443	-0.043	0.002
90	0.400	0.363	0.037	0.001	0.422	-0.022	0.000	0.425	-0.025	0.001
91	0.5	0.379	0.121	0.015	0.376	0.124	0.015	0.544	-0.044	0.002

92	0.5	0.389	0.111	0.012	0.483	0.017	0.000	0.559	-0.059	0.003
93	0.5	0.446	0.054	0.003	0.545	-0.045	0.002	0.539	-0.039	0.001
94	0.5	0.488	0.012	0.000	0.471	0.029	0.001	0.509	-0.009	0.000
95	0.5	0.527	-0.027	0.001	0.467	0.033	0.001	0.511	-0.011	0.000
96	0.5	0.674	-0.174	0.030	0.423	0.077	0.006	0.508	-0.008	0.000
97	0.5	0.474	0.026	0.001	0.397	0.103	0.011	0.552	-0.052	0.003
98	0.5	0.423	0.077	0.006	0.501	-0.001	0.000	0.516	-0.016	0.000
99	0.5	0.530	-0.030	0.001	0.668	-0.168	0.028	0.600	-0.100	0.010
100	0.5	0.445	0.055	0.003	0.744	-0.244	0.059	0.592	-0.092	0.008
101	1	0.991	0.009	0.000	1.006	-0.006	0.000	1.082	-0.082	0.007
102	1	0.965	0.035	0.001	1.096	-0.096	0.009	1.120	-0.120	0.014
103	1	0.974	0.026	0.001	1.041	-0.041	0.002	1.033	-0.033	0.001
104	1	1.054	-0.054	0.003	1.012	-0.012	0.000	1.021	-0.021	0.000
105	1	1.119	-0.119	0.014	1.029	-0.029	0.001	1.069	-0.069	0.005
106	1	1.157	-0.157	0.025	1.112	-0.112	0.013	0.975	0.025	0.001
107	1	1.132	-0.132	0.018	1.143	-0.143	0.020	1.058	-0.058	0.003
108	1	1.003	-0.003	0.000	1.203	-0.203	0.041	1.087	-0.087	0.008
109	1	1.071	-0.071	0.005	1.105	-0.105	0.011	1.037	-0.037	0.001
110	1	1.004	-0.004	0.000	1.064	-0.064	0.004	1.017	-0.017	0.000
111	2	1.973	0.027	0.001	1.682	0.318	0.101	2.036	-0.036	0.001
112	2	1.978	0.022	0.000	1.890	0.110	0.012	2.014	-0.014	0.000
113	2	1.988	0.012	0.000	1.995	0.005	0.000	2.054	-0.054	0.003
114	2	1.945	0.055	0.003	2.094	-0.094	0.009	2.057	-0.057	0.003
115	2	1.973	0.027	0.001	2.124	-0.124	0.015	2.053	-0.053	0.003
116	2	1.932	0.068	0.005	1.985	0.015	0.000	2.079	-0.079	0.006
117	2	1.918	0.082	0.007	1.846	0.154	0.024	2.079	-0.079	0.006
118	2	1.903	0.097	0.009	1.899	0.101	0.010	2.054	-0.054	0.003
119	2	1.894	0.106	0.011	1.992	0.008	0.000	2.097	-0.097	0.009
120	2	1.880	0.120	0.014	1.988	0.012	0.000	2.050	-0.050	0.002
121	3	3.021	-0.021	0.000	2.952	0.048	0.002	3.060	-0.060	0.004
122	3	2.987	0.013	0.000	3.020	-0.020	0.000	3.060	-0.060	0.004
123	3	3.047	-0.047	0.002	3.148	-0.148	0.022	3.070	-0.070	0.005

124	3	3.048	-0.048	0.002	3.086	-0.086	0.007	3.080	-0.080	0.006
125	3	2.993	0.007	0.000	3.013	-0.013	0.000	3.124	-0.124	0.015
126	3	2.898	0.102	0.010	3.007	-0.007	0.000	3.104	-0.104	0.011
127	3	3.024	-0.024	0.001	2.985	0.015	0.000	3.149	-0.149	0.022
128	3	3.123	-0.123	0.015	2.958	0.042	0.002	3.075	-0.075	0.006
129	3	3.139	-0.139	0.019	2.997	0.003	0.000	3.052	-0.052	0.003
130	3	3.130	-0.130	0.017	3.095	-0.095	0.009	3.129	-0.129	0.017
131	4	3.936	0.064	0.004	3.870	0.130	0.017	4.077	-0.077	0.006
132	4	4.006	-0.006	0.000	4.106	-0.106	0.011	4.071	-0.071	0.005
133	4	3.943	0.057	0.003	4.096	-0.096	0.009	4.106	-0.106	0.011
134	4	3.857	0.143	0.020	3.967	0.033	0.001	4.130	-0.130	0.017
135	4	3.954	0.046	0.002	3.728	0.272	0.074	4.060	-0.060	0.004
136	4	4.091	-0.091	0.008	3.966	0.034	0.001	4.034	-0.034	0.001
137	4	4.118	-0.118	0.014	4.224	-0.224	0.050	4.119	-0.119	0.014
138	4	4.154	-0.154	0.024	4.178	-0.178	0.032	4.066	-0.066	0.004
139	4	3.938	0.062	0.004	4.307	-0.307	0.094	4.032	-0.032	0.001
140	4	3.934	0.066	0.004	4.089	-0.089	0.008	4.032	-0.032	0.001
141	5	5.037	-0.037	0.001	5.073	-0.073	0.005	5.121	-0.121	0.015
142	5	4.968	0.032	0.001	4.954	0.046	0.002	5.122	-0.122	0.015
143	5	4.994	0.006	0.000	4.955	0.045	0.002	5.121	-0.121	0.015
144	5	5.099	-0.099	0.010	5.061	-0.061	0.004	5.079	-0.079	0.006
145	5	4.935	0.065	0.004	5.085	-0.085	0.007	5.068	-0.068	0.005
146	5	5.009	-0.009	0.000	5.091	-0.091	0.008	5.079	-0.079	0.006
147	5	4.896	0.104	0.011	5.064	-0.064	0.004	5.086	-0.086	0.007
148	5	5.002	-0.002	0.000	5.052	-0.052	0.003	5.079	-0.079	0.006
149	5	5.065	-0.065	0.004	4.945	0.055	0.003	5.120	-0.120	0.014
150	5	5.068	-0.068	0.005	4.905	0.095	0.009	5.057	-0.057	0.003
151	15	15.209	-0.209	0.044	15.085	-0.085	0.007	15.117	-0.117	0.014
152	15	15.182	-0.182	0.033	15.288	-0.288	0.083	15.237	-0.237	0.056
153	15	15.076	-0.076	0.006	15.467	-0.467	0.218	15.155	-0.155	0.024
154	15	15.103	-0.103	0.011	15.501	-0.501	0.251	15.139	-0.139	0.019
155	15	15.129	-0.129	0.017	15.210	-0.210	0.044	15.149	-0.149	0.022

156	15	15.096	-0.096	0.009	15.160	-0.160	0.026	15.188	-0.188	0.035
157	15	15.175	-0.175	0.031	14.858	0.142	0.020	15.188	-0.188	0.035
158	15	15.122	-0.122	0.015	14.672	0.328	0.107	15.114	-0.114	0.013
159	15	14.962	0.038	0.001	14.858	0.142	0.020	15.159	-0.159	0.025
160	15	15.118	-0.118	0.014	15.056	-0.056	0.003	15.127	-0.127	0.016
	Max	**15.209**	**0.178**	**0.118**	**15.501**	**0.328**	**0.251**	**15.237**	**0.074**	**0.056**
	Min	**0.028**	**-0.344**	**0.000**	**0.040**	**-0.501**	**0.000**	**0.000**	**-0.237**	**0.000**
	Average	**2.040**	**-0.061**	**0.014**	**2.065**	**-0.087**	**0.024**	**2.025**	**-0.047**	**0.006**
	Standard Deviation	**3.690**	**0.102**	**0.020**	**3.684**	**0.128**	**0.038**	**3.726**	**0.060**	**0.008**

CPU RMSE (run 1) = 0.119454 mm **CPU RMSE (run 2) = 0.154383 mm** **DRRACC RMSE = 0.075858 mm**

Calcaneus 7451 (run 2)

CALC 7451		*CPU (Run 1)*			*CPU (Run 2)*			*DRRACC*		
Frame Number	Expected Position	Calculated Position	Difference	Squared Difference	Calculated Position	Difference	Squared Difference	Calculated Position	Difference	Squared Difference
1	0.01	0.293	-0.283	0.080	0.226	0.216	0.047	0.091	0.081	0.007
2	0.01	0.354	-0.344	0.118	0.195	0.185	0.034	0.091	0.081	0.007
3	0.01	0.221	-0.211	0.044	0.226	0.216	0.047	0.125	0.115	0.013
4	0.01	0.078	-0.068	0.005	0.138	0.128	0.016	0.138	0.128	0.016
5	0.01	0.139	-0.129	0.017	0.165	0.155	0.024	0.134	0.124	0.015
6	0.01	0.237	-0.227	0.051	0.169	0.159	0.025	0.108	0.098	0.010
7	0.01	0.240	-0.230	0.053	0.164	0.154	0.024	0.193	0.183	0.033
8	0.01	0.300	-0.290	0.084	0.164	0.154	0.024	0.122	0.112	0.012
9	0.01	0.222	-0.212	0.045	0.304	0.294	0.087	0.192	0.182	0.033
10	0.01	0.165	-0.155	0.024	0.050	0.040	0.002	0.069	0.059	0.003
11	0.02	0.250	-0.230	0.053	0.134	0.114	0.013	0.160	0.140	0.020
12	0.02	0.168	-0.148	0.022	0.150	0.130	0.017	0.154	0.134	0.018
13	0.02	0.152	-0.132	0.017	0.273	0.253	0.064	0.162	0.142	0.020
14	0.02	0.251	-0.231	0.053	0.195	0.175	0.030	0.160	0.140	0.020
15	0.02	0.220	-0.200	0.040	0.260	0.240	0.057	0.102	0.082	0.007
16	0.02	0.181	-0.161	0.026	0.285	0.265	0.070	0.097	0.077	0.006
17	0.02	0.236	-0.216	0.047	0.330	0.310	0.096	0.128	0.108	0.012
18	0.02	0.195	-0.175	0.031	0.228	0.208	0.043	0.158	0.138	0.019
19	0.02	0.305	-0.285	0.081	0.170	0.150	0.022	0.157	0.137	0.019
20	0.02	0.291	-0.271	0.073	0.155	0.135	0.018	0.092	0.072	0.005
21	0.03	0.091	-0.061	0.004	0.122	0.092	0.008	0.093	0.063	0.004
22	0.03	0.090	-0.060	0.004	0.184	0.154	0.024	0.089	0.059	0.003
23	0.03	0.060	-0.030	0.001	0.194	0.164	0.027	0.089	0.059	0.003
24	0.03	0.028	0.002	0.000	0.139	0.109	0.012	0.166	0.136	0.018
25	0.03	0.131	-0.101	0.010	0.257	0.227	0.051	0.167	0.137	0.019
26	0.03	0.114	-0.084	0.007	0.319	0.289	0.083	0.147	0.117	0.014
27	0.03	0.155	-0.125	0.016	0.291	0.261	0.068	0.148	0.118	0.014
28	0.03	0.113	-0.083	0.007	0.193	0.163	0.027	0.146	0.116	0.013

29	0.03	0.107	-0.077	0.006	0.200	0.170	0.029	0.084	0.054	0.003
30	0.03	0.244	-0.214	0.046	0.268	0.238	0.057	0.055	0.025	0.001
31	0.04	0.348	-0.308	0.095	0.343	0.303	0.092	0.162	0.122	0.015
32	0.04	0.169	-0.129	0.017	0.434	0.394	0.155	0.146	0.106	0.011
33	0.04	0.253	-0.213	0.045	0.342	0.302	0.091	0.089	0.049	0.002
34	0.04	0.169	-0.129	0.017	0.222	0.182	0.033	0.092	0.052	0.003
35	0.04	0.162	-0.122	0.015	0.175	0.135	0.018	0.060	0.020	0.000
36	0.04	0.132	-0.092	0.009	0.072	0.032	0.001	0.139	0.099	0.010
37	0.04	0.115	-0.075	0.006	0.129	0.089	0.008	0.216	0.176	0.031
38	0.04	0.116	-0.076	0.006	0.179	0.139	0.019	0.079	0.039	0.002
39	0.04	0.175	-0.135	0.018	0.129	0.089	0.008	0.150	0.110	0.012
40	0.04	0.206	-0.166	0.028	0.161	0.121	0.015	0.206	0.166	0.028
41	0.05	0.213	-0.163	0.027	0.181	0.131	0.017	0.130	0.080	0.006
42	0.05	0.261	-0.211	0.044	0.223	0.173	0.030	0.145	0.095	0.009
43	0.05	0.242	-0.192	0.037	0.211	0.161	0.026	0.141	0.091	0.008
44	0.05	0.201	-0.151	0.023	0.189	0.139	0.019	0.137	0.087	0.008
45	0.05	0.108	-0.058	0.003	0.183	0.133	0.018	0.153	0.103	0.011
46	0.05	0.221	-0.171	0.029	0.309	0.259	0.067	0.048	-0.002	0.000
47	0.05	0.151	-0.101	0.010	0.040	-0.010	0.000	0.148	0.098	0.010
48	0.05	0.174	-0.124	0.015	0.057	0.007	0.000	0.117	0.067	0.004
49	0.05	0.058	-0.008	0.000	0.179	0.129	0.017	0.182	0.132	0.018
50	0.05	0.113	-0.063	0.004	0.167	0.117	0.014	0.147	0.097	0.009
51	0.1	0.176	-0.076	0.006	0.122	0.022	0.000	0.154	0.054	0.003
52	0.1	0.150	-0.050	0.003	0.127	0.027	0.001	0.122	0.022	0.000
53	0.1	0.062	0.038	0.001	0.208	0.108	0.012	0.122	0.022	0.000
54	0.1	0.101	-0.001	0.000	0.226	0.126	0.016	0.118	0.018	0.000
55	0.1	0.139	-0.039	0.002	0.234	0.134	0.018	0.180	0.080	0.006
56	0.1	0.187	-0.087	0.008	0.192	0.092	0.009	0.085	-0.015	0.000
57	0.1	0.180	-0.080	0.006	0.222	0.122	0.015	0.141	0.041	0.002
58	0.1	0.182	-0.082	0.007	0.236	0.136	0.019	0.124	0.024	0.001
59	0.1	0.180	-0.080	0.006	0.152	0.052	0.003	0.181	0.081	0.006
60	0.1	0.142	-0.042	0.002	0.195	0.095	0.009	0.160	0.060	0.004

61	0.2	0.251	-0.051	0.003	0.353	0.153	0.023	0.178	-0.022	0.000
62	0.2	0.303	-0.103	0.011	0.260	0.060	0.004	0.159	-0.041	0.002
63	0.2	0.413	-0.213	0.046	0.189	-0.011	0.000	0.159	-0.041	0.002
64	0.2	0.127	0.073	0.005	0.232	0.032	0.001	0.206	0.006	0.000
65	0.2	0.196	0.004	0.000	0.440	0.240	0.058	0.228	0.028	0.001
66	0.2	0.201	-0.001	0.000	0.303	0.103	0.011	0.108	-0.092	0.008
67	0.2	0.219	-0.019	0.000	0.205	0.005	0.000	0.203	0.003	0.000
68	0.2	0.274	-0.074	0.005	0.269	0.069	0.005	0.156	-0.044	0.002
69	0.2	0.256	-0.056	0.003	0.242	0.042	0.002	0.229	0.029	0.001
70	0.2	0.251	-0.051	0.003	0.266	0.066	0.004	0.227	0.027	0.001
71	0.3	0.294	0.006	0.000	0.327	0.027	0.001	0.291	-0.009	0.000
72	0.3	0.317	-0.017	0.000	0.315	0.015	0.000	0.226	-0.074	0.005
73	0.3	0.369	-0.069	0.005	0.262	-0.038	0.001	0.329	0.029	0.001
74	0.3	0.288	0.012	0.000	0.322	0.022	0.000	0.268	-0.032	0.001
75	0.3	0.243	0.057	0.003	0.407	0.107	0.012	0.225	-0.075	0.006
76	0.3	0.317	-0.017	0.000	0.343	0.043	0.002	0.212	-0.088	0.008
77	0.3	0.198	0.102	0.010	0.412	0.112	0.013	0.252	-0.048	0.002
78	0.3	0.241	0.059	0.003	0.430	0.130	0.017	0.313	0.013	0.000
79	0.3	0.240	0.060	0.004	0.414	0.114	0.013	0.265	-0.035	0.001
80	0.3	0.359	-0.059	0.003	0.248	-0.052	0.003	0.175	-0.125	0.016
81	0.400	0.427	-0.027	0.001	0.491	0.091	0.008	0.296	-0.104	0.011
82	0.400	0.352	0.048	0.002	0.814	0.414	0.171	0.359	-0.041	0.002
83	0.400	0.222	0.178	0.032	0.378	-0.022	0.000	0.278	-0.122	0.015
84	0.400	0.424	-0.024	0.001	0.420	0.020	0.000	0.286	-0.114	0.013
85	0.400	0.497	-0.097	0.009	0.443	0.043	0.002	0.286	-0.114	0.013
86	0.400	0.549	-0.149	0.022	0.444	0.044	0.002	0.272	-0.128	0.017
87	0.400	0.410	-0.010	0.000	0.405	0.005	0.000	0.270	-0.130	0.017
88	0.400	0.379	0.021	0.000	0.402	0.002	0.000	0.396	-0.004	0.000
89	0.400	0.426	-0.026	0.001	0.396	-0.004	0.000	0.292	-0.108	0.012
90	0.400	0.363	0.037	0.001	0.422	0.022	0.000	0.354	-0.046	0.002
91	0.5	0.379	0.121	0.015	0.376	-0.124	0.015	0.389	-0.111	0.012
92	0.5	0.389	0.111	0.012	0.483	-0.017	0.000	0.465	-0.035	0.001

93	0.5	0.446	0.054	0.003	0.545	0.045	0.002	0.415	-0.085	0.007
94	0.5	0.488	0.012	0.000	0.471	-0.029	0.001	0.388	-0.112	0.013
95	0.5	0.527	-0.027	0.001	0.467	-0.033	0.001	0.448	-0.052	0.003
96	0.5	0.674	-0.174	0.030	0.423	-0.077	0.006	0.374	-0.126	0.016
97	0.5	0.474	0.026	0.001	0.397	-0.103	0.011	0.490	-0.010	0.000
98	0.5	0.423	0.077	0.006	0.501	0.001	0.000	0.495	-0.005	0.000
99	0.5	0.530	-0.030	0.001	0.668	0.168	0.028	0.495	-0.005	0.000
100	0.5	0.445	0.055	0.003	0.744	0.244	0.059	0.475	-0.025	0.001
101	1	0.991	0.009	0.000	1.006	0.006	0.000	0.994	-0.006	0.000
102	1	0.965	0.035	0.001	1.096	0.096	0.009	1.026	0.026	0.001
103	1	0.974	0.026	0.001	1.041	0.041	0.002	0.941	-0.059	0.003
104	1	1.054	-0.054	0.003	1.012	0.012	0.000	0.972	-0.028	0.001
105	1	1.119	-0.119	0.014	1.029	0.029	0.001	0.951	-0.049	0.002
106	1	1.157	-0.157	0.025	1.112	0.112	0.013	0.948	-0.052	0.003
107	1	1.132	-0.132	0.018	1.143	0.143	0.020	0.956	-0.044	0.002
108	1	1.003	-0.003	0.000	1.203	0.203	0.041	0.897	-0.103	0.011
109	1	1.071	-0.071	0.005	1.105	0.105	0.011	0.931	-0.069	0.005
110	1	1.004	-0.004	0.000	1.064	0.064	0.004	0.969	-0.031	0.001
111	2	1.973	0.027	0.001	1.682	-0.318	0.101	2.047	0.047	0.002
112	2	1.978	0.022	0.000	1.890	-0.110	0.012	1.985	-0.015	0.000
113	2	1.988	0.012	0.000	1.995	-0.005	0.000	2.001	0.001	0.000
114	2	1.945	0.055	0.003	2.094	0.094	0.009	2.022	0.022	0.000
115	2	1.973	0.027	0.001	2.124	0.124	0.015	1.991	-0.009	0.000
116	2	1.932	0.068	0.005	1.985	-0.015	0.000	1.994	-0.006	0.000
117	2	1.918	0.082	0.007	1.846	-0.154	0.024	2.011	0.011	0.000
118	2	1.903	0.097	0.009	1.899	-0.101	0.010	1.964	-0.036	0.001
119	2	1.894	0.106	0.011	1.992	-0.008	0.000	2.010	0.010	0.000
120	2	1.880	0.120	0.014	1.988	-0.012	0.000	2.011	0.011	0.000
121	3	3.021	-0.021	0.000	2.952	-0.048	0.002	2.988	-0.012	0.000
122	3	2.987	0.013	0.000	3.020	0.020	0.000	3.026	0.026	0.001
123	3	3.047	-0.047	0.002	3.148	0.148	0.022	3.036	0.036	0.001
124	3	3.048	-0.048	0.002	3.086	0.086	0.007	3.021	0.021	0.000

125	3	2.993	0.007	0.000	3.013	0.013	0.000	3.097	0.097	0.009
126	3	2.898	0.102	0.010	3.007	0.007	0.000	3.063	0.063	0.004
127	3	3.024	-0.024	0.001	2.985	-0.015	0.000	3.063	0.063	0.004
128	3	3.123	-0.123	0.015	2.958	-0.042	0.002	3.039	0.039	0.001
129	3	3.139	-0.139	0.019	2.997	-0.003	0.000	2.966	-0.034	0.001
130	3	3.130	-0.130	0.017	3.095	0.095	0.009	3.058	0.058	0.003
131	4	3.936	0.064	0.004	3.870	-0.130	0.017	4.044	0.044	0.002
132	4	4.006	-0.006	0.000	4.106	0.106	0.011	4.039	0.039	0.001
133	4	3.943	0.057	0.003	4.096	0.096	0.009	4.065	0.065	0.004
134	4	3.857	0.143	0.020	3.967	-0.033	0.001	4.066	0.066	0.004
135	4	3.954	0.046	0.002	3.728	-0.272	0.074	4.000	0.000	0.000
136	4	4.091	-0.091	0.008	3.966	-0.034	0.001	3.959	-0.041	0.002
137	4	4.118	-0.118	0.014	4.224	0.224	0.050	4.013	0.013	0.000
138	4	4.154	-0.154	0.024	4.178	0.178	0.032	4.017	0.017	0.000
139	4	3.938	0.062	0.004	4.307	0.307	0.094	3.997	-0.003	0.000
140	4	3.934	0.066	0.004	4.089	0.089	0.008	4.023	0.023	0.001
141	5	5.037	-0.037	0.001	5.073	0.073	0.005	5.067	0.067	0.005
142	5	4.968	0.032	0.001	4.954	-0.046	0.002	5.064	0.064	0.004
143	5	4.994	0.006	0.000	4.955	-0.045	0.002	5.080	0.080	0.006
144	5	5.099	-0.099	0.010	5.061	0.061	0.004	5.012	0.012	0.000
145	5	4.935	0.065	0.004	5.085	0.085	0.007	5.014	0.014	0.000
146	5	5.009	-0.009	0.000	5.091	0.091	0.008	5.024	0.024	0.001
147	5	4.896	0.104	0.011	5.064	0.064	0.004	5.027	0.027	0.001
148	5	5.002	-0.002	0.000	5.052	0.052	0.003	5.015	0.015	0.000
149	5	5.065	-0.065	0.004	4.945	-0.055	0.003	5.065	0.065	0.004
150	5	5.068	-0.068	0.005	4.905	-0.095	0.009	5.017	0.017	0.000
151	15	15.209	-0.209	0.044	15.085	0.085	0.007	15.096	0.096	0.009
152	15	15.182	-0.182	0.033	15.288	0.288	0.083	15.192	0.192	0.037
153	15	15.076	-0.076	0.006	15.467	0.467	0.218	15.143	0.143	0.020
154	15	15.103	-0.103	0.011	15.501	0.501	0.251	15.142	0.142	0.020
155	15	15.129	-0.129	0.017	15.210	0.210	0.044	15.080	0.080	0.006
156	15	15.096	-0.096	0.009	15.160	0.160	0.026	15.129	0.129	0.017

157	15	15.175	-0.175	0.031	14.858	-0.142	0.020	15.147	0.147	0.022
158	15	15.122	-0.122	0.015	14.672	-0.328	0.107	15.118	0.118	0.014
159	15	14.962	0.038	0.001	14.858	-0.142	0.020	15.113	0.113	0.013
160	15	15.118	-0.118	0.014	15.056	0.056	0.003	15.112	0.112	0.012
	Max	**15.209**	**0.178**	**0.118**	**15.501**	**0.501**	**0.251**	**15.192**	**0.192**	**0.037**
	Min	**0.028**	**-0.344**	**0.000**	**0.040**	**-0.328**	**0.000**	**0.048**	**-0.130**	**0.000**
	Average	**2.040**	**-0.061**	**0.014**	**2.065**	**0.087**	**0.024**	**2.010**	**0.032**	**0.007**
	Standard Deviation	**3.690**	**0.102**	**0.020**	**3.684**	**0.128**	**0.038**	**3.711**	**0.076**	**0.008**

CPU RMSE (run 1) = 0.119454 mm **CPU RMSE (run 2) = 0.154383 mm** **DRRACC RMSE = 0.082276 mm**

Appendix F: Image Formation Validation

This notebook was created to validate DRRACC's image formation method

Set the locations of the beads based on the CMM data

```
cmmData = Import["BallCMMCoordinates.txt", "Data"]
```

In[5]:=
```
cmmData = {{12.39`, 25.006`, 26.818`, 4.778`}, {12.456`, 50.098`, 61.223`, 3.179`}, {12.463`, 87.541`, 26.835`, 4.759`},
  {24.88`, 12.383`, 95.682`, 1.781`}, {24.903`, 74.898`, 95.788`, 1.784`}, {37.501`, 50.049`, 26.845`, 4.753`},
  {49.934`, 12.586`, 61.112`, 3.237`}, {49.948`, 50.125`, 61.166`, 3.23`}, {49.956`, 87.585`, 61.18`, 3.199`},
  {62.389`, 49.891`, 95.684`, 1.781`}, {75.009`, 25.069`, 26.839`, 4.762`}, {75.033`, 87.558`, 26.847`, 4.763`},
  {87.381`, 12.388`, 95.682`, 1.775`}, {87.399`, 74.886`, 95.67`, 1.786`}, {87.421`, 50.094`, 61.041`, 3.254`}};
```

```
Dimensions[cmmData]
```

```
{15, 4}
```

In[6]:=
```
beads = Table[Drop[cmmData[[i]], 1], {i, 1, 15}]
```

Out[6]=
```
{{12.39, 25.006, 26.818}, {12.456, 50.098, 61.223}, {12.463, 87.541, 26.835},
 {24.88, 12.383, 95.682}, {24.903, 74.898, 95.788}, {37.501, 50.049, 26.845}, {49.934, 12.586, 61.112},
 {49.948, 50.125, 61.166}, {49.956, 87.585, 61.18}, {62.389, 49.891, 95.684}, {75.009, 25.069, 26.839},
 {75.033, 87.558, 26.847}, {87.381, 12.388, 95.682}, {87.399, 74.886, 95.67}, {87.421, 50.094, 61.041}}
```

In[7]:=
```
sizes = Table[Drop[cmmData[[i]], 3], {i, 1, 15}]
```

Out[7]=
```
{{4.778}, {3.179}, {4.759}, {1.781}, {1.784}, {4.753},
 {3.237}, {3.23}, {3.199}, {1.781}, {4.762}, {4.763}, {1.775}, {1.786}, {3.254}}
```

In[8]:=
```
beads3d = Table[Sphere[Drop[cmmData[[i]], 1], sizes[[i]][[1]]], {i, 1, 15}];
```

Focal point (BLUE)

In[9]:=
```
eyeb = {672.4768961908517, 1009.5425632975199, 1208.844091252439};
```

Focal point (GREEN)

In[10]:=
```
eyeg = {767.8220168057279, 353.26746505378617, 966.4097178002654};
```

Screen computation/creation (BLUE)

In[11]:=
```
uL = {290.0542715777285, 150.59278778999476, 52.077653273049464};
```

In[12]:=
```
uR = {111.30087628283718, 50.23983685144l6, 318.1360187076548};
```

In[13]:=
```
LL = {57.15024475058999, 328.87233240581725, 30.172545109682005};
```

In[14]:=
```
dirX = uR - uL
```

```
1152
```

In[15]:=
```
dirY = LL - uL
```

```
896
```

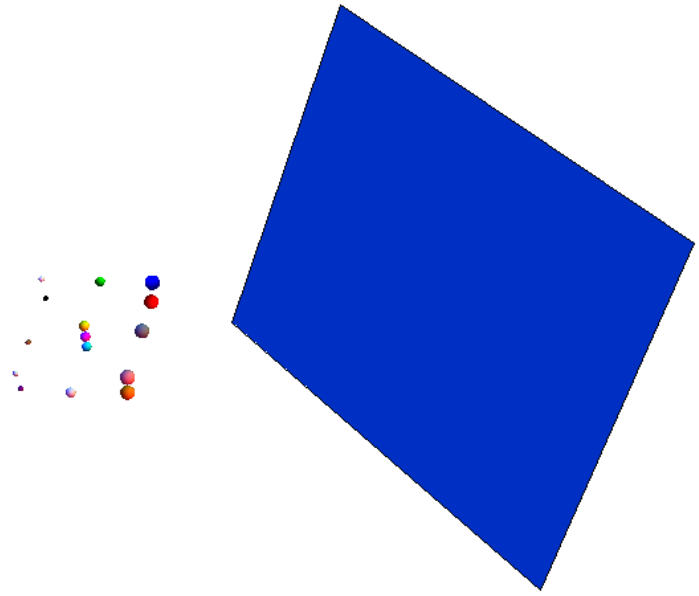
In[16]:=
```
LR = LL + dirX
```

```
1152
```

In[17]:=
```
screen1 = Polygon[{LL, LR, uR, uL}];
```

In[19]:=
```
Show[{Graphics3D[{Blue,Sphere[eyeb,100]}],
  Graphics3D[{
    Red, beads3d[[1]],
    Green, beads3d[[2]],
    Blue, beads3d[[3]],
    Black, beads3d[[4]],
    White, beads3d[[5]],
    Gray, beads3d[[6]],
    Cyan, beads3d[[7]],
    Magenta, beads3d[[8]],
    Yellow, beads3d[[9]],
    Brown, beads3d[[10]],
    Orange, beads3d[[11]],
    Pink, beads3d[[12]],
    Purple, beads3d[[13]],
    LightGreen, beads3d[[14]],
    LightBrown, beads3d[[15]]
  }], Graphics3D[{Cyan, screen1}], Boxed -> False}]
```

Out[19]=



Screen computation/creation (GREEN)

In[20]:=
```
uLg = {346.41766156116694, 203.4132250633579, 24.955167228992195};
```

In[21]:=
```
uRg = {90.20314430703422, 145.6305770648056, 289.76166922614937};
```

In[22]:=
```
LLg = {425.50028764938213, 75.16900723573724, 40.68324704342342};
```

In[23]:=
```
dirXg = uRg - uLg
```
Out[23]= 1152

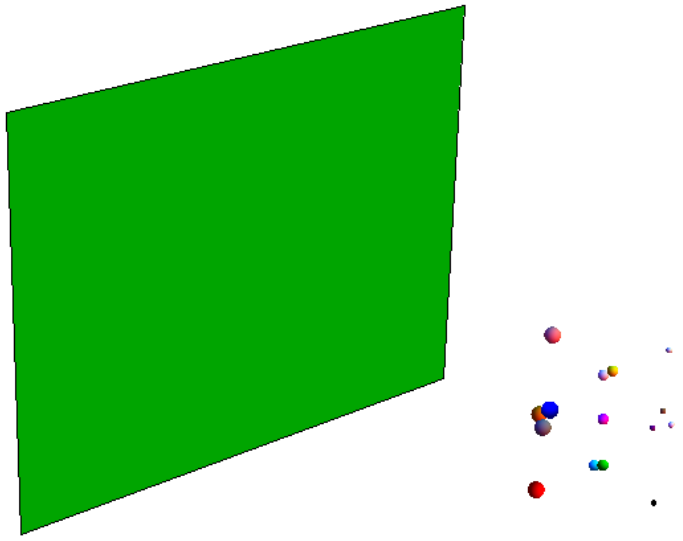
In[24]:=
```
dirYg = LLg - uLg
```
Out[24]= 896

In[25]:=
```
LRg = LLg + dirXg
```
Out[25]= 1152

In[26]:=
```
screeng = Polygon[{LLg, uLg, uRg, LRg}];
```

In[27]:=
```
Show[{Graphics3D[{Blue, Sphere[eyeb, 100]}], Graphics3D[
    {Red, beads3d[1], Green, beads3d[2], Blue, beads3d[3],
     Black, beads3d[4], White, beads3d[5], Gray, beads3d[6],
     Cyan, beads3d[7], Magenta, beads3d[8], Yellow, beads3d[9],
     Brown, beads3d[10], Orange, beads3d[11], Pink, beads3d[12],
     Purple, beads3d[13], LightGreen, beads3d[14],
     LightBrown, beads3d[15]}]}], Graphics3D[{Green, screeng}], Boxed -> False]
```

Out[27]=



Transformation matrix

In[28]:=
```
tMat = {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};
```

Computation of transformed focus and screen corners (BLUE)

In[29]:=
```
tEye = Drop[tMat.{eyeb[[1]], eyeb[[2]], eyeb[[3]], 1}, -1]
```

In[30]:=
```
tuL = Drop[tMat.{uL[[1]], uL[[2]], uL[[3]], 1}, -1]
```

In[31]:=
```
tuR = Drop[tMat.{uR[[1]], uR[[2]], uR[[3]], 1}, -1]
```

In[32]:=
```
tLL = Drop[tMat.{LL[[1]], LL[[2]], LL[[3]], 1}, -1]
```

In[33]:=
```
tLR = Drop[tMat.{LR[[1]], LR[[2]], LR[[3]], 1}, -1]
```

In[34]:=
```
tscreen1 = Polygon[{tLL, tLR, tuR, tuL}];
```

Computation of transformed focus and screen corners (GREEN)

In[35]:=
```
tEyeg = Drop[tMat.{eyeg[[1]], eyeg[[2]], eyeg[[3]], 1}, -1]
```

In[36]:=
```
tuLg = Drop[tMat.{uLg[[1]], uLg[[2]], uLg[[3]], 1}, -1]
```

In[37]:=
```
tuRg = Drop[tMat.{uRg[[1]], uRg[[2]], uRg[[3]], 1}, -1]
```

In[38]:=
```
tLLg = Drop[tMat.{LLg[[1]], LLg[[2]], LLg[[3]], 1}, -1]
```

In[39]:=
```
tLRg = Drop[tMat.{LRg[[1]], LRg[[2]], LRg[[3]], 1}, -1]
```

In[40]:=
```
tscreen1g = Polygon[{tLLg, tLRg, tuRg, tuLg}];
```

Normal of plane of screen (BLUE)

In[48]:=
```
norm = Cross[tuR - tuL, tLL - tuL];
```

In[49]:=
```
unorm = norm / Sqrt[norm.norm]
```

Normal of plane of screen (GREEN)

In[50]:=
```
normg = Cross[tuRg - tuLg, tLLg - tuLg];
```

In[51]:=
```
unormg = normg / Sqrt[normg.normg]
```

Location of beads projected onto screen (BLUE)

```
In[52]:= beadProjB :=
  Table[ beads[[ i ]] + unorm.[ tuL - beads[[ i ]] ] beads[[ i ]] - tEye / Sqrt[ beads[[ i ]] - tEye .[ beads[[ i ]] - tEye ] ,
   { i, 1, 15 } ];

In[53]:= beadProjBsp := Table[ Sphere[ beadProjB[[ i ]] , sizes[[ i ]] ], { i, 1, 15 } ];
```

Find the 2D location of the beads (BLUE)

```
In[55]:= cmm2DBx := Table[ { beadProjB[[ i ]] - uL . dirX } [[ 1 ]] , { i, 1, 15 } ] // Flatten;

In[56]:= cmm2DBy := Table[ { beadProjB[[ i ]] - uL . dirY } [[ 2 ]] , { i, 1, 15 } ] // Flatten;

In[57]:= cmm2DB := Table[ { cmm2DBx[[ i ]] , cmm2DBy[[ i ]] } , { i, 1, 15 } ];
```

Location of beads projected onto screen (GREEN)

```
In[58]:= beadProjG :=
  Table[ beads[[ i ]] + unormg.[ tuLg - beads[[ i ]] ] beads[[ i ]] - tEyeg / Sqrt[ beads[[ i ]] - tEyeg .[ beads[[ i ]] - tEyeg ] ,
   { i, 1, 15 } ];

In[59]:= beadProjGsp := Table[ Sphere[ beadProjG[[ i ]] , sizes[[ i ]] ], { i, 1, 15 } ];
```

Find the 2D location of the beads (GREEN)

```
In[60]:= cmm2DGx := Table[ { beadProjG[[ i ]] - uLg . dirXg } [[ 1 ]] , { i, 1, 15 } ] // Flatten;

In[61]:= cmm2DGy := Table[ { beadProjG[[ i ]] - uLg . dirYg } [[ 2 ]] , { i, 1, 15 } ] // Flatten;

In[62]:= cmm2DG := Table[ { cmm2DGx[[ i ]] , cmm2DGy[[ i ]] } , { i, 1, 15 } ];

In[63]:= cmm2DGDisk := Table[ Disk[ cmm2DG[[ i ]] , 2 ] , { i, 1, 15 } ];

In[64]:= Show[ Graphics[ Disk[ { 0, 0 } , 5 ] ] , Graphics[ Disk[ { 1152, 0 } , 5 ] ] , Graphics[ Disk[ { 1152, 896 } , 5 ] ] ,
  Graphics[ Disk[ { 0, 896 } , 5 ] ] , Graphics[ cmm2DGDisk ] ]
```

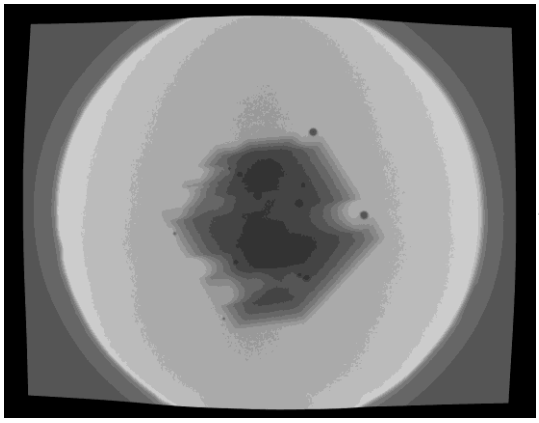
Determine the location of the beads in the fluoroscope images

```
SetDirectory[ "C:\\Users\\Grant\\Desktop\\Calibration Block Data 10.1.2013" ] ;
```

Find the locations for the BLUE image

```
f1 := Import[ "blue_cal_AvgRate10_0001_CorrectedBilinear.tif" ]
```

```
f :=
```



```
;
```

```
f2 := Partition[ BinaryReadList[ "bc_ngrad.bin", "Real32" ] , 1152 ] ;

Image[ f2 - Max[ f2 ] + 1000 ]

Export[ "15beadsBlue.raw", Flatten[ ImageData[ f ] ] , "Real32" ]
```

```
15beadsBlue.raw
```

Use the “Get Indices” function to find the bead indices

v1 - Manual selection using Mathematica tools

Appendix G: Focal Length Parameterization Profile Plots

Focal position - 10% closer to intensifier screen

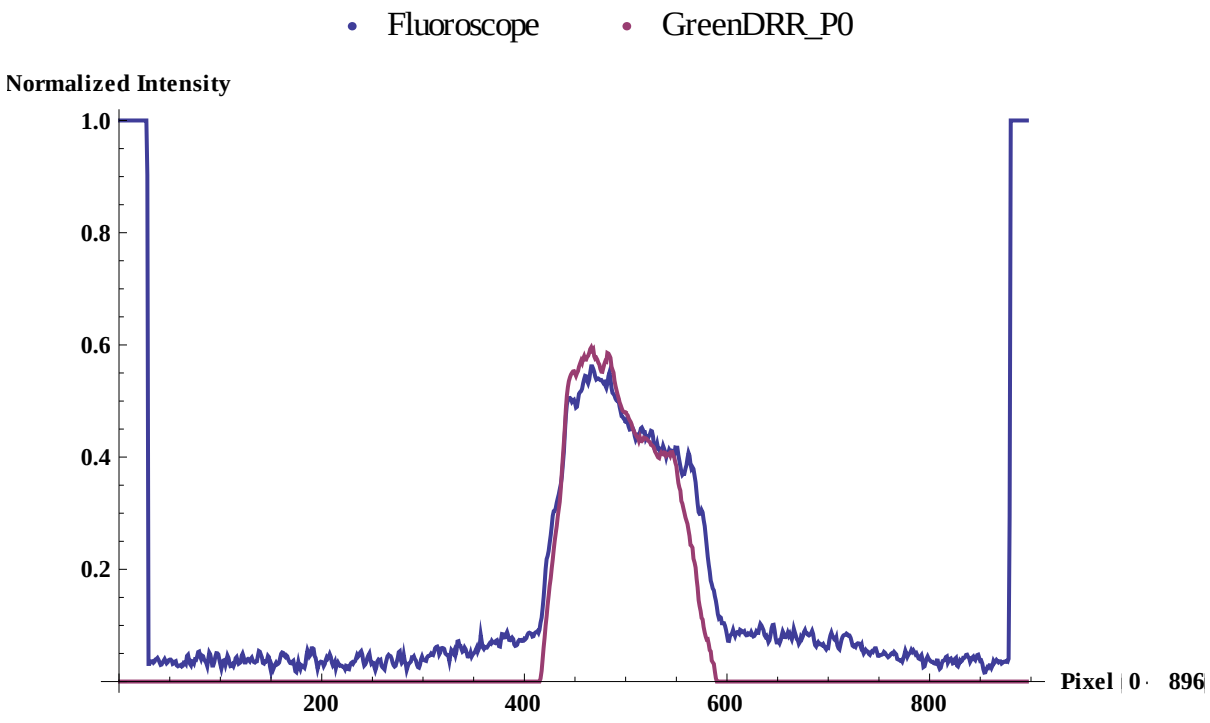
```
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

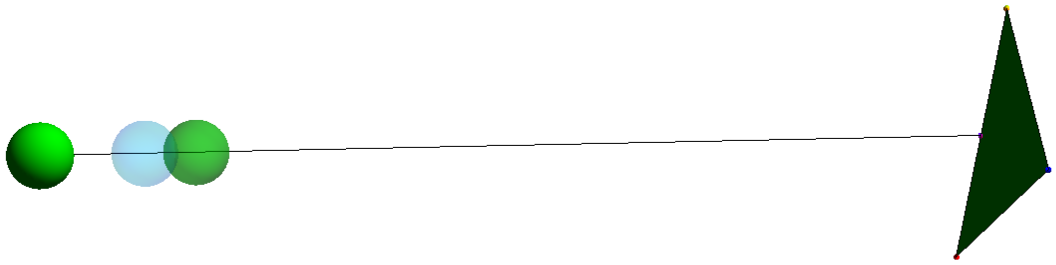
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

Plot fluoro vs. DRR - Green

```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined→True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```



Focal position - 15% closer to intensifier screen



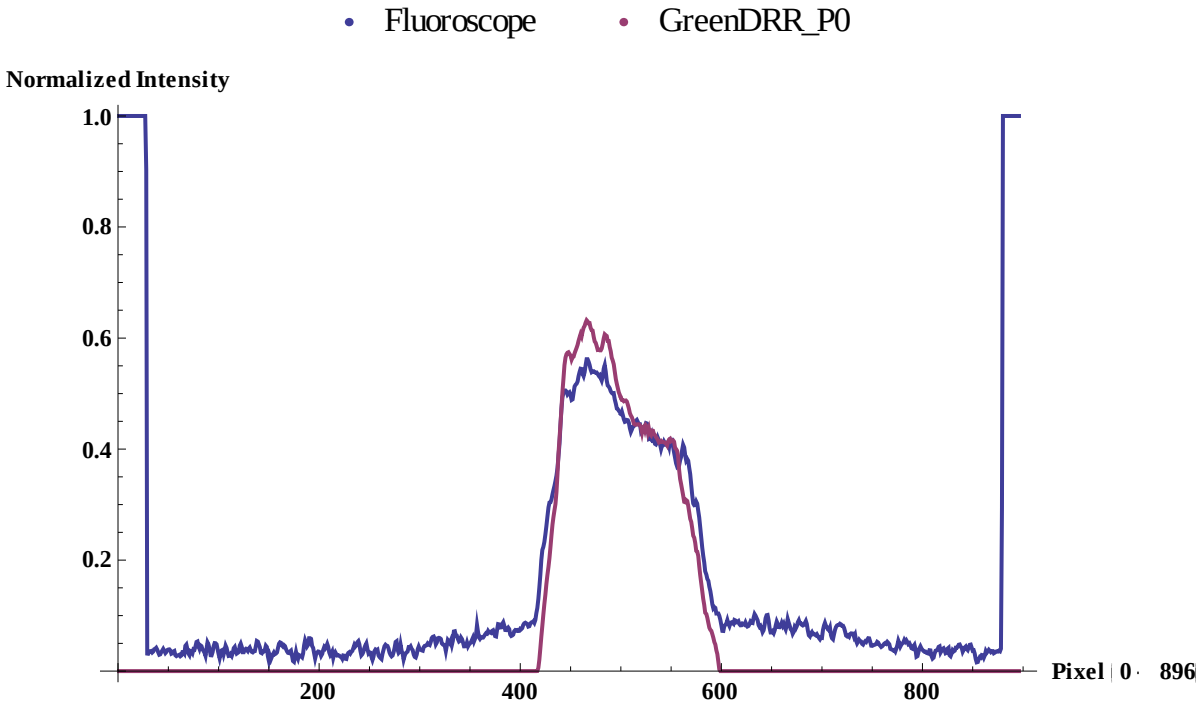
```
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

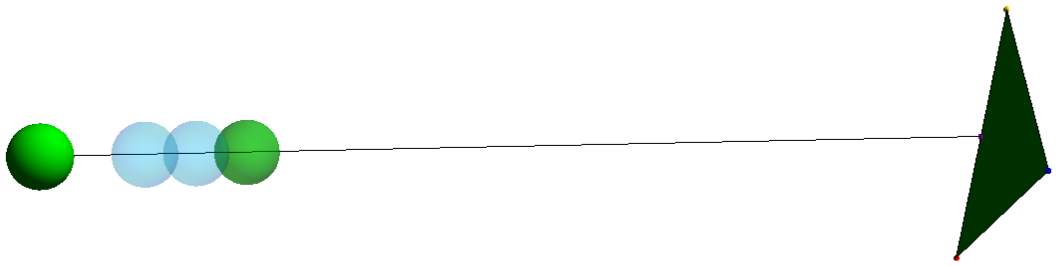
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

Plot fluoro vs. DRR - Green

```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined→True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```



Focal position - 20% closer to intensifier screen



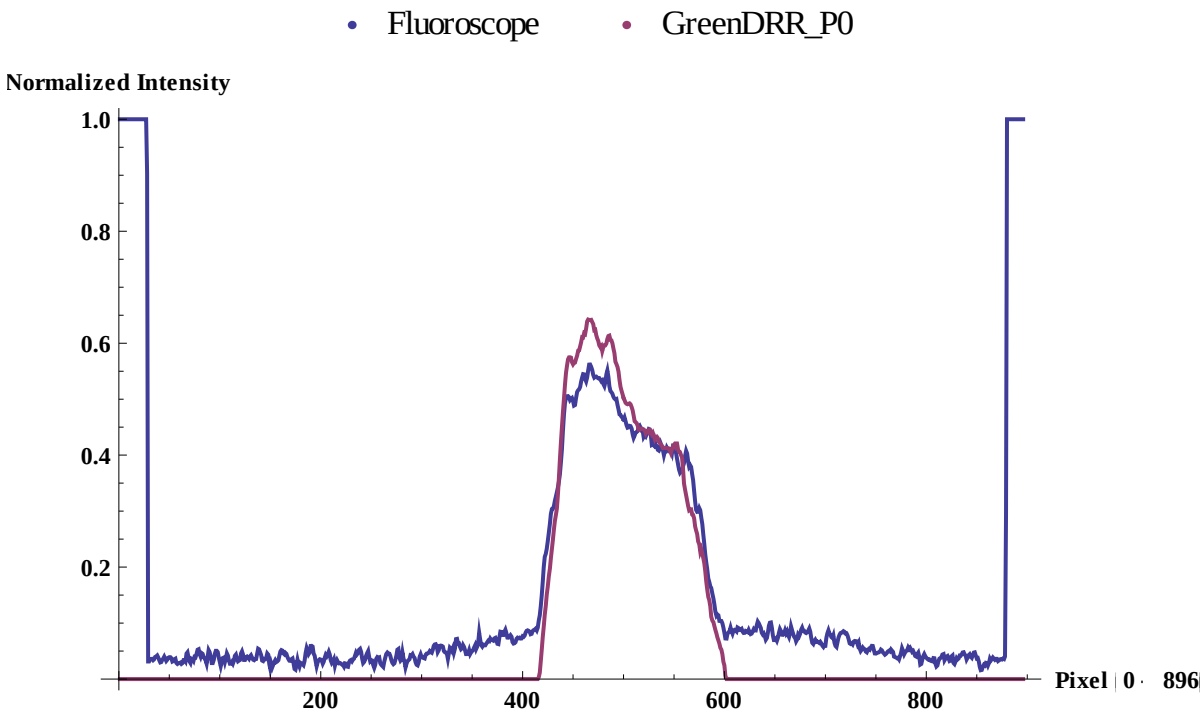
```
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vRC-3.0\\DRRACC v2"];
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vBeta\\DRRACC v2\\stage
rotation"];
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

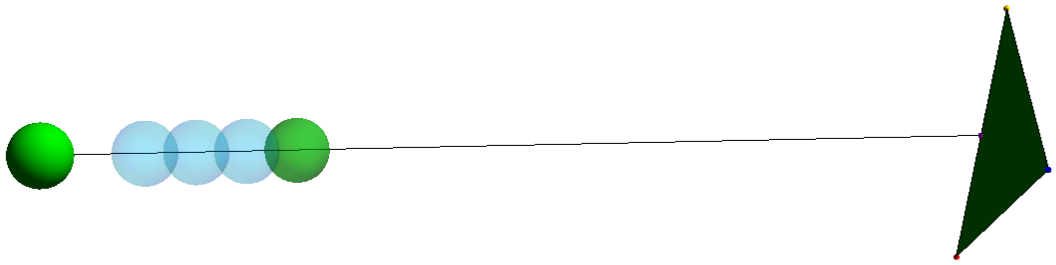
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

Plot fluoro vs. DRR - Green

```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined□True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```



Focal position - 25% closer to intensifier screen



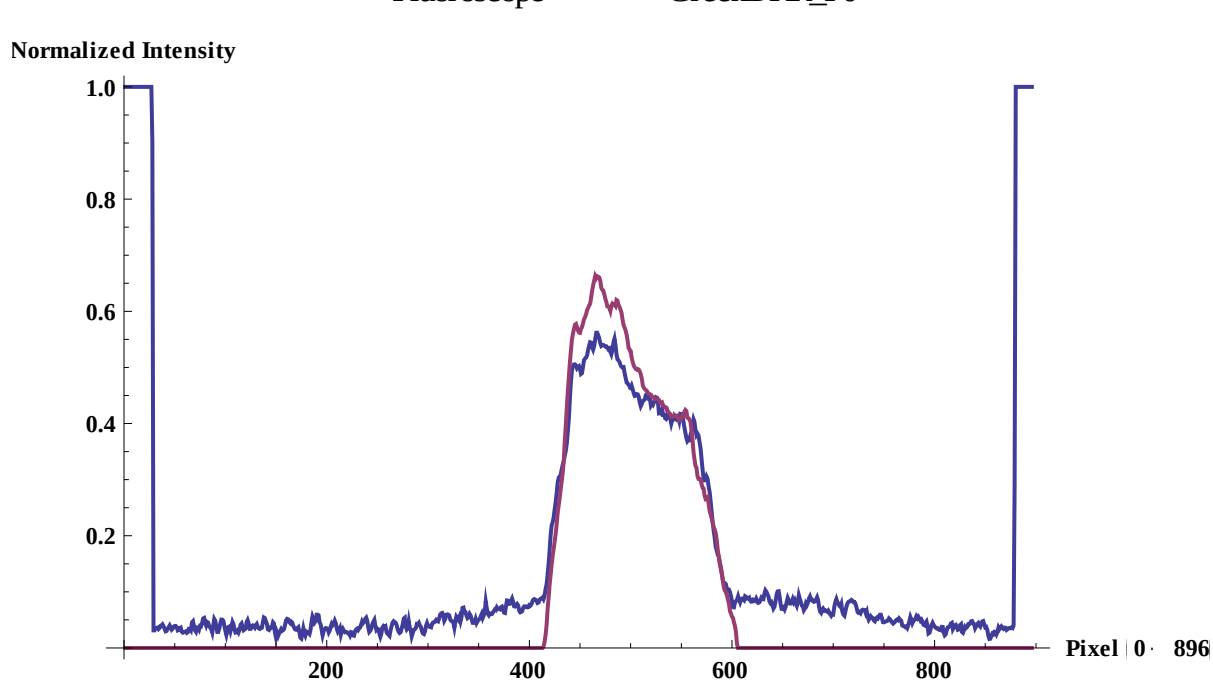
```
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vRC-3.0\\DRRACC v2"];
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vBeta\\DRRACC v2\\stage
rotation"];
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

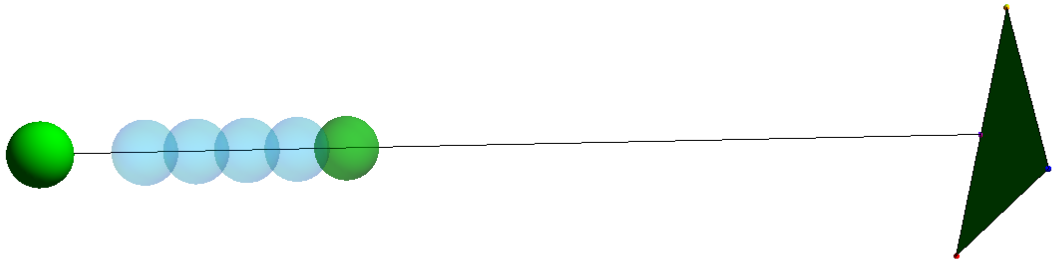
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

Plot fluoro vs. DRR - Green

```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined→True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```



Focal position - 30% closer to intensifier screen



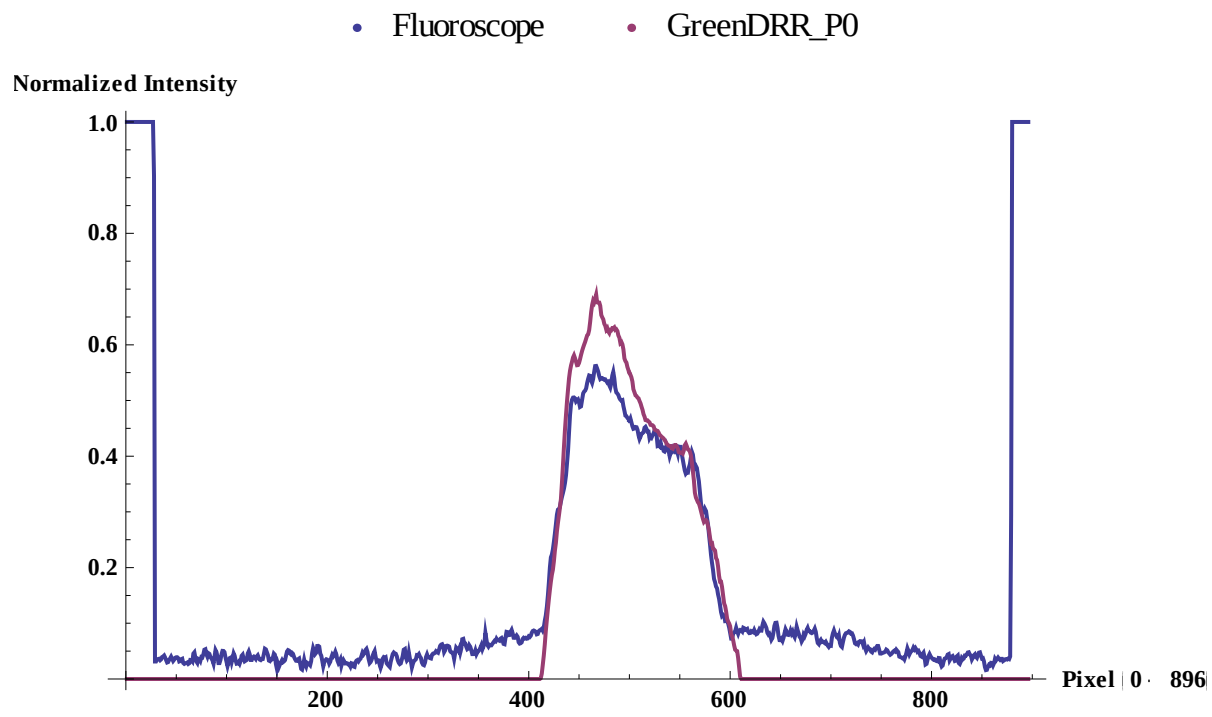
```
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vRC-3.0\\DRRACC v2"];
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vBeta\\DRRACC v2\\stage
rotation"];
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

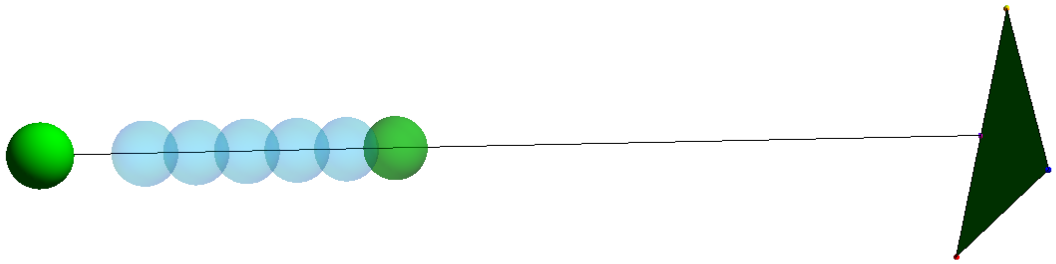
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

Plot fluoro vs. DRR - Green

```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined→True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```



Focal position - 35% closer to intensifier screen



```
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vRC-3.0\\DRRACC v2"];
pb0=Partition[BinaryReadList["blue.bin","Real32"],1152];
ib0=Image[pb0/Max[pb0]];
pg0=Partition[BinaryReadList["green.bin","Real32"],1152];
ig0=Image[pg0/Max[pg0]];
SetDirectory["C:\\Users\\Grant\\Desktop\\DRRACC vBeta\\DRRACC v2\\stage
rotation"];
```

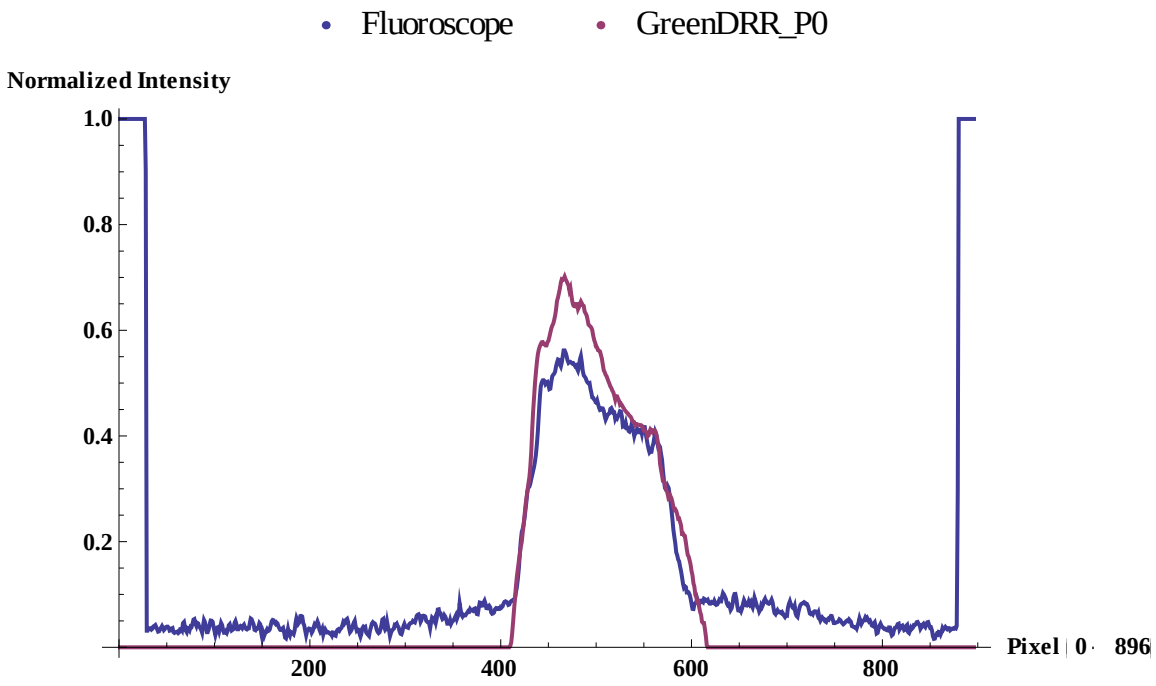
```
fb0=Partition[BinaryReadList["br0.raw","Real32"],1152];
ifb0=Image[fb0/Max[fb0]];
fg0=Partition[BinaryReadList["gr0.raw","Real32"],1152];
ifg0=Image[fg0/Max[fg0]];
```

Extract column 625 - Green

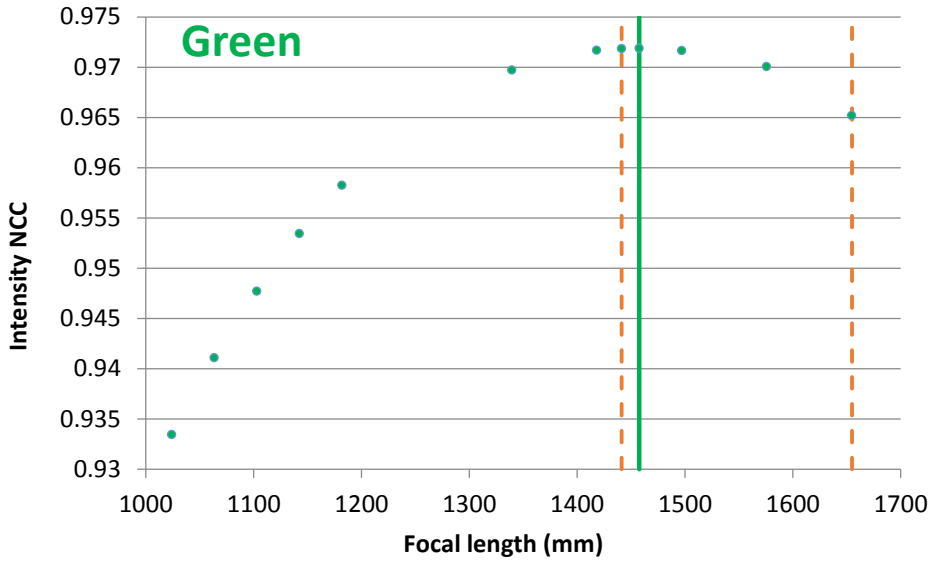
```
temp5=Take[pg0[[All,625]]]/Max[pg0];
temp6=Table[{i,temp5[[i]]},{i,1,Length[temp5]}];
temp7=Take[fg0[[All,625]]];
temp8=Table[{i,temp7[[i]]},{i,1,Length[temp7]}];
```

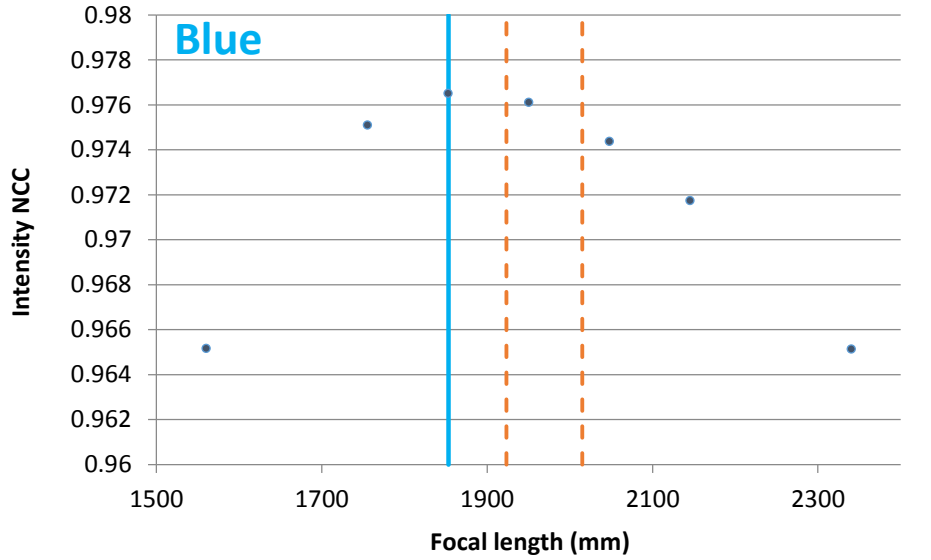
Plot fluoro vs. DRR - Green

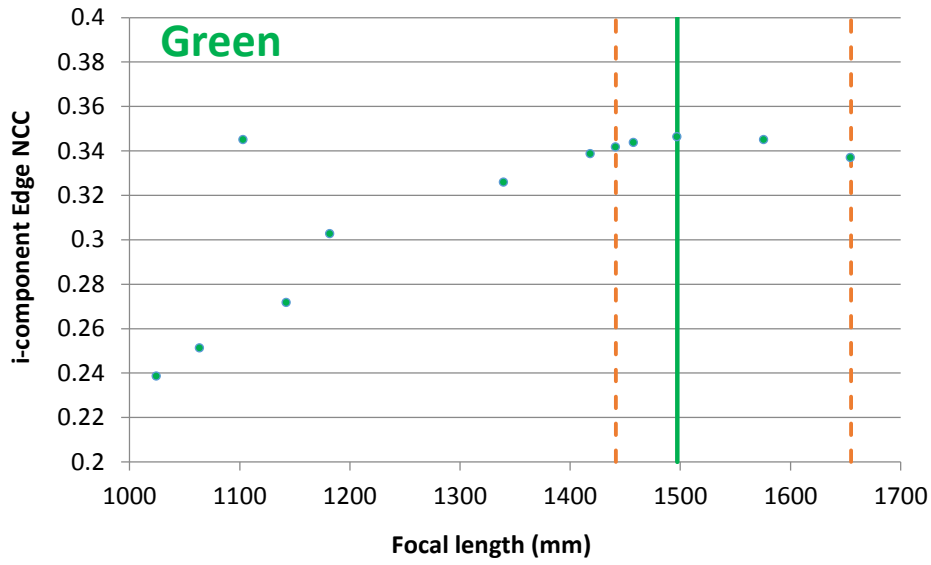
```
Show[ListPlot[{temp8,temp6},PlotLegends→Placed[{"Fluoroscope","GreenDRR_P0"}
,Above],PlotStyle→Thick,Joined→True],AxesLabel→{"Pixel (0 -
896)","Normalized Intensity"},LabelStyle→{Medium,Bold},PlotRange→{0,1}]
```

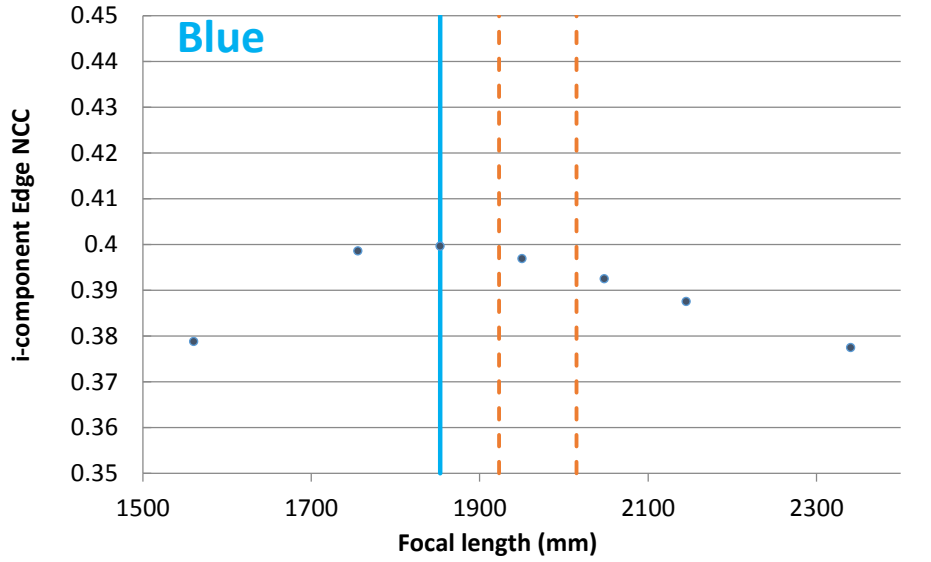


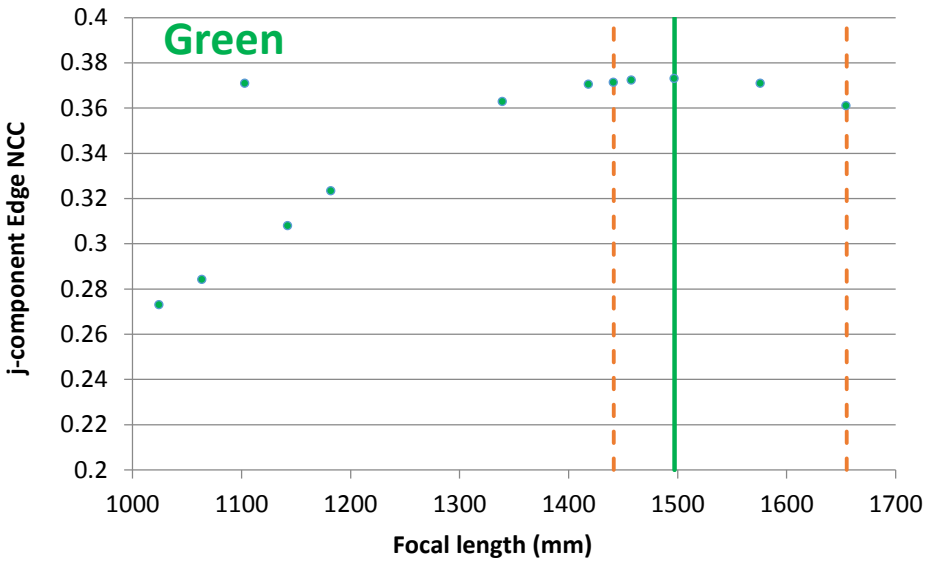
Appendix H: NCC Sensitivity to Focal Length

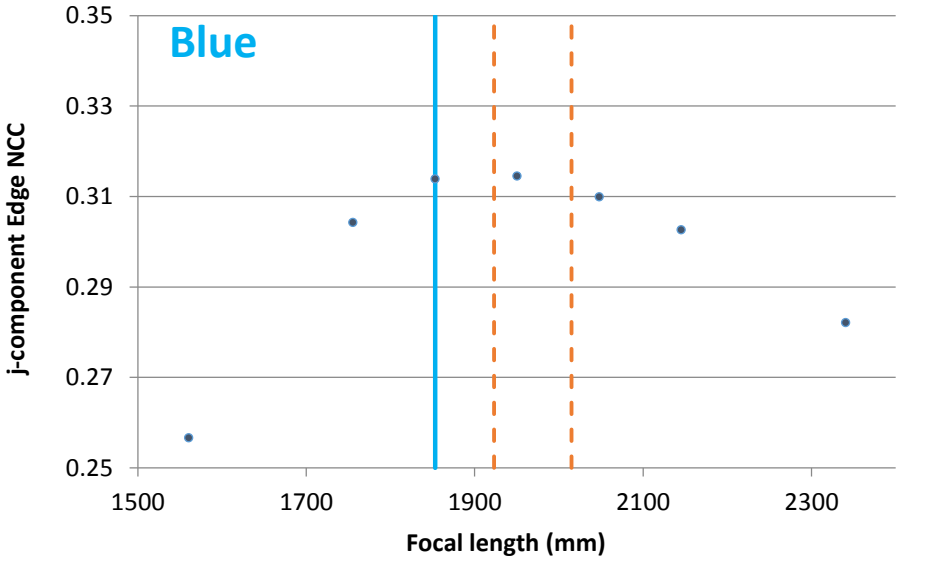





Green
i-component Edge NCC
0.4
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0.2
1000
1100
1200
1300
1400
1500
1600
1700
Focal length (mm)


Blue
i-component Edge NCC
0.45
0.44
0.43
0.42
0.41
0.4
0.39
0.38
0.37
0.36
0.35
1500
1700
1900
2100
2300
Focal length (mm)

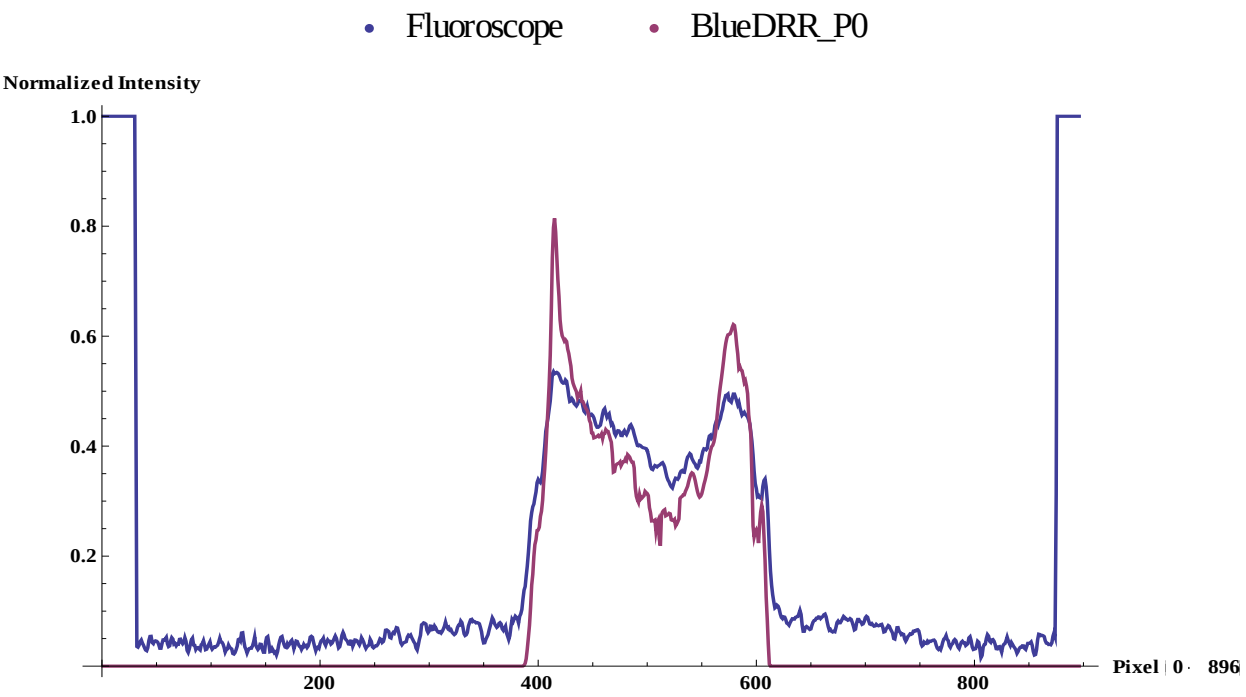

Green
j-component Edge NCC
0.4
0.38
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0.2
1000
1100
1200
1300
1400
1500
1600
1700
Focal length (mm)


Blue
j-component Edge NCC
0.35
0.33
0.31
0.29
0.27
0.25
1500
1700
1900
2100
2300
Focal length (mm)

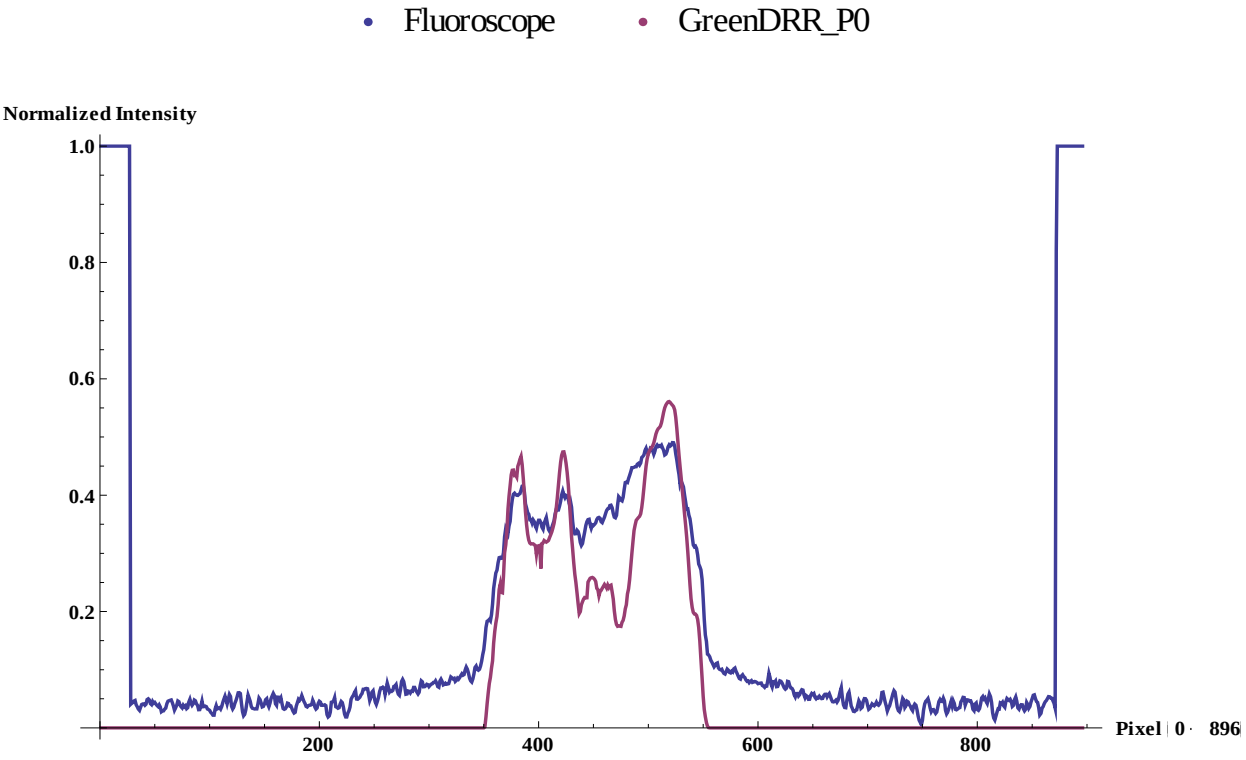
Appendix I: Stage Translation Profile Plots

P0

Plot fluoro vs. DRR - Blue

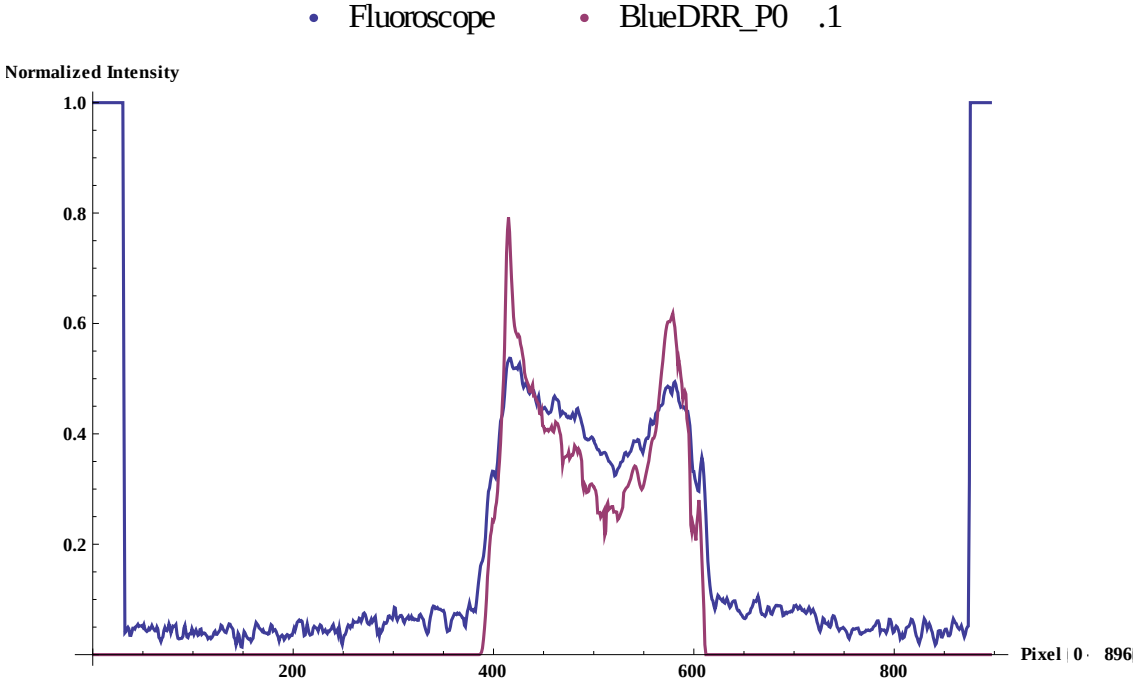


Plot fluoro vs. DRR - Green

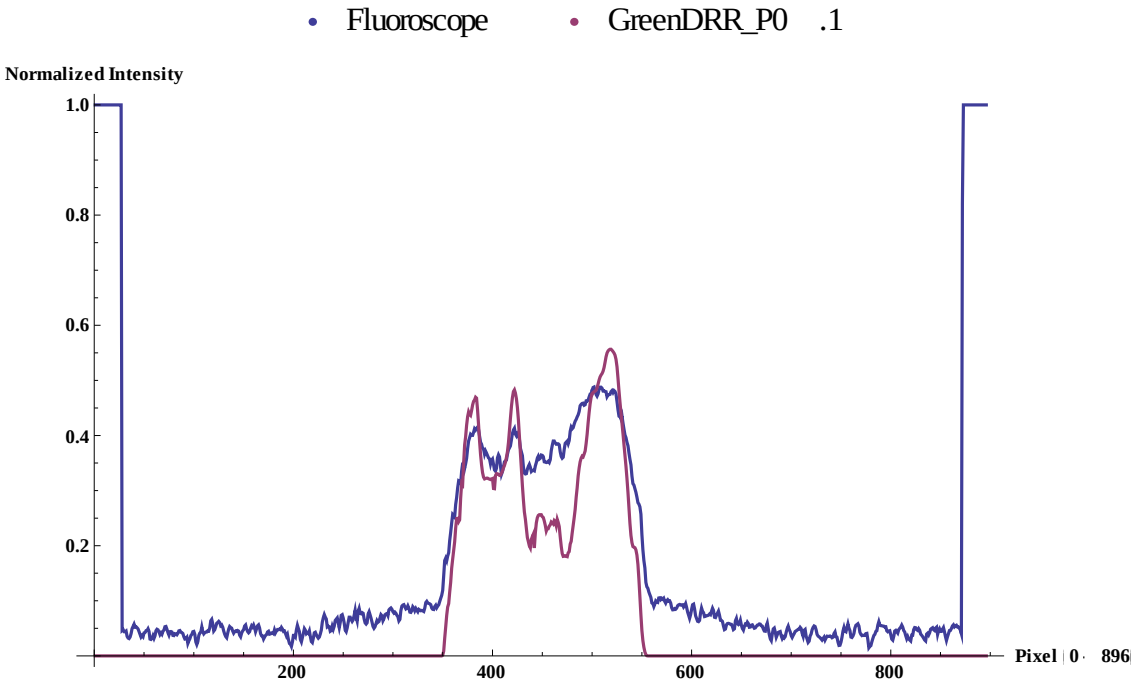


P0.1

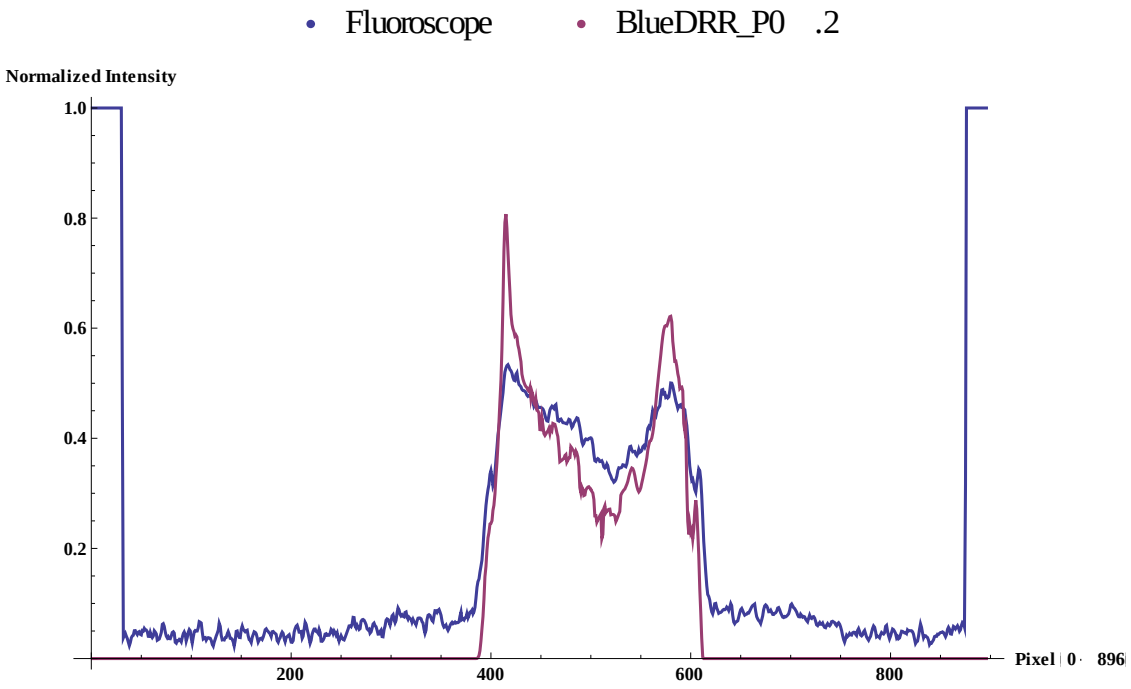
Plot fluoro vs. DRR - Blue



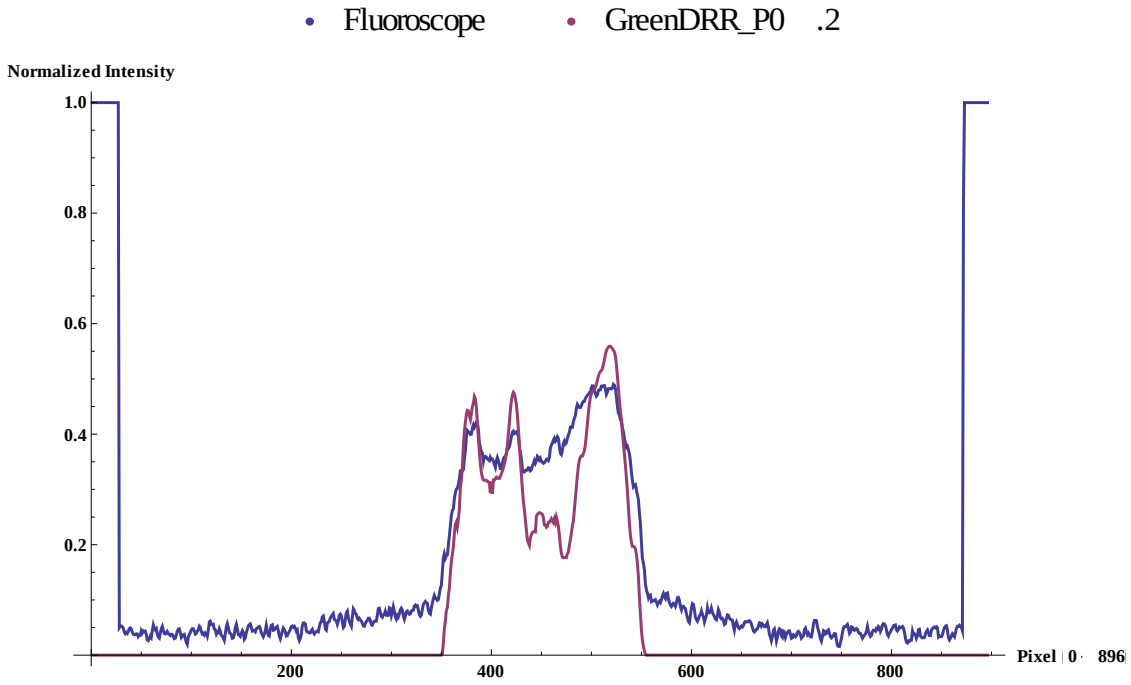
Plot fluoro vs. DRR - Green



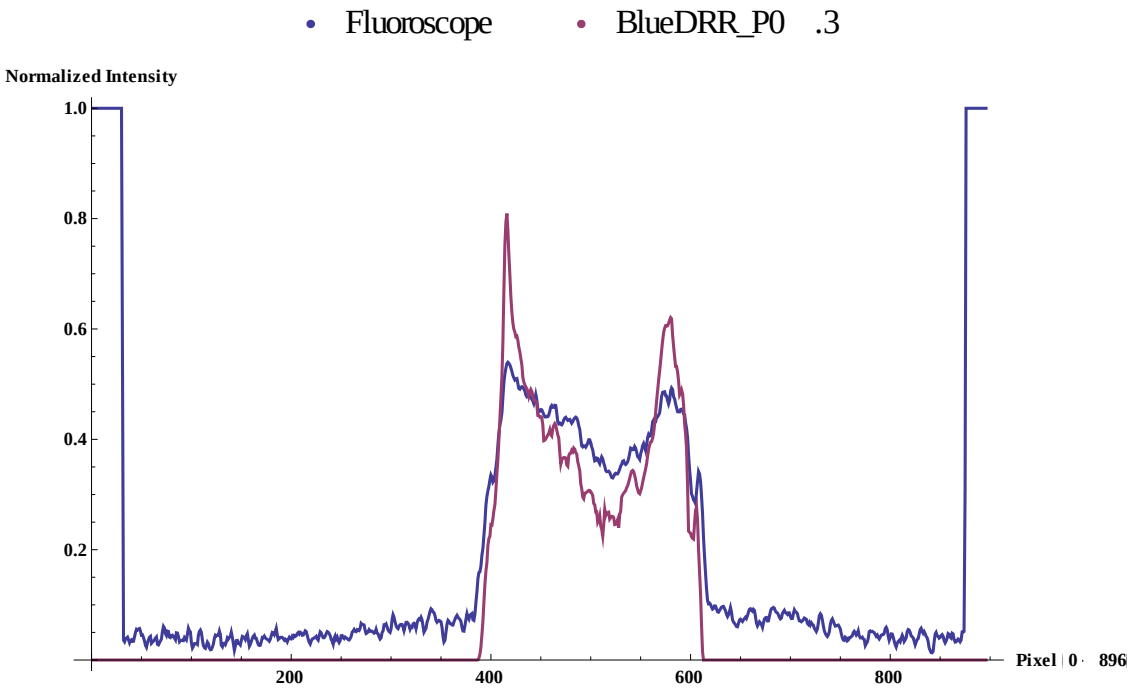
P0.2
Plot fluoro vs. DRR - Blue



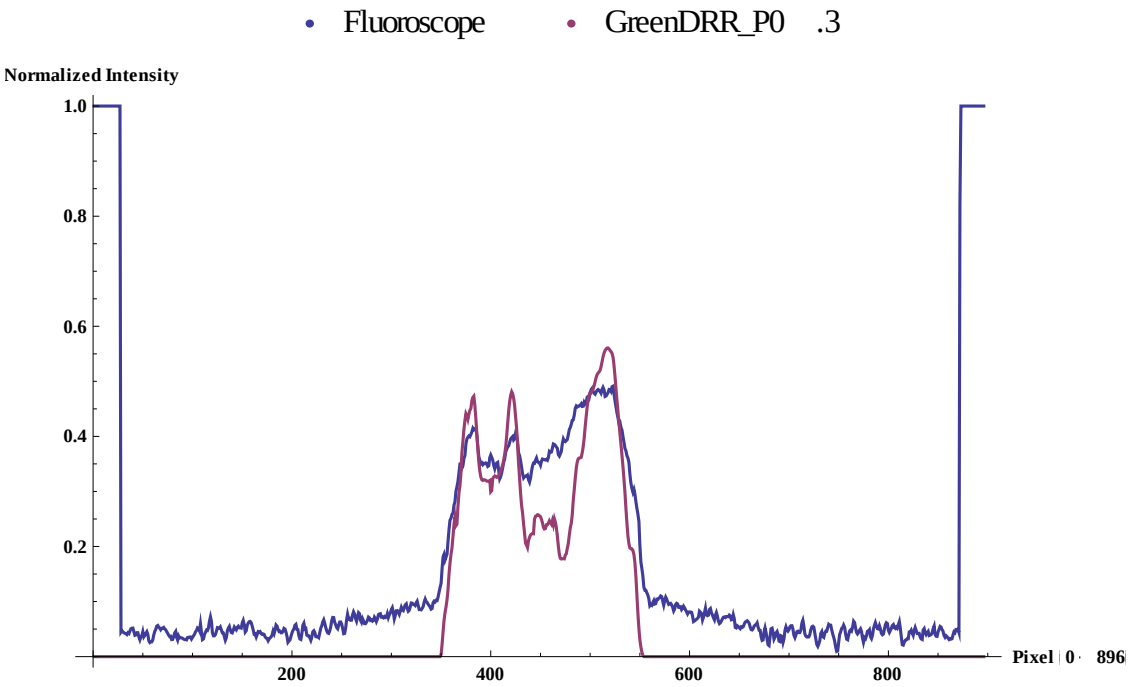
Plot fluoro vs. DRR - Green



P0.3
Plot fluoro vs. DRR - Blue

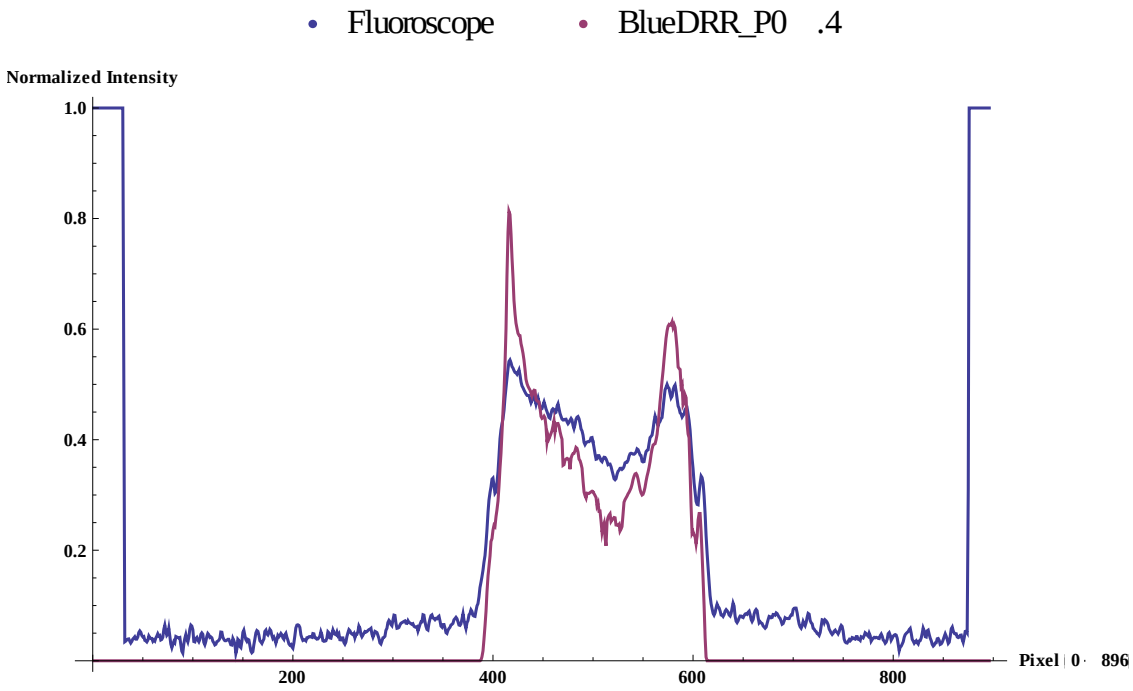


Plot fluoro vs. DRR - Green

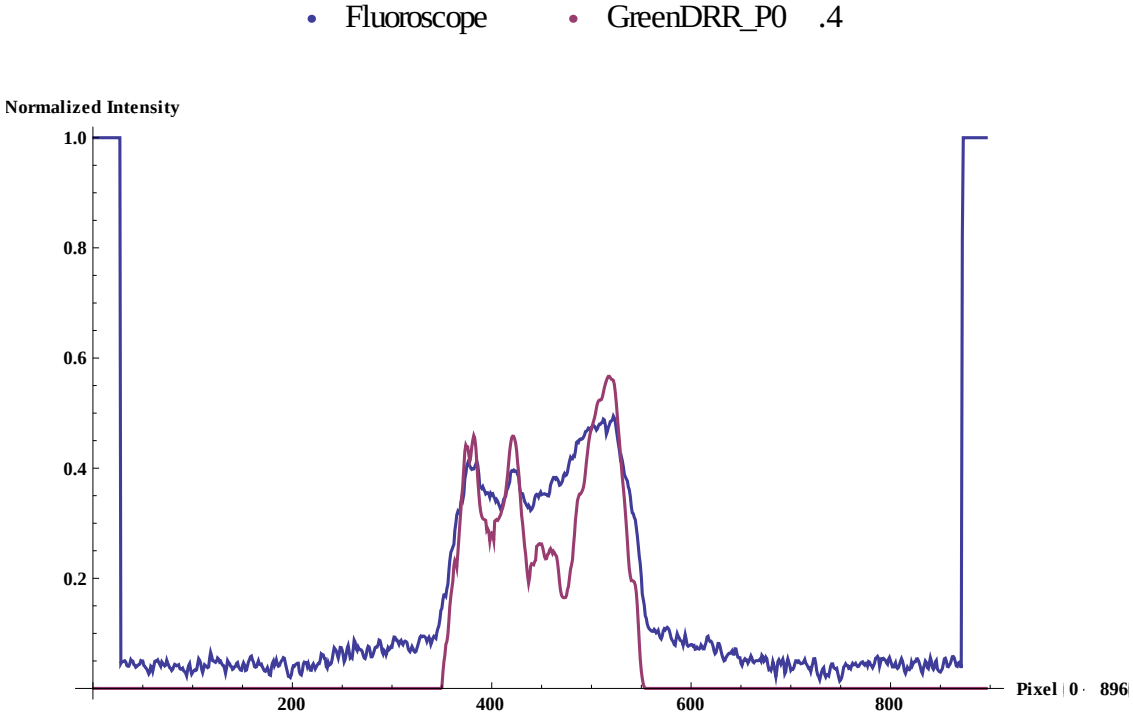


P0.4

Plot fluoro vs. DRR - Blue

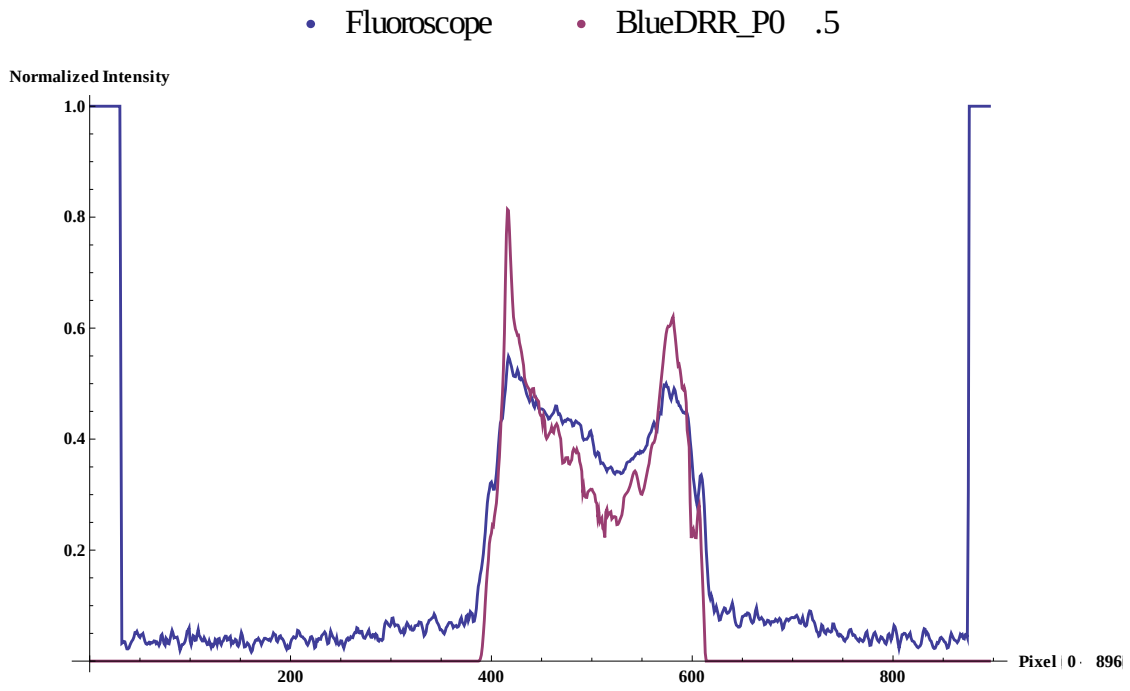


Plot fluoro vs. DRR - Green

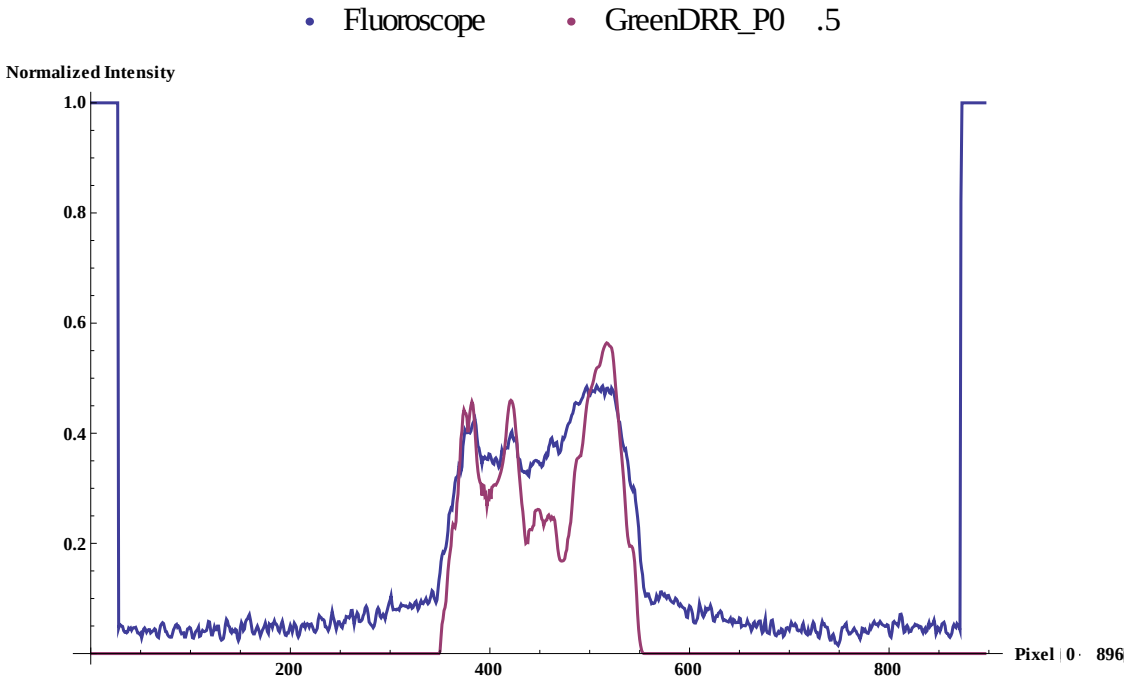


P0.5

Plot fluoro vs. DRR - Blue

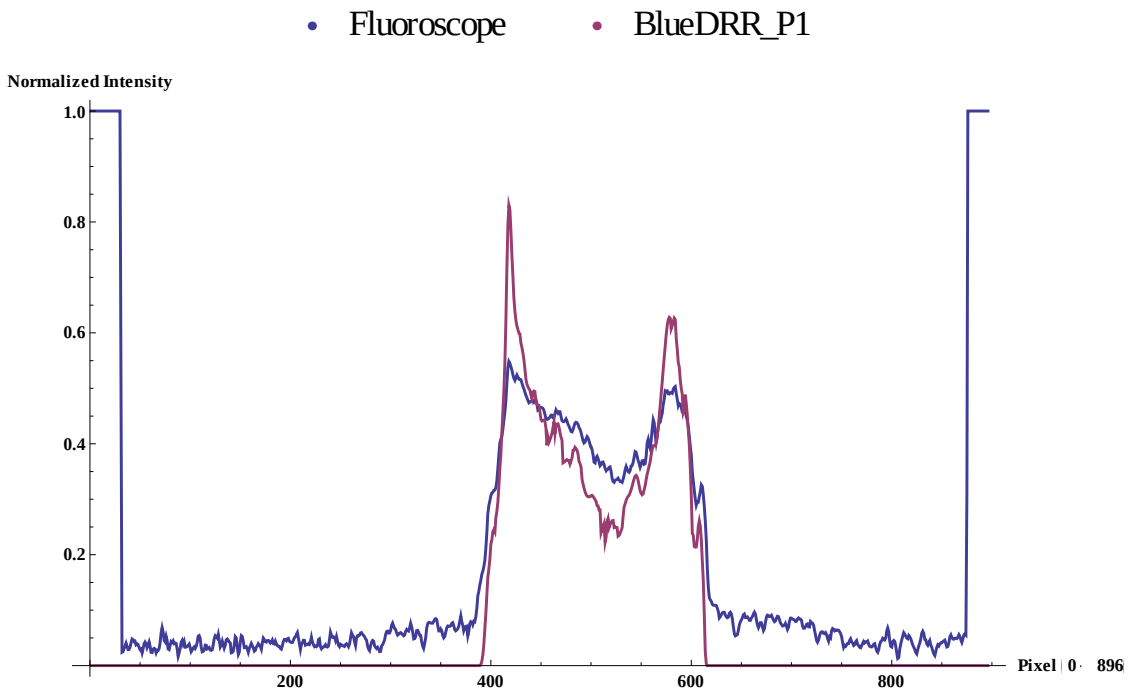


Plot fluoro vs. DRR - Green

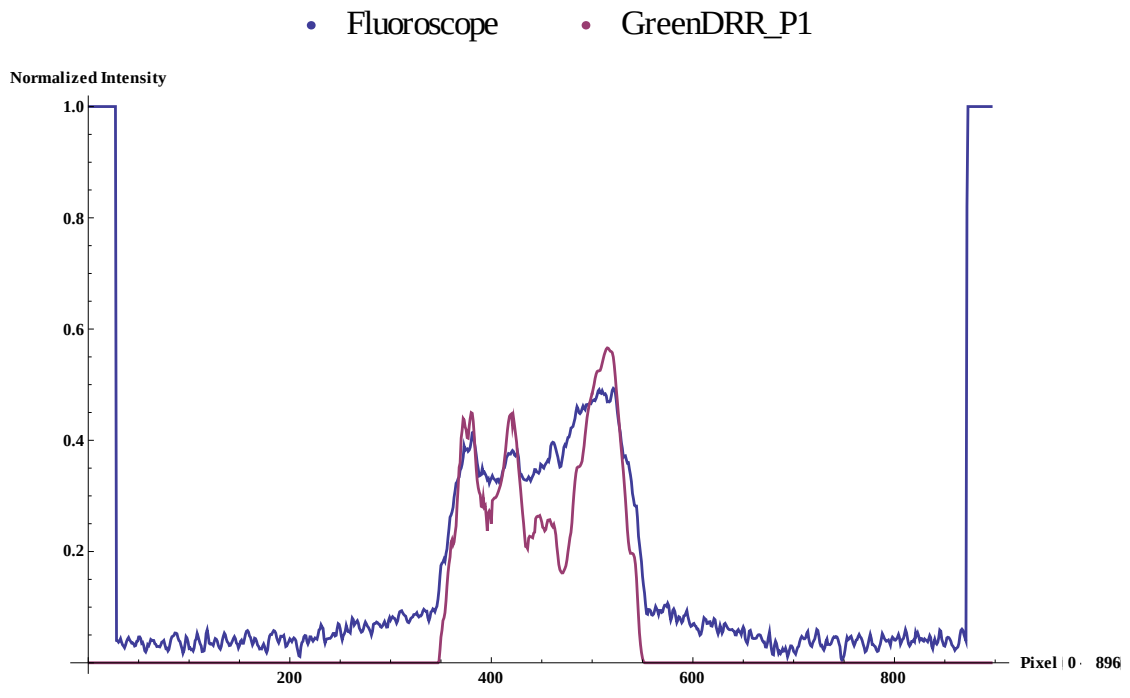


P1

Plot fluoro vs. DRR - Blue

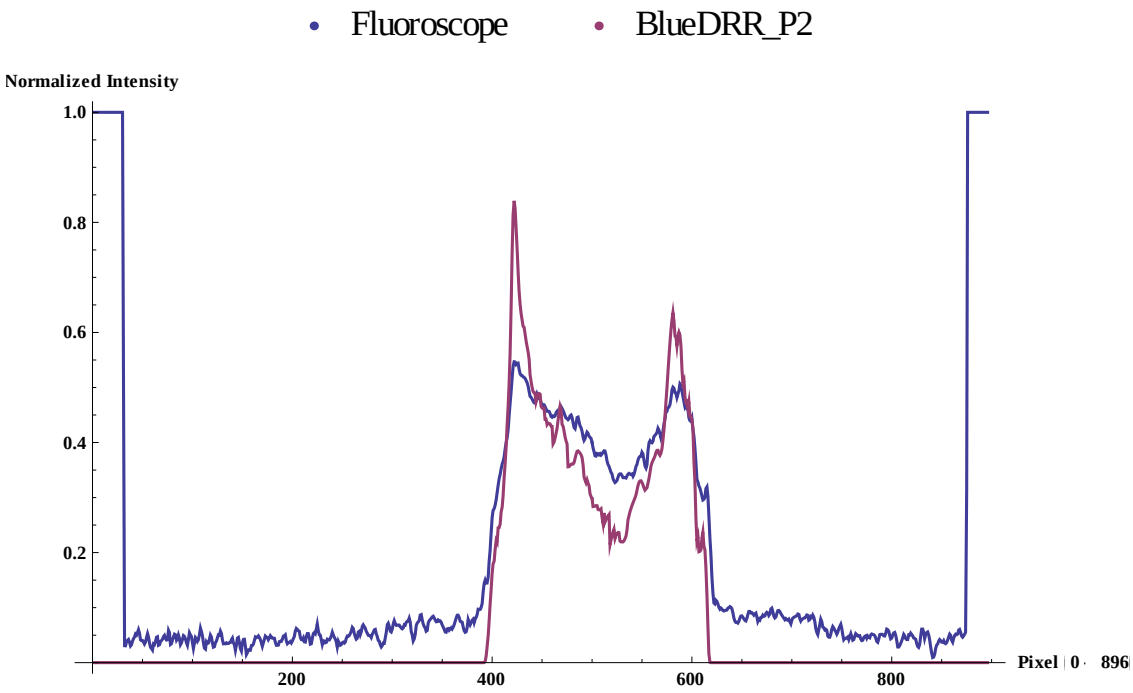


Plot fluoro vs. DRR - Green

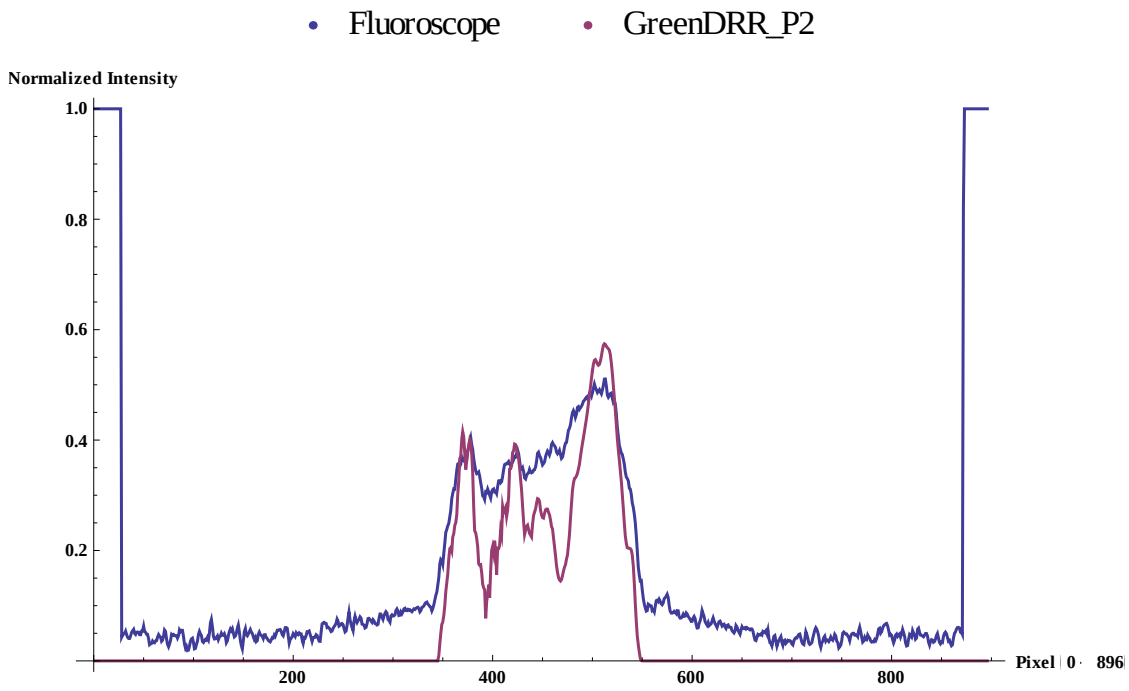


P2

Plot fluoro vs. DRR - Blue

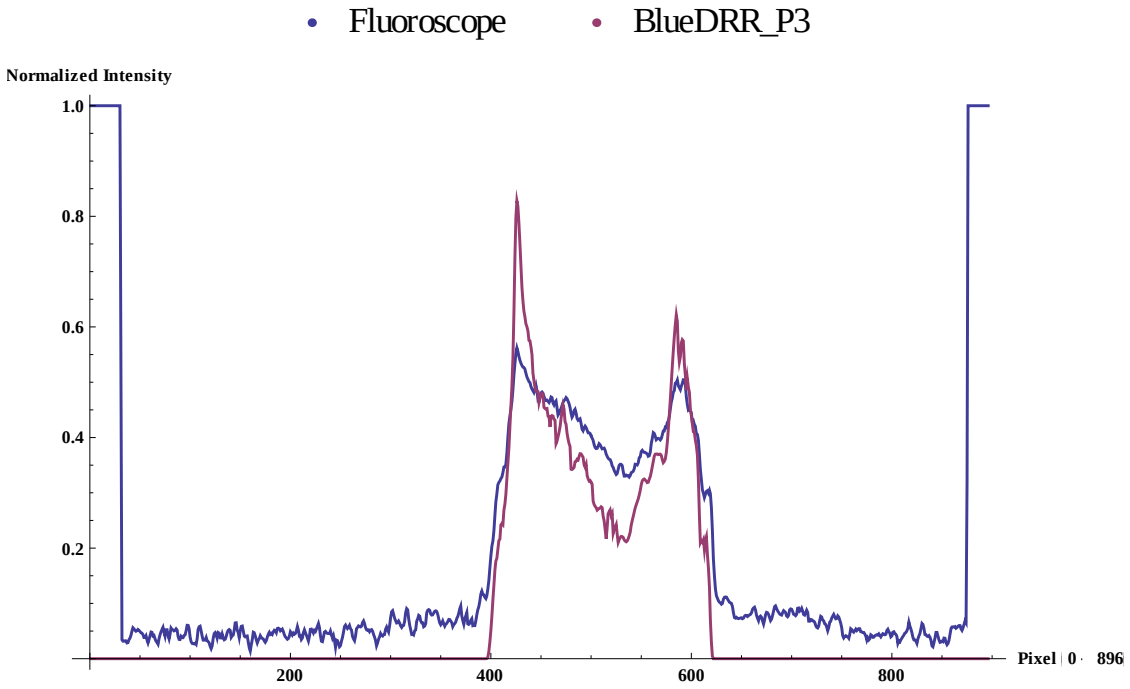


Plot fluoro vs. DRR - Green

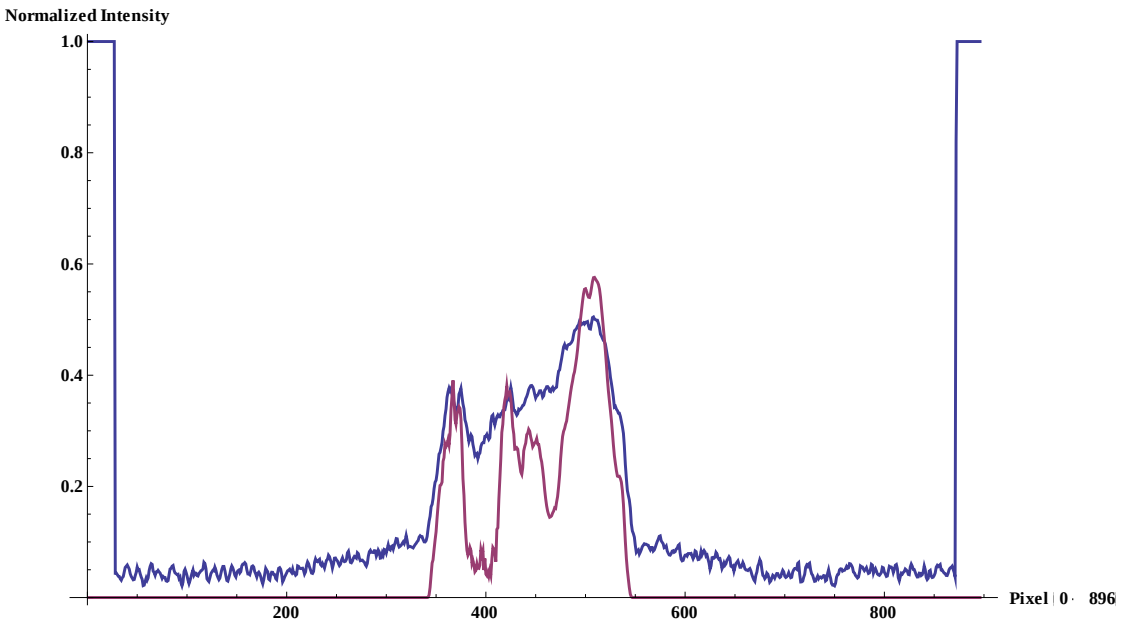


P3

Plot fluoro vs. DRR - Blue

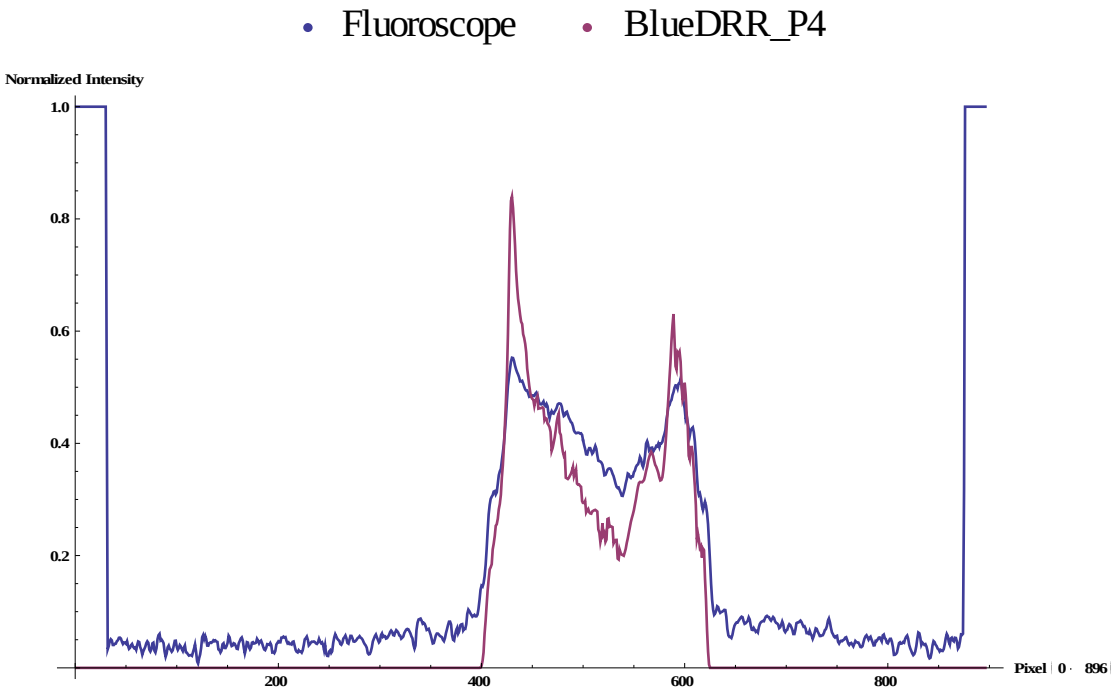


Plot fluoro vs. DRR - Green

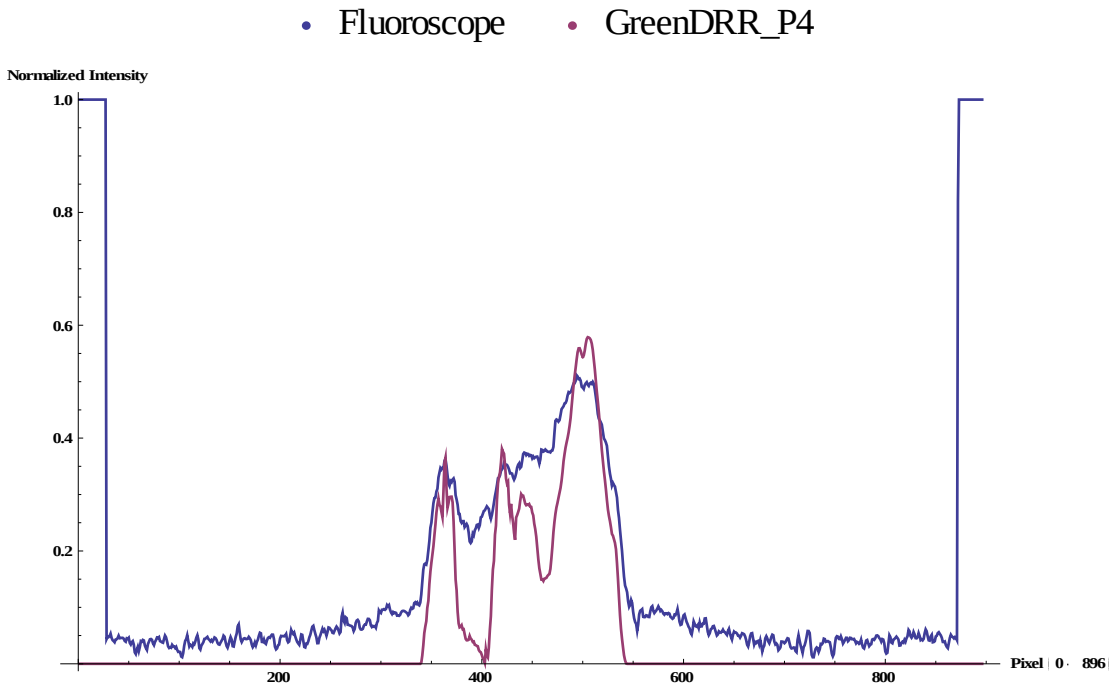


P4

Plot fluoro vs. DRR - Blue

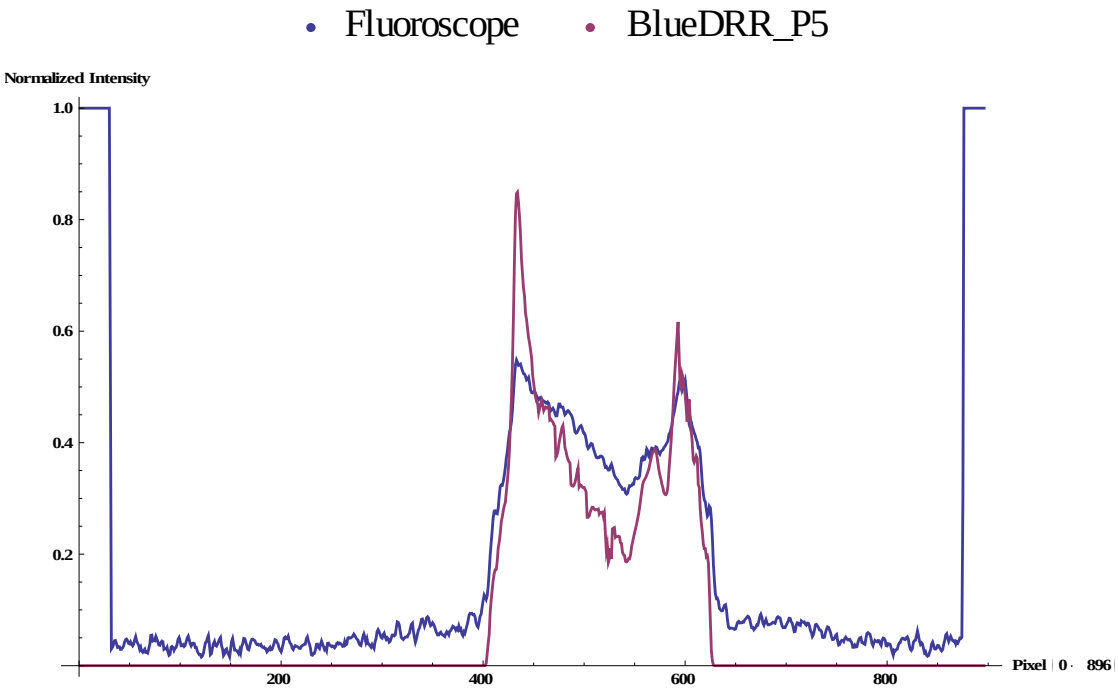


Plot fluoro vs. DRR - Green

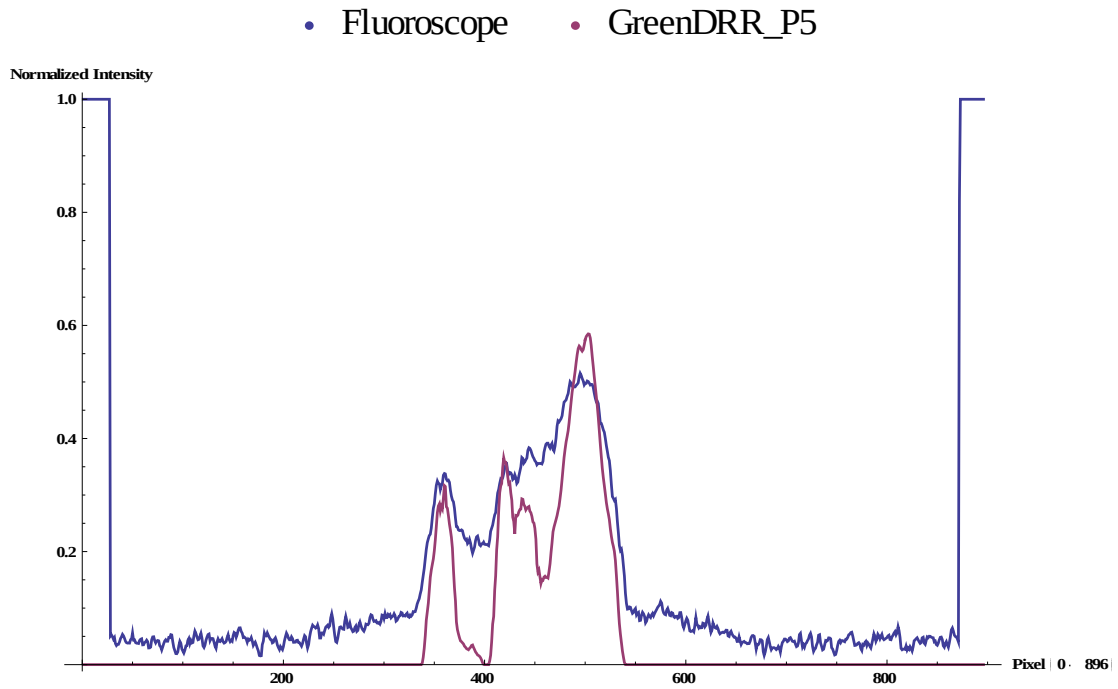


P5

Plot fluoro vs. DRR - Blue

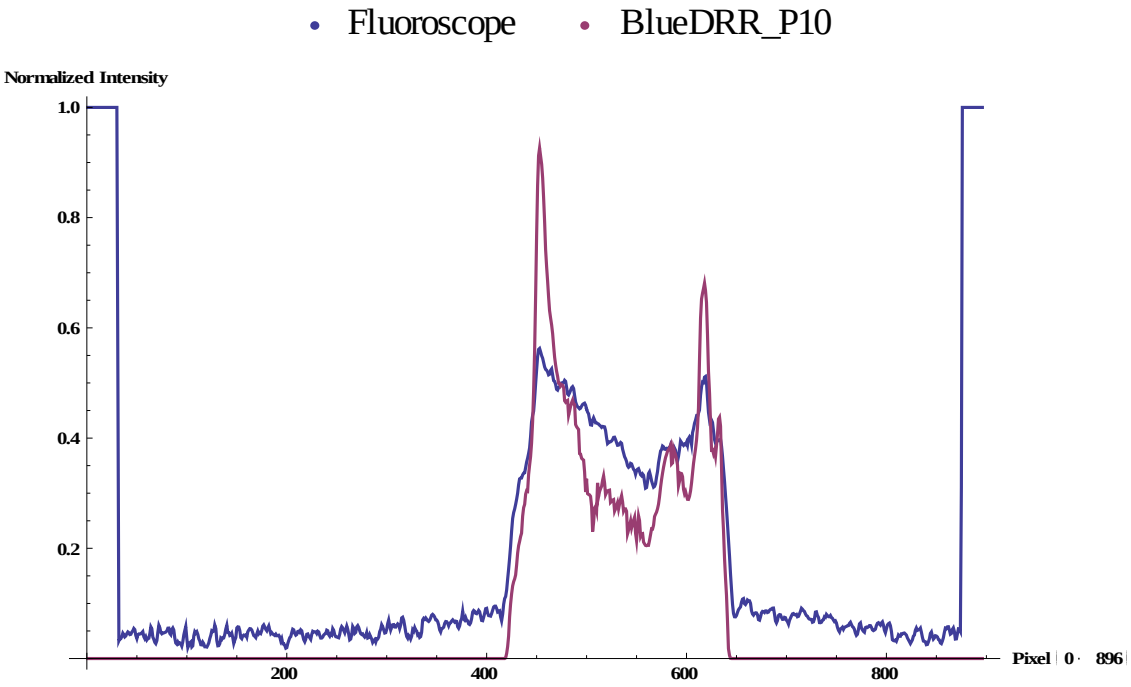


Plot fluoro vs. DRR - Green

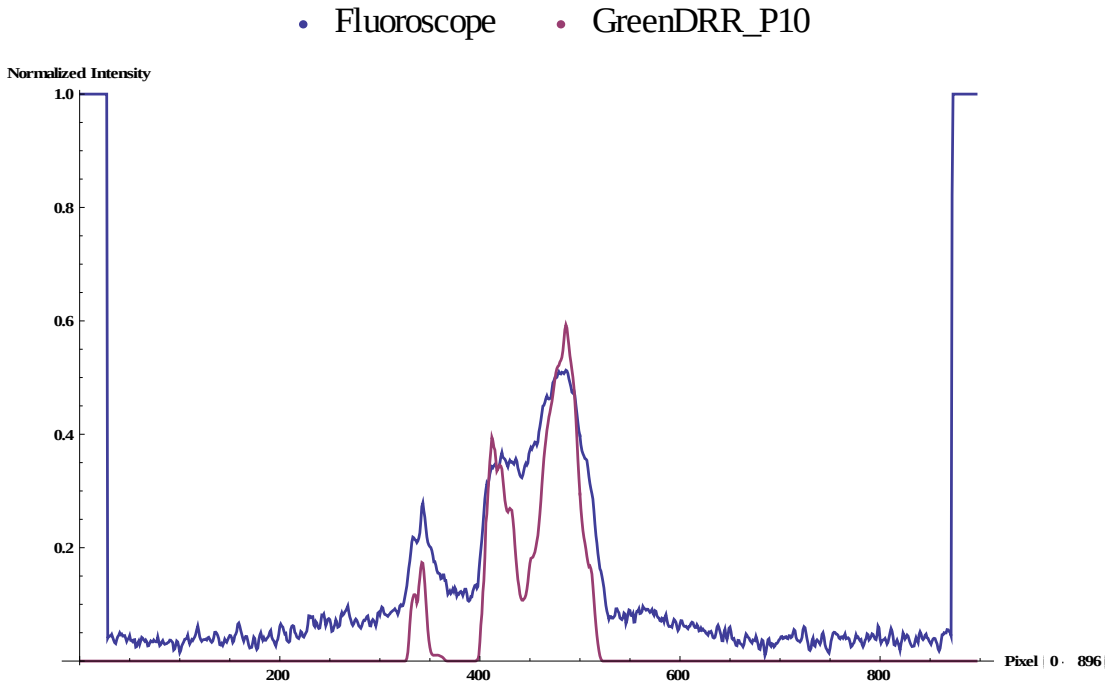


P10

Plot fluoro vs. DRR - Blue

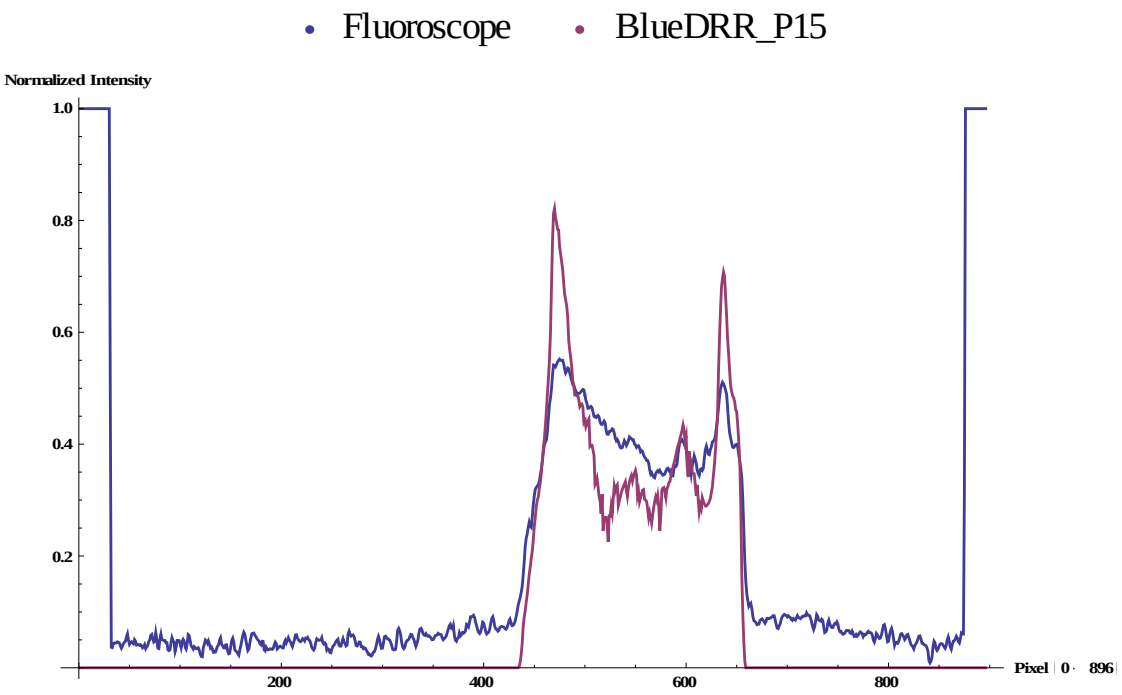


Plot fluoro vs. DRR - Green

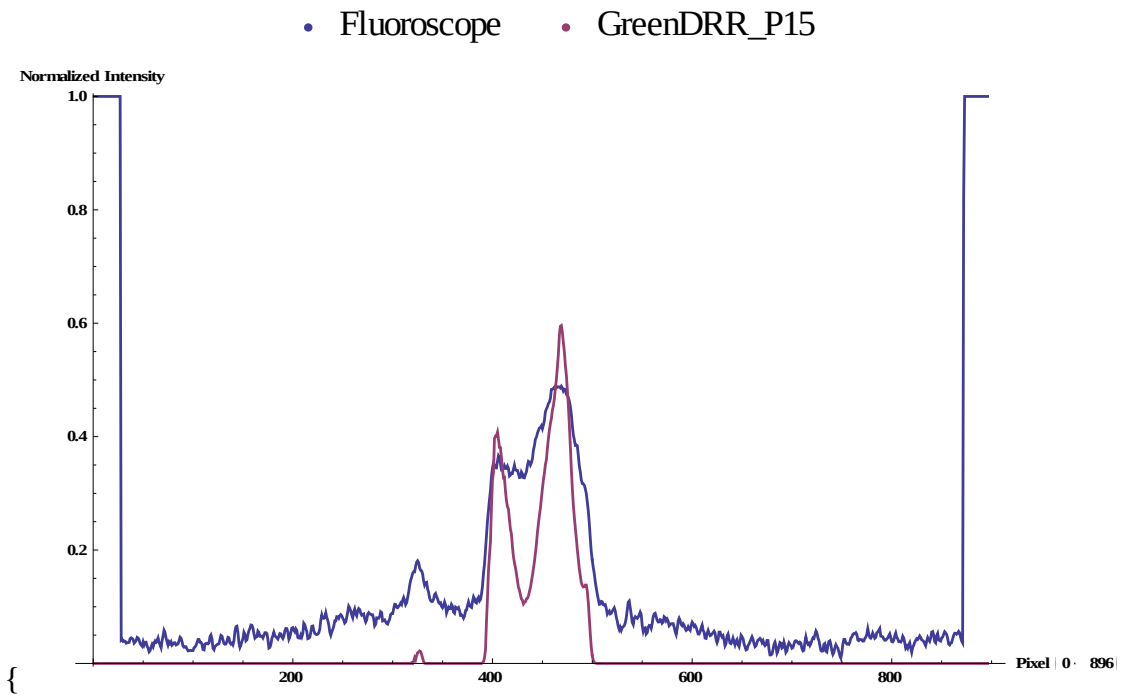


P15

Plot fluoro vs. DRR - Blue



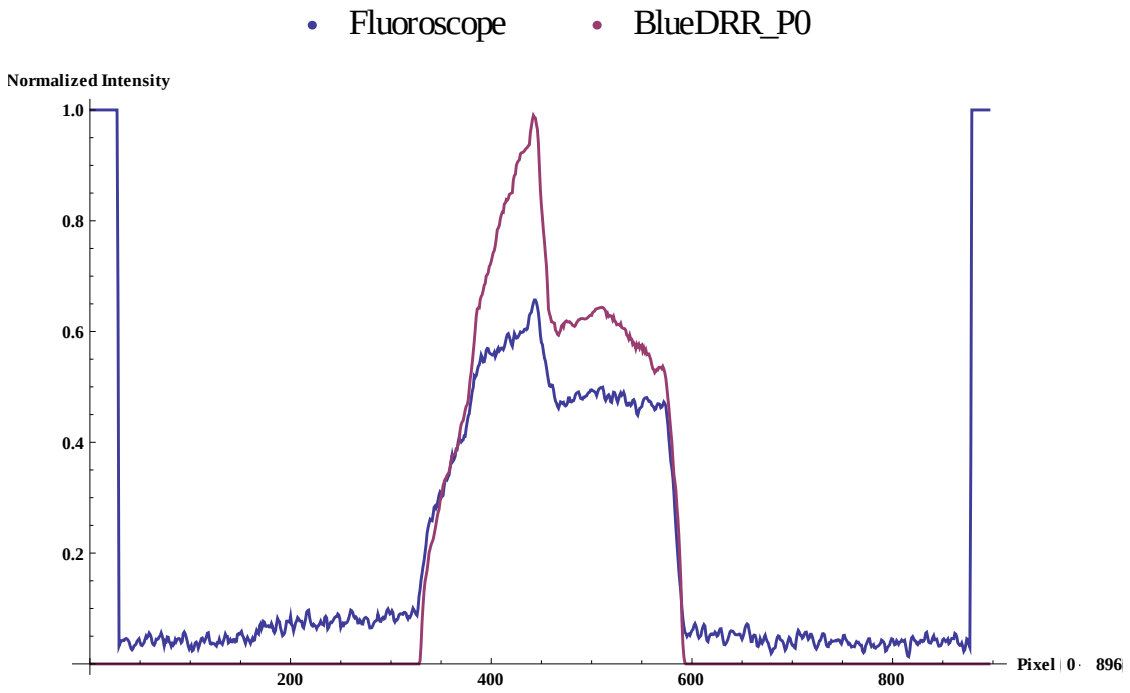
Plot fluoro vs. DRR - Green

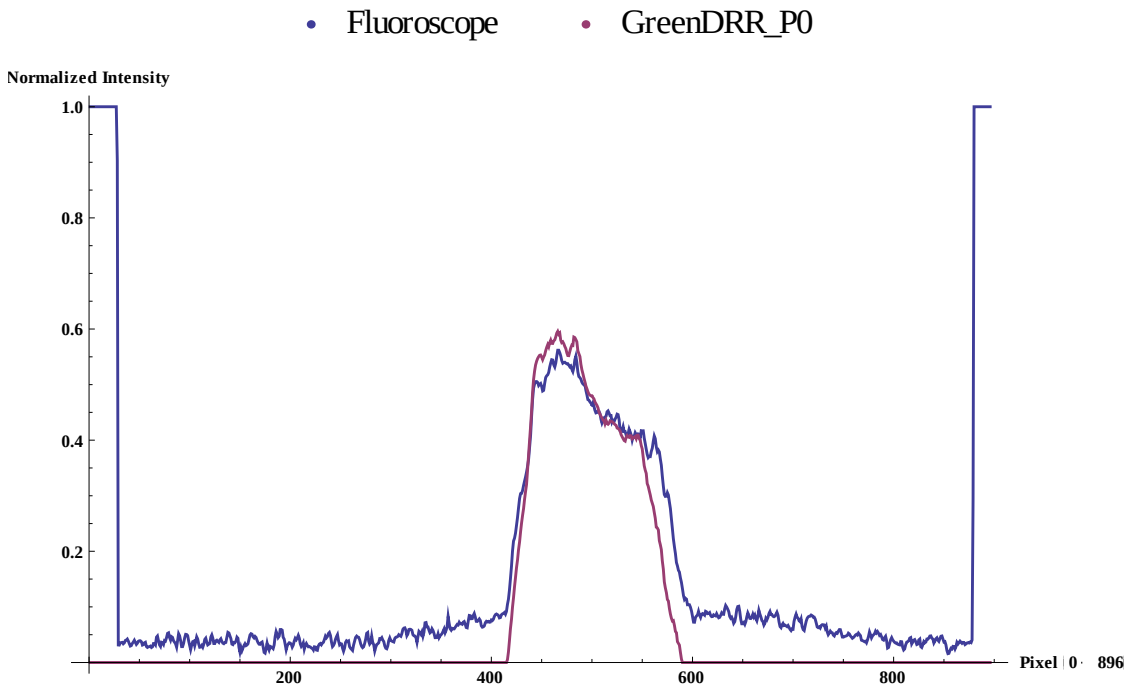


{

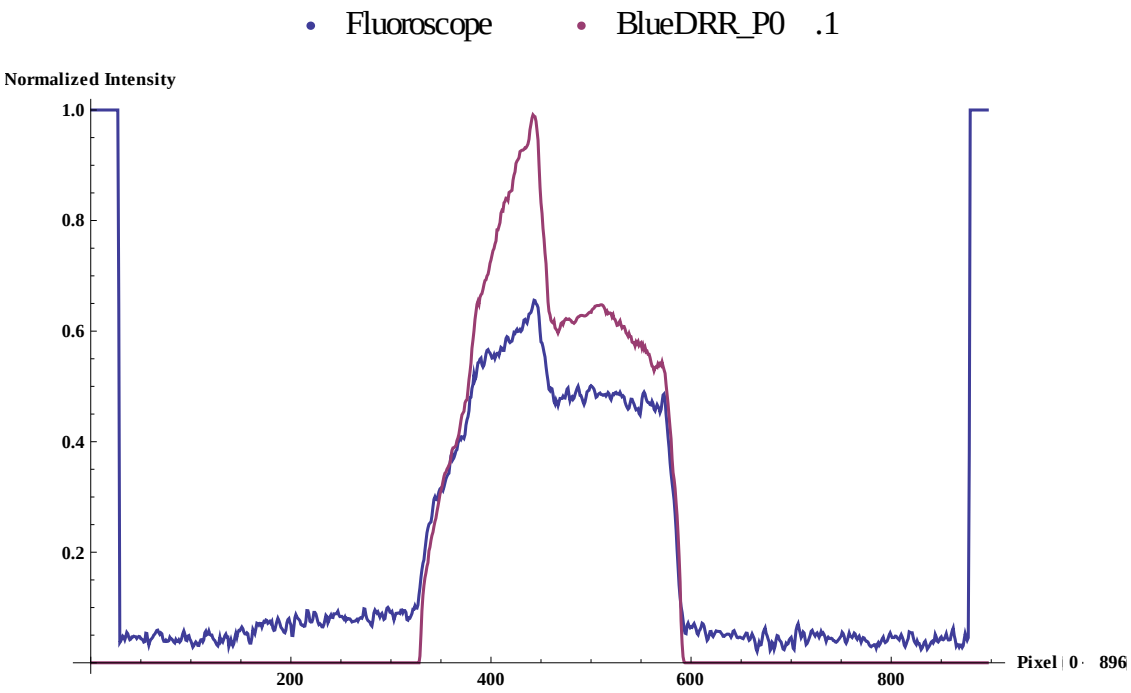
Appendix J: Stage Rotation Profile Plots

0 degrees
Plot fluoro vs. DRR - Blue

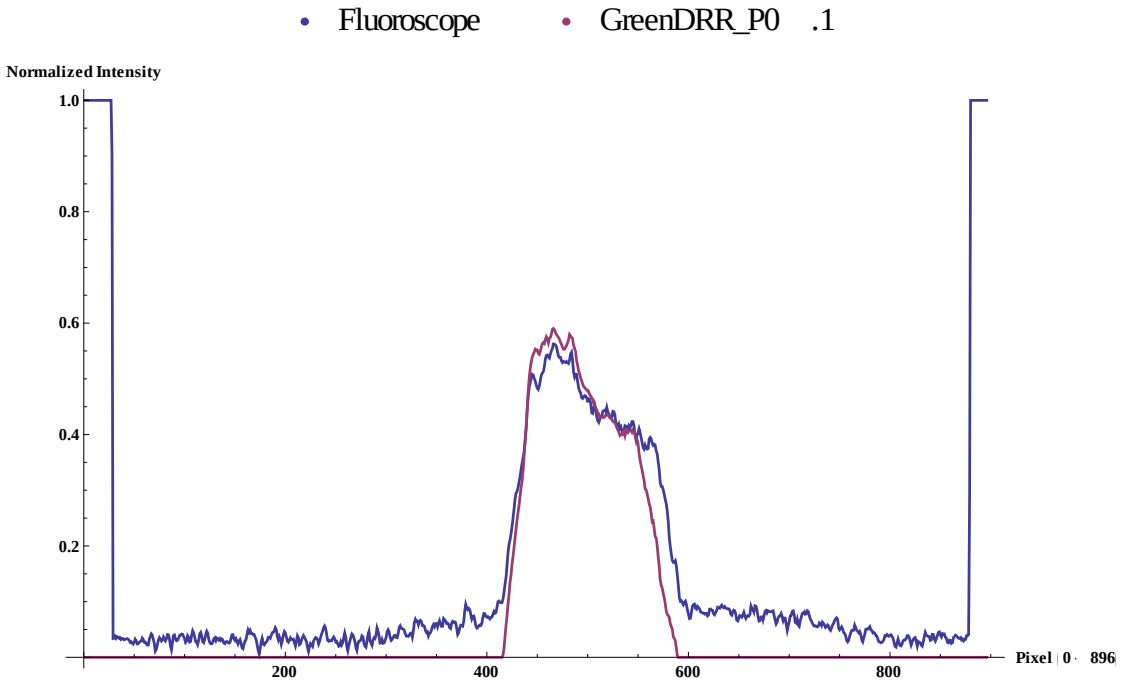




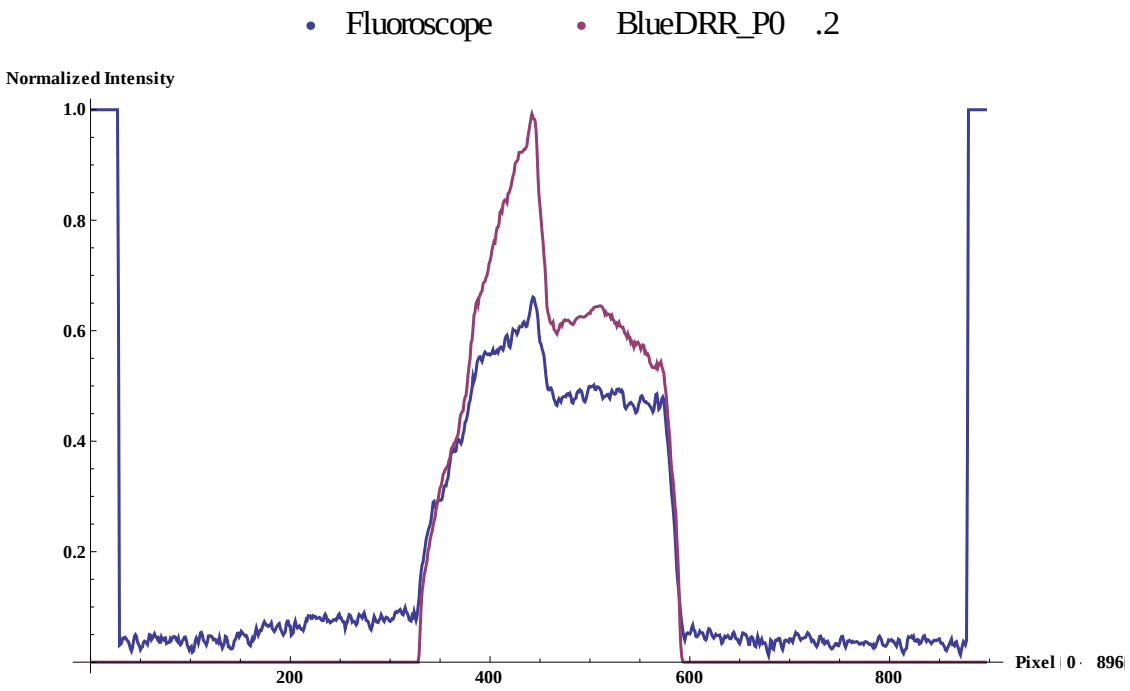
0.1 degrees
Plot fluoro vs. DRR - Blue



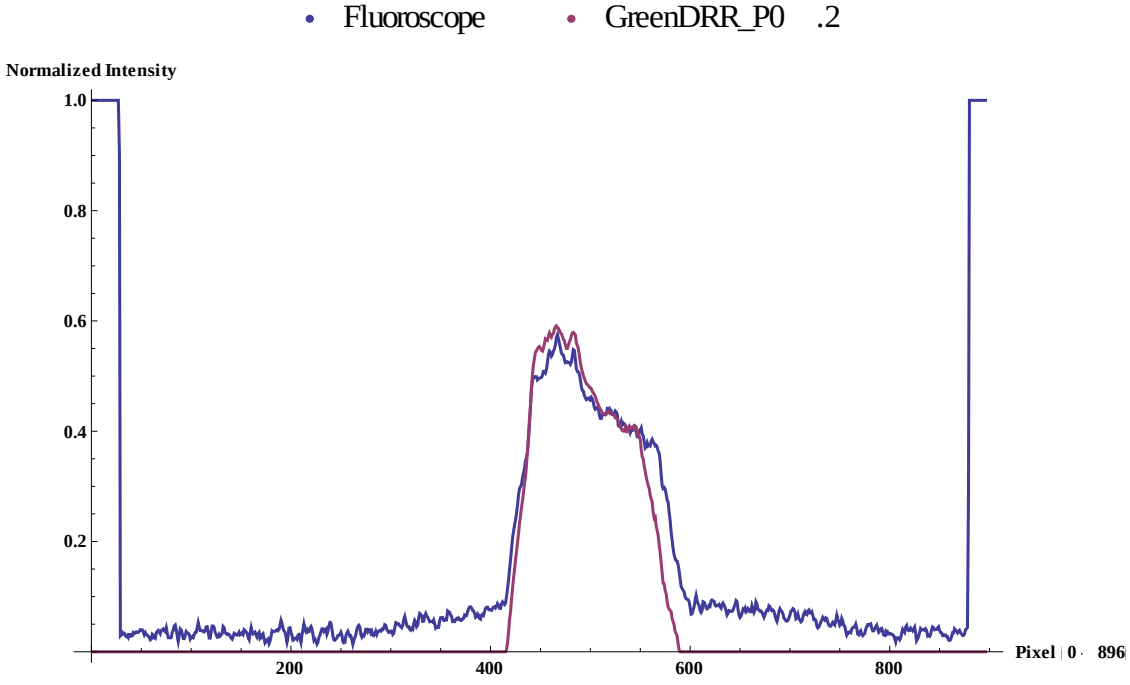
Plot fluoro vs. DRR - Green



0.2 degrees
Plot fluoro vs. DRR - Blue

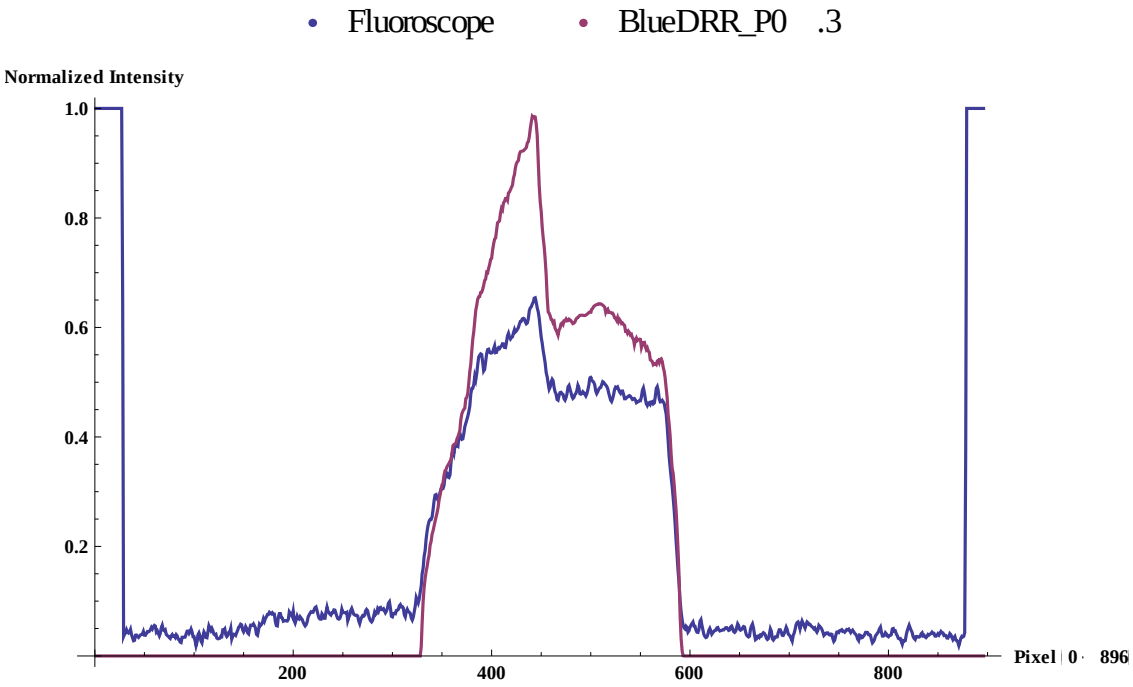


Plot fluoro vs. DRR - Green

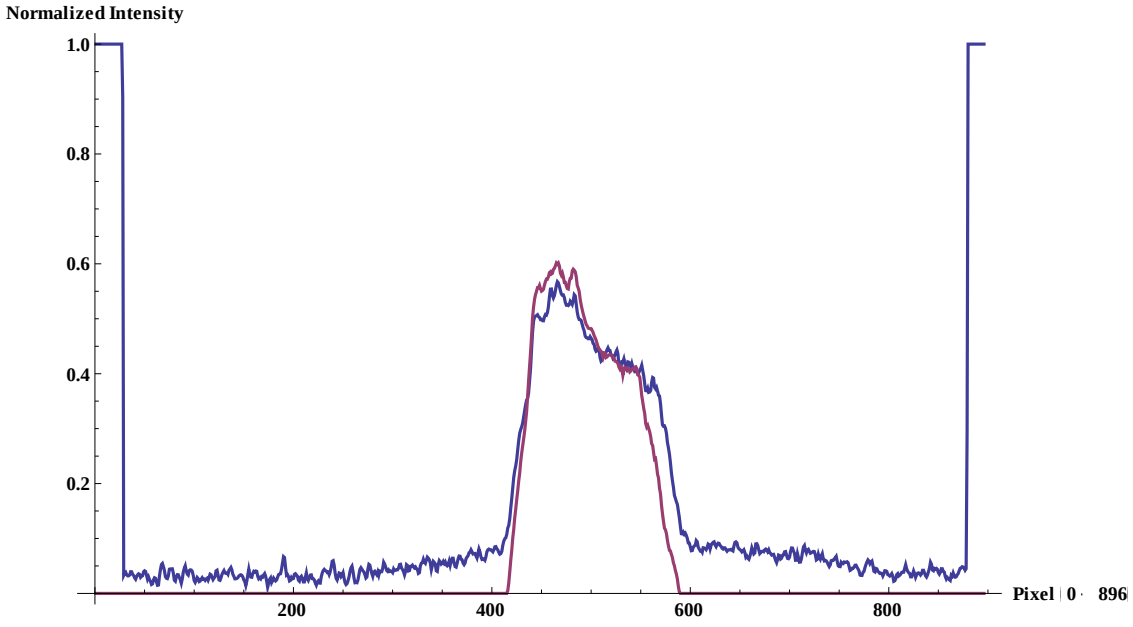


0.3 degrees

Plot fluoro vs. DRR - Blue

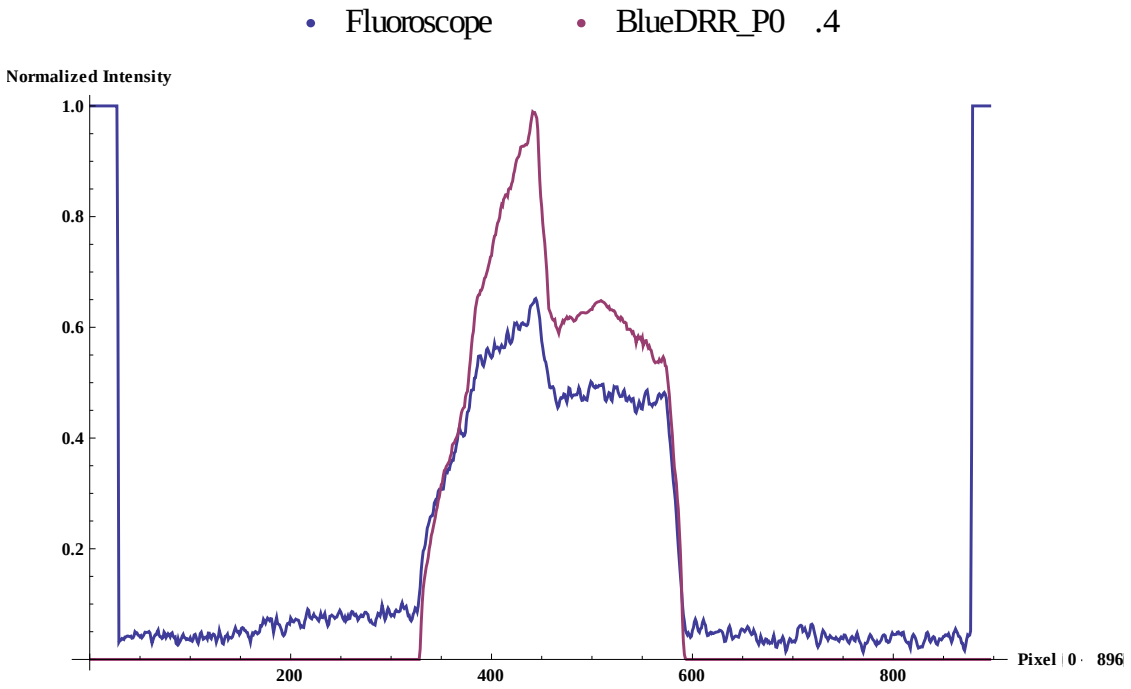


Plot fluoro vs. DRR - Green

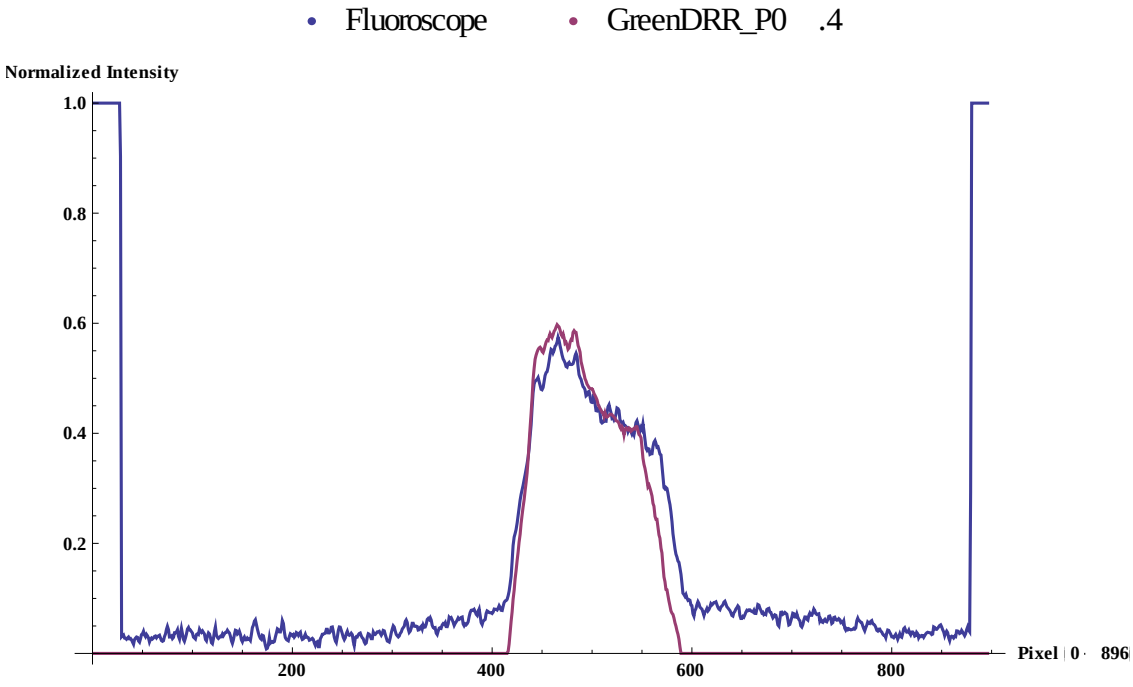


0.4 degrees

Plot fluoro vs. DRR - Blue

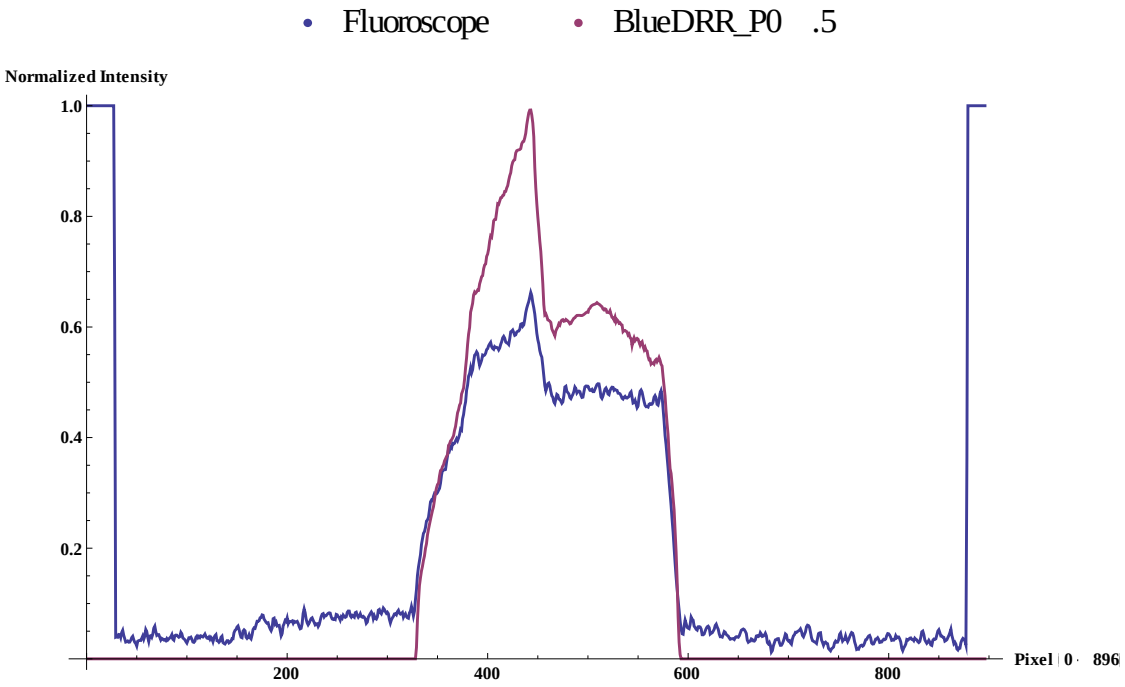


Plot fluoro vs. DRR - Green



0.5 degrees

Plot fluoro vs. DRR - Blue



Plot fluoro vs. DRR - Green

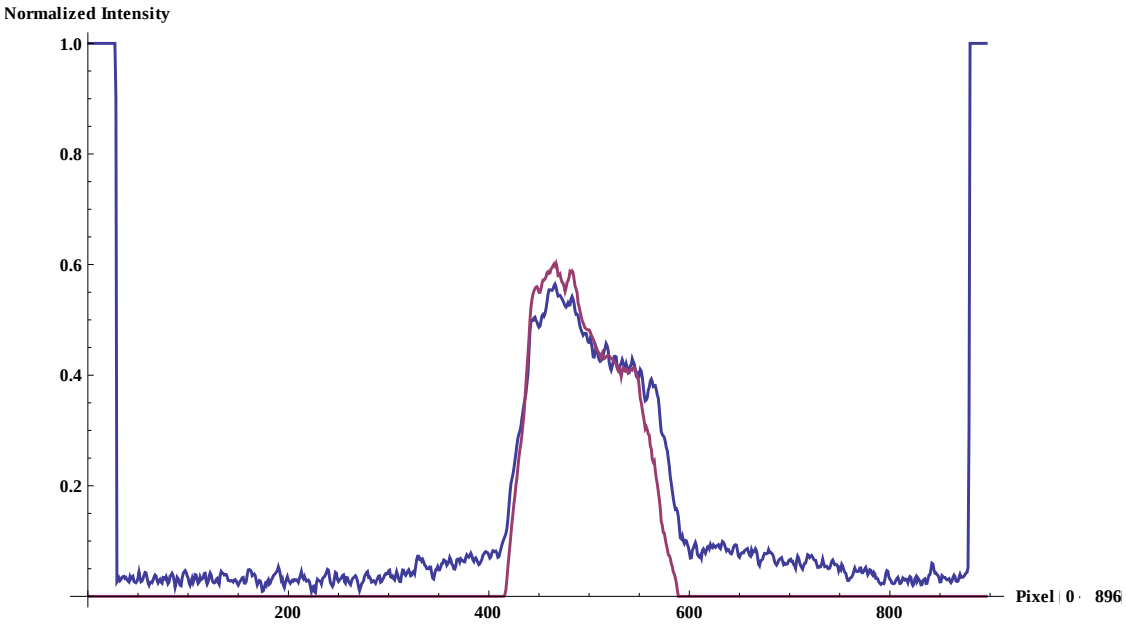
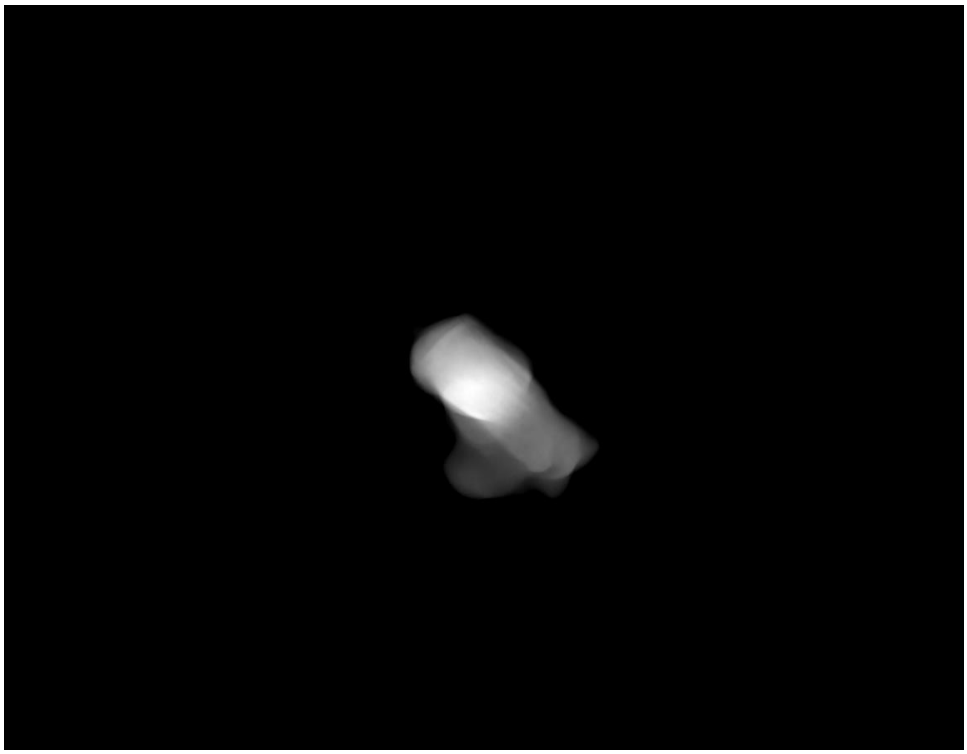
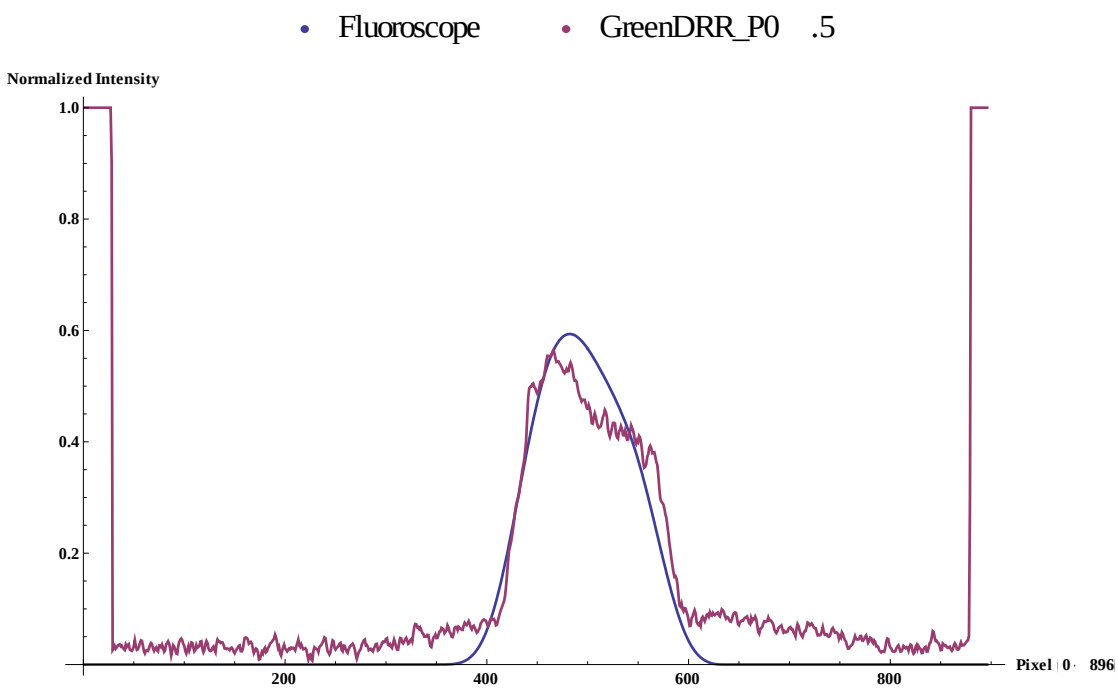


Image without Gaussian Blur



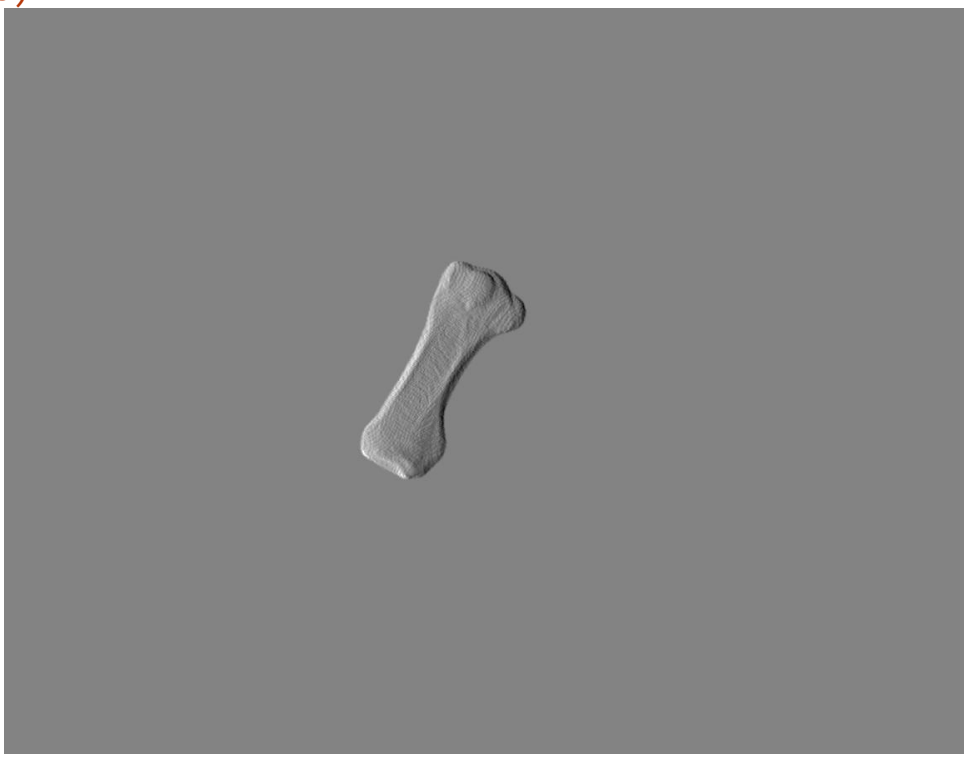
With Gaussian Blur (kernel radius = 40 pixels)


Fluoroscope
GreenDRR_P0 .5
Normalized Intensity
1.0
0.8
0.6
0.4
0.2
200
400
600
800
Pixel | 0 · 896

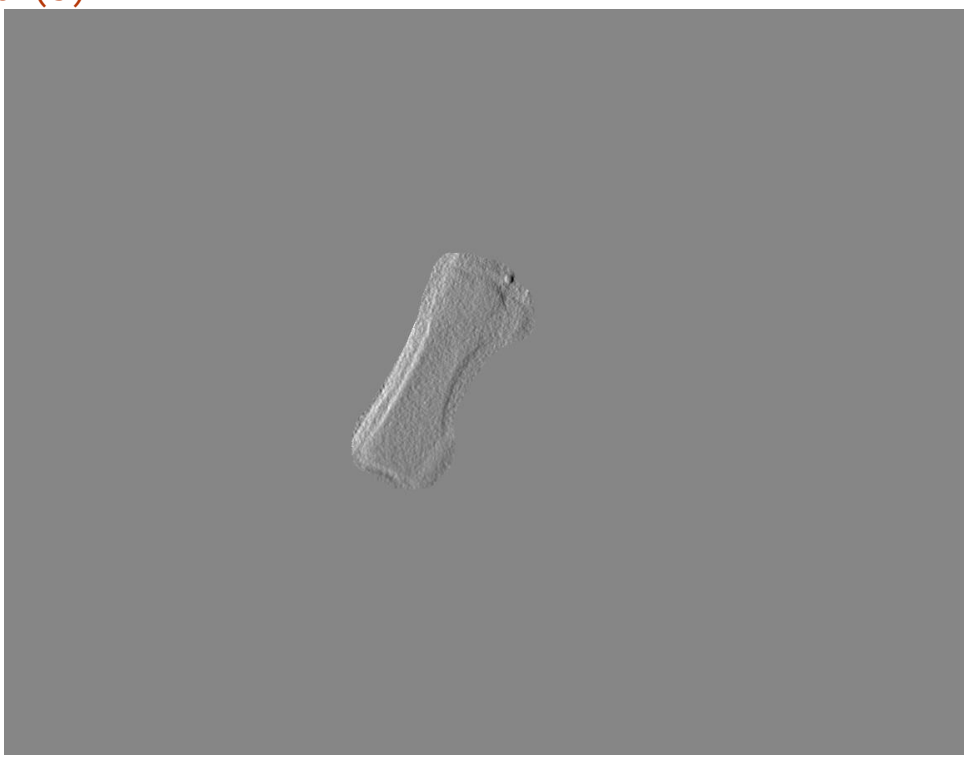
Appendix K: Single- Versus Dual-Stage Blurring

Single Stage blur

i-drr (b)

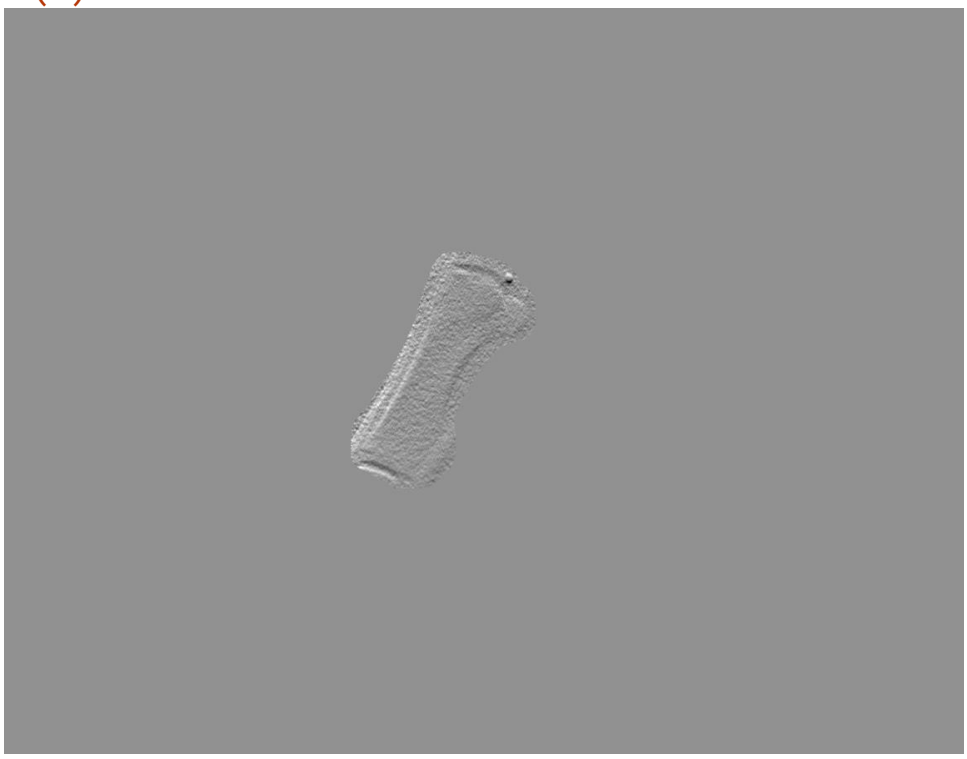


i-fluoro (b)



j-drr (b)

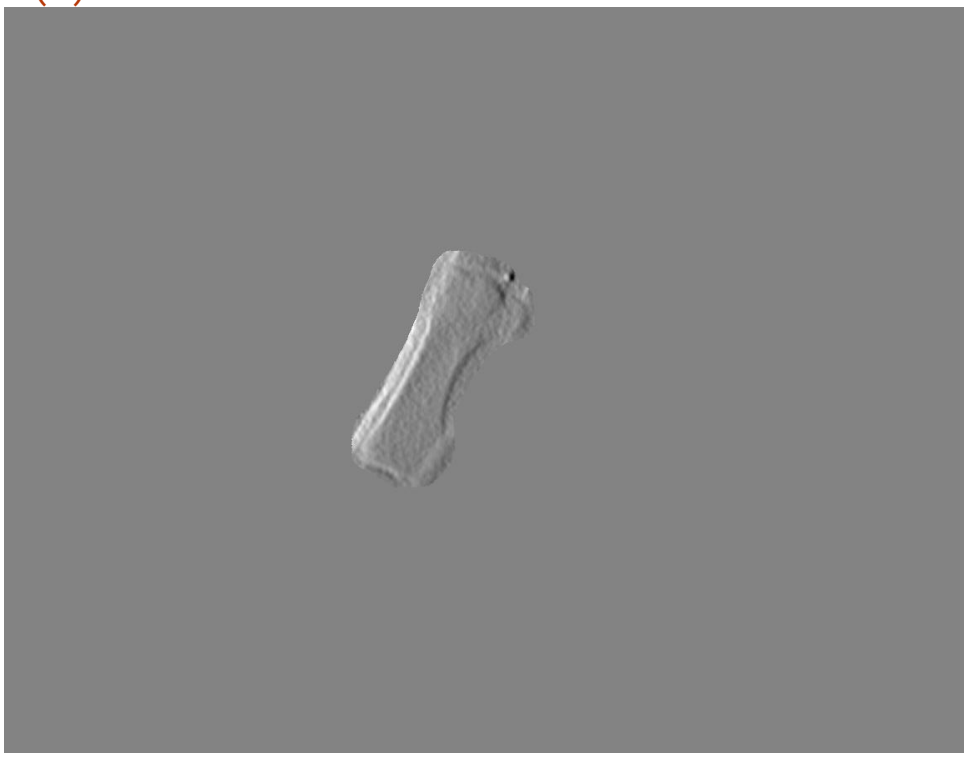
j-fluoro (b)



Dual-Stage Blur

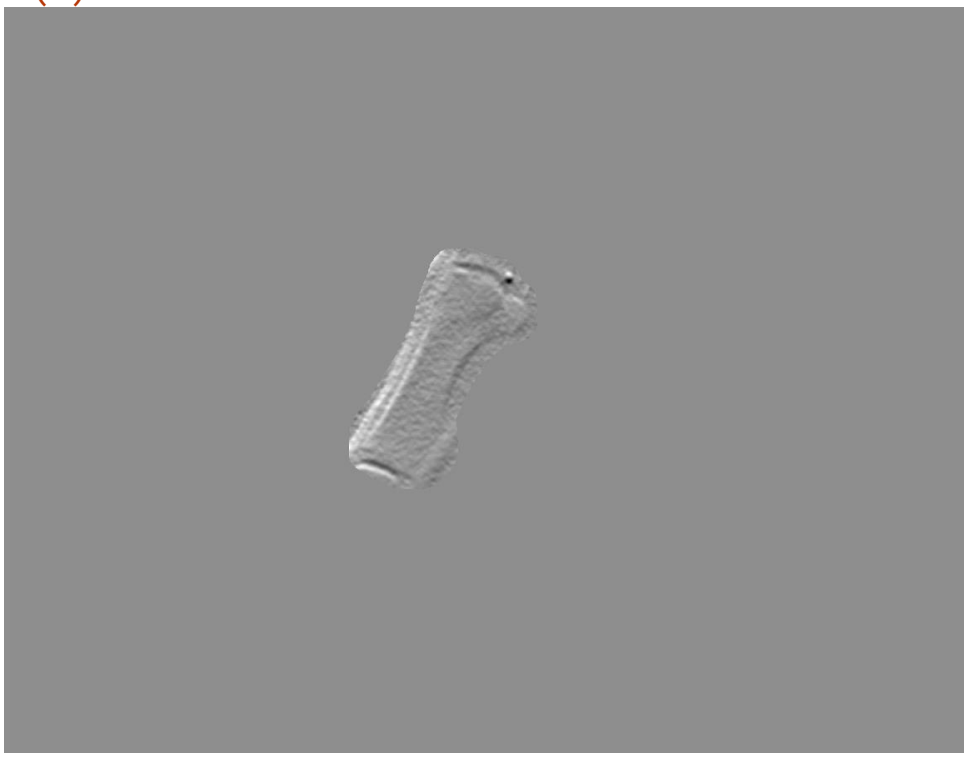
i-drr (b)

i-fluoro (b)



j-drr (b)

j-fluoro (b)



Appendix L: Mathematica Code for 3D Distance Field of Polygonized Data

```
$Line = Infinity;
<<CompiledFunctionTools`
SetDirectory["C:\\Users\\Grant\\Desktop\\bone data"];
polyList=Import["c15.ply","PolygonData"];
vertList=Import["c15.ply","VertexData"];
Dimensions[polyList]
{16068,3}
Compiled Version of Normalization Function
Input: v = coordinate
Output: real
cUnitize=Compile[{{v,_Real,1}}, v/Sqrt[v.v]];
Find normal vector
cTriNorm=Compile[{{coords,_Real,2}},
   Cross[coords[[2]]-coords[[1]],coords[[3]]-coords[[1]]]
   ];
Compiled Version of Point to Vertex Distance w/ absolute value to
remove irrelevant sign (see vertSign to produce appropriate sign)
Input: v = coordinates of point in space; pt = coordinates of a vertex
Output: Absolute distance (real)
cDistToVertex=Compile[{{v,_Real,1},{pt,_Real,1}},Module[{r},
    r=v-pt;
    Abs[Sqrt[r.r]]]];
Compiled Version of Point to Edge Distance w/ absolute value to remove
potentially incorrect sign (see edgeSign to produce appropriate sign)
Input: v = coordinates of point in space; pt = coordinates of a
triangle (a triple of triples)
Output: Absolute distance (real)
cDistToEdge=Compile[{{v,_Real,1},{pt,_Real,2}},Module[{u,r, proj},
    u= cUnitize[pt[[2]]-pt[[1]]];
    r=v-pt[[1]];
    proj = r-(r.u) u;
    Abs[Sqrt[proj.proj]]
    ]];
Compiled Version of Point to Plane of Triangle Distance
Input: pt = coordinates of a triangle (a triple of triples); v =
coordinates of point in space
Output: Absolute distance (real)
cDPerp=Compile[{{pt,_Real,2},{v,_Real,1}},
   Abs[cUnitize[cTriNorm[pt] ].(v-pt[[1]])]];
Edge sign applies the appropriate sign to the distance if the closest
entity is an edge
Input: entity = a list of two vertex numbers that define an edge;
polyList = a list of triples of integers (vertex numbers); vertList =
a list of triples of reals (vertex coordinates)
Output: +/- 1 depending on which “side” of the edge the point in space
is on as determined by the dot product of one edge vector on one
```

```
triangle with another edge vector on the adjacent triangle
cEdgeSign=Compile[{{entity,
_Integer,1},{polyList,_Integer,2},{vertList,_Real,2}},


Module[{adjacentPoly,adjacentCoords,extraPoly1,extraPoly2,extraVert1,extraVer
t2,edgeVerts,n1,v2},

    adjacentPoly=Select[polyList,Length[Intersection[entity,#]]==2&];
    (*If[Length[adjacentPoly]!= 2,"ERROR: Not a manifold Solid"];*)
    adjacentCoords=vertList[[#]]&/@adjacentPoly[[1]];
    extraPoly1=Complement[adjacentPoly[[1]],adjacentPoly[[2]]];
    extraPoly2=Complement[adjacentPoly[[2]],adjacentPoly[[1]]];
    extraVert1=First[vertList[[extraPoly1]]];
    extraVert2=First[vertList[[extraPoly2]]];
    (*Print["extraPoly1 =",extraPoly1];
    Print["extraPoly2 =",extraPoly2];*)
    edgeVerts=vertList[[entity]];

    n1=Cross[adjacentCoords[[2]]-adjacentCoords[[1]],adjacentCoords[[3]]-
adjacentCoords[[1]]];
    (*Print["n1 =",n1];*)

    v2=extraVert2-edgeVerts[[1]];
    (*Print["v2 =",v2];
    Print["n1.v2 =",n1.v2];*)
    If[-n1.v2 < 0,-1,1]
    ]
   ];
Vert sign
Input: entity = a list of length 1 (vertex number); polyList = a list
of triples in integer form (vertex numbers); vertList = a list of
triples in real form (vertex coordinates); ptInSpace = a triple of
reals (coordinates of the point in space)
Output: +/- 1 depending on if the point is on the outside (or inside)
of a convex (or concave) feature as determined by the direction vector
from the point in space to the vertex (read-in) dotted with an edge
vector associated with one of the triangles that contains the vertex
cVertSign[{4},polyList,vertList,{0,1.25,-1}]

cVertSign=Compile[{{entity,
_Integer,1},{polyList,_Integer,2},{vertList,_Real,2},{ptInSpace,_Real,1}},


Module[{adjacentPolys,adjacentCoords,incidentE,extraVert1,ptToVert,edgeVector
,edges,edgeCoords,pAlphas,alphas,angles,normals,normSubAlpha},

    adjacentPolys=Select[polyList,Length[Intersection[entity,#]]==1&];
    (*Print[ "adjacent Polygons = ",adjacentPolys];*)
```

```
    adjacentCoords=vertList[[#]]&/@adjacentPolys;
    (*Print[adjacentCoords];*)

incidentE=Table[Complement[adjacentPolys[[i]],entity],{i,1,Length[adjacentPol
ys]}];
    (*Print["incident edges = ",incidentE,entity];*)

edges=Union[Table[Complement[adjacentPolys[[i]],entity],{i,1,Length[adjacentP
olys]}]//Flatten];

normals=Table[cUnitize[cTriNorm[adjacentCoords[[i]]]],{i,1,Length[adjacentCoo
rds]}];
    (*Print["norms = ",normals];*)
    edgeCoords=vertList[[#]]&/@incidentE;
    (*Print["edgeCoords  = ",edgeCoords];
    Print[edges,entity];*)
    ptToVert=ptInSpace-First[vertList[[entity]]];
    (*Print["direction vector = ",ptToVert];*)
    edgeVector=Table[edgeCoords[[i,j]]-
First[vertList[[entity]]],{i,1,Length[edgeCoords]},{j,1,2}];
    (*Print["edgeVector = ",edgeVector];*)

alphas=Table[ArcCos[(edgeVector[[i,1]].edgeVector[[i,2]])/(Norm[edgeVector[[i
,1]]]*Norm[edgeVector[[i,2]]])],{i,1,Length[edgeVector]}];
    (*Print["alphas = ",alphas];
    Print["Weighted Normal = ",
alphas.normals/Norm[alphas.normals]];ptToVert.(pAlphas*normals/Abs[pAlphas*no
rmals])*)
    (*
    Print["sum = ",angles.normals];
    Print["absSum = ",Abs[angles.normals]];
    normSubAlpha={10000.,10000.,10000.};
    *)
    If[ptToVert.(alphas.normals/Norm[alphas.normals])>0,1,-1]
    ]
   ];
vertList[[{1,2,3,4,6}]]
{{1.21597×10^-6,5.32263×10^-6,-0.499986},{-0.0000345958,0.499965,-6.45812×10^-
6},{0.00011218,-0.499889,5.28063×10^-7},{0.499947,0.0000249911,-
0.0000378997},{-0.499975,-0.0000166341,-0.000016093}}
cDist2Triangle[polyList,vertList,{0,1.25,-1}]
Vert Case _4
incident edges = _{{1,5},{1,3},{2,3},{2,5}}_{4}
norms = _{{0.,0.,-0.5},{0.,0.,-0.5},{1.,1.,1.},{-1.,1.,1.}}
edgeCoords  = _{{{0.,0.5,0.},{-
1.,0.,0.}},{{0.,0.5,0.},{1.,0.,0.}},{{0.,0.,1.},{1.,0.,0.}},{{0.,0.,1.},{-
1.,0.,0.}}}
direction vector = _{0.,0.25,-1.}
alphas = _{0.785398,0.785398,1.0472,1.0472}
Weighted Normal = _{0.,0.847998,0.529999}
```

```
-1.03078
Determines which side of the triangle edge the point is on using the
floating point coordinates of the triangle, the normal to the
triangle, and the point in space:
Input: coords = a list of triples of reals (coordinates of the
polygon); nFace = a triple of reals (the triangle normal vector);
ptInSpace = a triple of reals (coordinates of the point in space)
Output: +/- 1 depending on which “side” of the edge the point is on
cTestEdge=Compile[{{coords,_Real,2},{nFace,_Real,1},{ptInSpace,_Real,1}},

   Module[{edge,nEdge},

    edge = coords[[2]]-coords[[1]];

    nEdge=Cross[edge,nFace];

    nEdge.(ptInSpace-coords[[1]])
    ]
   ];
Determines which side of the voronoi boundary the point is on based on
2 floating point coordinates, the normal to the triangle, and the
point in space:
Input: coords = a list of triples of reals (coordinates of the
polygon); nFace = a triple of reals (the triangle normal vector);
ptInSpace = a triple of reals (coordinates of the point in space)
Output: +/- 1 depending on which “side” of the voronoi edge the point
is on
cVoronoiEdge=Compile[{{coords,_Real,2},{nFace,_Real,1},{ptInSpace,_Real,1}},

   Module[{edge,nEdge,vEdge,side},

    edge=coords[[2]]-coords[[1]];

    nEdge=Cross[nFace,edge];

    vEdge=Cross[nFace,nEdge];

    side=vEdge.(ptInSpace-coords[[1]])
    ]
   ];
Finds the closest entity and its absolute distance:
Input: triangle = a triple of integers (vertex numbers that define a
polygon); coords = a list of triples of reals (coordinates of the
polygon); nFace = a triple of reals (the triangle normal vector);
ptInSpace = a triple of reals (coordinates of the point in space)
Output: a list with the first entry being the absolute distance from
the point to the (nearest) entity, the second entry being a vertex
number, the third entry being a 0 if closest to a vertex and the
```

vertex number if closest to an edge, the fourth entry being -1 if closest to a vertex, -2 if closest to an edge, and 0 if closest to the plane of the triangle. If the nearest entity is the plane of the triangle, an arbitrary list of {1.5, 0, 0, 0} is returned.

```
cCheckEdge=Compile[{{triangle,_Integer,1},{coords,_Real,2},{nFace,_Real,1},{p
tInSpace,_Real,1}},

   If[cTestEdge[Take[coords,2],nFace,ptInSpace]≠ 0,

    If[cVoronoiEdge[{coords[[1]],coords[[2]]},nFace,ptInSpace]≠ 0,
     {cDistToVertex[coords[[1]],ptInSpace],triangle[[1]],0,-1},

     If[cVoronoiEdge[{coords[[2]],coords[[1]]},nFace,ptInSpace]≠ 0,
      {cDistToVertex[coords[[2]],ptInSpace],triangle[[2]],0,-1},

      {cDistToEdge[ptInSpace,Take[coords,2]],triangle[[1]],triangle[[2]],-2}
      ]
     ],
    {1.5,0,0,0}]
   ];
```

Returns the absolute distance to an entity (and the entity itself) based on checking for an empty set. The variable *triangle* is a triple of integers that represent a triangle:

Input: triangle = a triple of integers (vertex numbers that define a polygon); ptInSpace = a triple of reals (coordinates of the point in space); vertList = a list of triples in real form (vertex coordinates);

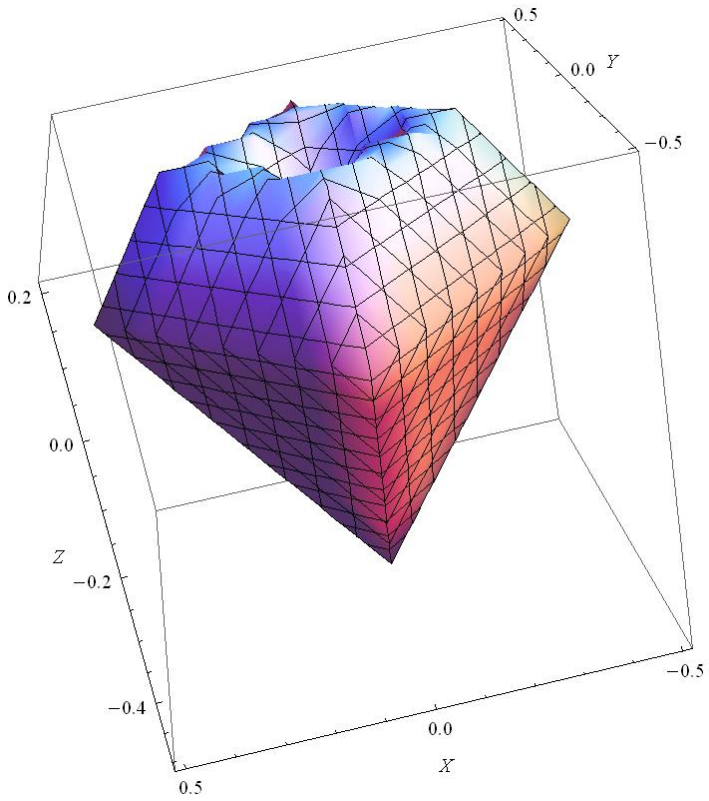
Output: a list with the first entry being the absolute distance from the point to the (nearest) entity, the second entry being a vertex number, the third entry being a 0 if closest to a vertex and the vertex number if closest to an edge, the fourth entry being -1 if closest to a vertex, -2 if closest to an edge, and a positive integer if closest to the plane of the triangle. If the nearest entity is the plane of the triangle, the second entry in the list is the sign of the distance (i.e., this determines which “side” of the plane of the triangle the point in space is on).

```
cDistAndEntity=Compile[{{triangle,_Integer,1},{ptInSpace,_Real,1},{vertList,_
Real,2}},

   Module[{coords,nFace,triTemp,list,dists,vertNos},

    triTemp=triangle;
    coords=vertList[[#]]&/@triangle;
    nFace=cTriNorm[coords];

    If[Last[cCheckEdge[triTemp,coords,nFace,ptInSpace]]≠ 0,
     cCheckEdge[triTemp,coords,nFace,ptInSpace],
     triTemp=RotateLeft[triTemp];
```

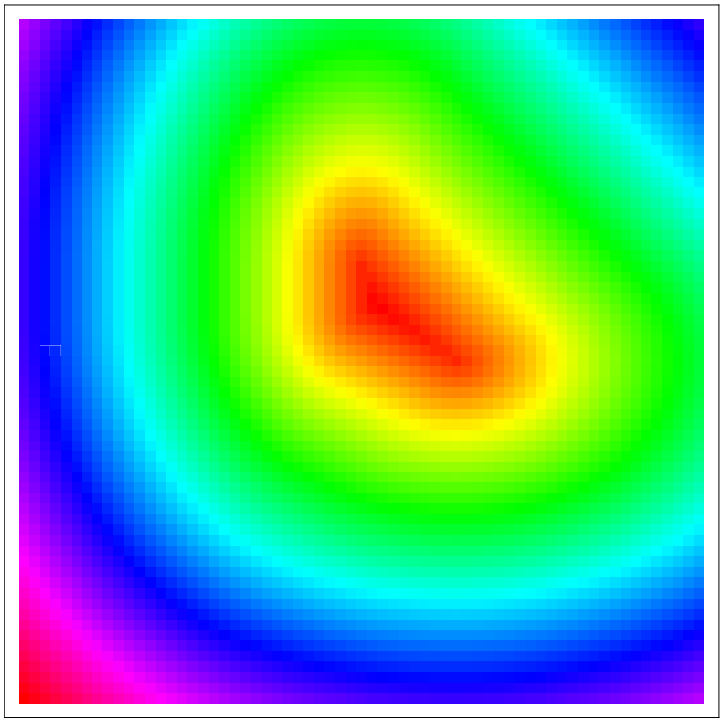
```
     coords=RotateLeft[coords];

     If[Last[cCheckEdge[triTemp,coords,nFace,ptInSpace]]≠0,
      cCheckEdge[triTemp,coords,nFace,ptInSpace],
      triTemp=RotateLeft[triTemp];
      coords=RotateLeft[coords];

      If[Last[cCheckEdge[triTemp,coords,nFace,ptInSpace]]≠0,
       cCheckEdge[triTemp,coords,nFace,ptInSpace],

       {cDPerp[coords,ptInSpace],(If[nFace.(ptInSpace-coords[[1]])<0,-
1,1])*triangle[[1]],triangle[[2]],triangle[[3]]}
        (*{cDPerp[coords,ptInSpace],triangle,nFace,coords[[1]]}*)
       ]
      ]
     ]
    ]
   ];
Distance Function: Calls all previous functions to generate the
distance from a point to a triangle in Euclidean 3-space
Input: polyList = a list of triples of integers (vertex numbers);
vertList = a list of triples of reals (vertex coordinates);  ptInSpace
= a triple of reals (coordinates of the point in space);
Output: a signed distance (real)
cDist2Triangle=Compile[{{polyList,_Integer,2},{vertList,_Real,2},{ptInSpace,_
Real,1}},


Module[{indices,dist,entity,sorted,closest,unsorted,vertSignDist,edgeSignDist
,planeSignDist,close2,close3,close4},
    unsorted=cDistAndEntity[#,ptInSpace,vertList]&/@polyList;
    (*Print[unsorted];*)
    sorted=Sort[unsorted,(Abs[#1[[1]] ]<Abs[#2[[1]] ]&)];
    (*Print[sorted];*)
    (*Print[Dimensions[sorted]];*)
    closest=sorted[[1]];
    (*Print[closest];*)
    dist=closest[[1]];
    close2=Round[closest[[2]] ];
    close3=Round[closest[[3]] ];
    close4=Round[closest[[4]] ];
    (*Print[cVertSign[{close2},polyList,vertList,ptInSpace]];*)
    If[close4 = -1,
     (*Print["close4 = -1"];Print["Vert Case ",close2
];*)cVertSign[{close2},polyList,vertList,ptInSpace]*dist,

     If[close4 = -2,
      (*Print["close4 = -
2"];*)cEdgeSign[{close2,close3},polyList,vertList]*dist,
```

```
      If[close4>0,
       (*If[closest[[2]]≠0,Print["ERROR: Non-positive vertex number"]];*)
       (*Print["close4 > 0"];*)Sign[close2 ]*dist,
       (*Print["Other"];*)Sign[close2 ]*10000
       ]
      ]
     ]
    ]
   (* ,RuntimeOptions→"Speed"*)];
numList=RandomReal[{-15,15},{100000,3}];
Timing[cDist2Triangle[polyList,vertList,#]&/@numList][[1]]
40.623
N[40^3]
64000.
1.5/50
0.03
cDist2Triangle[polyList,vertList,{-0.6,0.6,-1.1}]
1.03924
-0.25,0.25,-0.25
Test Point for “diamond.ply”
cDist2Triangle[polyList,vertList,{0.1,0.2,-0.7}]
-0.300006
Test point for “vshape.ply” {0,1.25,-1}
cDist2Triangle[polyList,vertList,{0,1.5,-1.2}]
Vert Case _4
-1.3
vertList[[4]]
{0.,1.,0.}
N[2/64]
0.03125
step=0.03125;
r=1;
Clear[testData]
testData=Table[cDist2Triangle[polyList,vertList,{x,y,z}],{x,-r,r,step}, {y,-
r,r,step},{z,-r,r,step}];
Dimensions[testData]
{21,21,21}
testData2=Table[cDist2Triangle[polyList,vertList,{x,y,z}],{x,-r,r,step}, {y,-
r,r,step},{z,-r,r,step}];
Export["DiamondField128.xls",testData]
VShapeField64.xls
```

```
ListContourPlot3D[testData, Contours→ {0.000001}, BoxRatios → {1,1,1},
DataRange → {{-r,r},{-r,r},{-r,r}}, AxesLabel → {Z,Y,X}]
```




```
ArrayPlot[Reverse[testData2[[10]]],ColorFunction→Hue,PlotRange→Automatic]
```



Education

Eastern Washington University — **B.S. Mechanical Engineering Technology**

Obtained August 2008 — (Minor: Physics)

University of Washington — **M.S. Mechanical Engineering**

Obtained August 2010 — (Glass/Ceramic Material Development)

University of Washington — **Ph.D. Mechanical Engineering**

Obtained September 2014 — (Medical Image Registration using the GPU)

Publications

1. Marchelli, G., Ganter, M., Storti, D. (2009). "New Material Systems for 3D Ceramic Printing." *Proceedings of the 2009 International Solid Freeform Fabrication Symposium*, Austin, TX.
2. Marchelli, G., Storti, D., Prabhakar, R., Ganter, M. (2010). "An Introduction to 3D Glass Printing." *Proceedings of the 2010 International Solid Freeform Fabrication Symposium*, Austin, TX. **Outstanding Paper Award**.
3. Marchelli, G., Storti, D., Prabhakar, R., Ganter, M. (2011). "The Guide to 3D Glass Printing: Developments, Methods, Diagnostics and Results." Rapid Prototyping Journal. 17(3).
4. Marchelli, G., Ledoux, W., Isvilanonda, V., Ganter, M., Storti, D. (2011). "An Automated Method for Creation of Patient-Specific Volumetric Articular Cartilage Elements in the Human Foot." Proceedings of the 2011 Design of Medical Devices Conference, Minneapolis, MN.
5. Marchelli, G., Ganter, M., Storti, D. (2011). "Exploring the Applicability of Common Young's Modulus-Porosity Models to Highly Porous Three-Dimensionally Printed (3DP) Stoneware Ceramic." Journal of Ceramic Science and Technology.
6. Marchelli, G., Haynor, D., Ledoux, W., Tsai, R., and Storti, D. (2013). "A flexible toolkit for rapid GPU-based generation of DRRs for 2D-3D registration." SPIE paper #8669-47.
7. Storti, D., and Marchelli, G. (2013). "Real-time Interaction with 3D Medical Imaging using 3D Textures." GTC 2013 P01790. Located at: http://www.gputechconf.com/gtcnew/on-demand-gtc.
8. Marchelli, G., Haynor, D., Ledoux, W., Ganter, M., and Storti, D. (2013). "Graphical User Interface for Human Intervention in 2D-3D Registration of Medical Images." Proceedings of the 9th International Conference on Multibody Systems, Nonlinear Dynamics and Control, Paper #DETC2013-13659
9. Marchelli, G., Ledoux, W., Isvilanonda, V., Ganter, M., Storti, D. (2014). "Joint-specific Distance Thresholds for Patient-specific approximations to Articular Cartilage Modeling in the First Ray of the Foot." Medical & Biological Engineering & Computing.
10. Kindig, M., Marchelli, G., Iaquinto, J., Storti, D., Haynor, D., Sangeorzan, B., Ledoux, W. (2014). "Model-Based Validation of Graphics Processing Unit Algorithm to Track Foot Bone Kinematics Using Fluoroscopy." 2014 Orthopaedic Research Society Annual Meeting, New Orleans, LA.

Patents

1. Marchelli, G., Prabhakar, R. (2011). International Patent Application PCT/US2011/038954: "Porous Glass Articles Formed Using Cold Work Process."

2. Marchelli, G., Storti, D., Ganter, M., Ledoux, W., Haynor, D., Cavanagh, P. (2011). U.S. Provisional Patent Application 61/560,666. "A Distance-Based Toolbox for Automated Computational Biomechanical Modeling." November 16, 2011.

Accolades

- PI on NSF SBIR Phases I & II Research Grants (2012-2015)
- Board Member for SPU Recycling Market Development Advisory Committee (2013)
- Entrepreneurial Board Member for "The Center for Engineering Innovation and Teaching" (2012)
- Invited Panelist at the Northwest Recycling/Processors Roundtable Event (2012)
- PI on UW C4C CGF Research Grant (2011-2012)
- Publication in Special Edition "Best Papers" of the Rapid Prototyping Journal (2011)
- Xerox Technical Minority Fellow (2009 & 2011)
- Regional Semifinalist in National Cleantech Open Competition (2010/2011)
- Grand Prize Winner of the UW Environmental Innovation Challenge (2010)
- Outstanding Paper Award at the International Solid Freeform Fabrication Symposium (2010)
- Semifinalist and Best Clean Tech Award in the UW Business Plan Competition (2010)
- Mathematica Student Certification (2009)
- Quality Service Initiative Award for excellence as a Mathematics Tutor at EWU (2008)
- Engineer-in-Training License #28605 (2007)

Experience

Doctoral Research Assistant University of Washington: Seattle, January 2010 - Present

- Development of GPU computing toolkit for rapid medical image registration
- Patient-specific model generation to minimize the pain and suffering in the feet of diabetic patients
- Developing an algorithm that automatically generates soft tissues (cartilage, joint capsules, etc.) in the foot
- Funding in part by the Dept. of Veterans Affairs Rehabilitation R&D Service grant A6973R.

Co-Founder/Chief Technology Officer EnVitrum, Inc.: Seattle, December 2009 - Present

- Co-inventor of novel manufacturing process for conversion of waste glass into green building materials
- Successfully co-built a technology driven company from the ground up
- Principal Investigator of National Science Foundation SBIR Phases I & II Research Grants
- Principal Investigator of University sponsored commercialization research grant
- Managing numerous MBA and undergraduate research interns
- Drafting business plans, performing market research and financial analyses
- Writing Small Business Innovation Research (SBIR) grant proposals

M.S. Research Assistant University of Washington: Seattle, March 2009 - August 2010

- Project lead in developing/characterizing new materials for 3D printing including design of experiment
- Research lead in forming the world's first 3D printed glass structure
- Conducting mechanical testing on new materials in accordance with ASTM International
- Development and optimization of complex heating schedules for 3DP ceramic and glass materials

Teaching Assistant (ME 599) University of Washington**:** Seattle, January 2012 - June 2013 (2 quarters)

- Periodic class lectures
- Creation of assignment templates
- Drafting of instruction guides

Teaching Assistant (ME480) University of Washington**:** Seattle, April 2009 - December 2013 (4 quarters)

- Periodic class lectures and weekly lab lectures
- Instruction built around cutting-edge Additive Manufacturing (AM) processes and research
- Operation, maintenance, & service of multiple AM equipment and furnaces
- Laboratory demonstrations and lectures on mold making, casting, and additive manufacturing
- Providing foundation for materials science research in ceramic, glass, and polymeric materials
- Maintenance of course website

Mathematics Tutor Eastern Washington University: Cheney, January 2007-June 2008

- Teaching mathematics ranging from basic algebra to advanced calculus and differential equations
- Leading both group and individual study sessions
- Freelance mathematics tutor for the EWU Army ROTC—instrumental in the design of the mathematics outreach program

Affiliations

American Society for Mechanical Engineers	American Society for Testing and Materials	Sigma Alpha Lambda National Honors & Leadership Society	Founding Father at EWU Delta Chi Fraternity	UW Hun Gar Kung Fu Club Officer