Textured Mesh Reconstruction of Indoor Environments Using RGB-D Camera

Collin Boots

A THESIS

in

Robotics

Presented to the Faculties of University of Pennsylvania in Partial

Fulfillment of the Requirements for the Degree of Master of Science in Engineering

2014

Dr. Daniel D. Lee
Supervisor of Thesis

Dr. Camillo J. Taylor
Graduate Group Chairperson

Acknowledgment

I would like to thank my advisor Dr. Daniel Lee for keeping my eyes to the sky and my feet on the ground as I pursued this project. Special thanks to Dr. Camillo J. Taylor and Dr. Kostas Daniilidis for serving on my thesis committee.

This thesis is powered by CUDA, so I need to thank Patrick Cozzi for teaching it to me. I would never have thought of most of the optimizations or algorithms essential to the success of this project without taking his CIS 565 GPU Programming and Architecture course. On that note, I must also thank Dalton Banks for partnering with me on the first draft of the pipeline code in CIS 565 last semester.

Finally, I doubt I would have made it through the semester with both my sanity intact and thesis complete without the constant support of my friends at the University of Pennsylvania. Thanks to you all.

Abstract

As robots continue to be incorporated into human environments, the need for high-speed systems that can recognize geometry increases. This thesis is intended to work towards such a system based on RGB-D cameras, triangle meshes, and the powerful parallel computing capability modern Graphics Processing Units (GPUs) offer. A robust, high-speed GPU based pipeline that converts raw RGB and depth frames into a 3D textured mesh representation of large planar surfaces in the field of view is presented. Triangle meshes are efficient to store, simple to manipulate and refine, and very versatile. Meshes have the added benefit of being well suited to GPU hardware (which was originally designed for just that purpose).

The created pipeline is capable of segmenting large planar surfaces and converting them to an efficient QuadTree based textured mesh format in real-time. The additional memory burden for storing the color data in texture format in most cases creates a net increase in memory storage for a single frame. However, the pipeline is intended to be part of a larger system that integrates data from multiple frames so this up-front memory cost would be mitigated over time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

As robots continue to be incorporated into human environments, the need for intelligent and high-speed reasoning about the objects around them increases dramatically. At the simplest level, mobile robots need to create a map of their environment for navigation. At a higher level, some robots need to recognize distinct objects in their environment, track object movement, and have some intuitive sense of object geometry that is easily stored and processed. Even more importantly, robots must be able to efficiently generate adaptable models of their environment, or worldview, from sensor data in real time. Many different methods for representing the world have been proposed and implemented, and they will be discussed below in more detail. Like the human brain, the robot should also be able to perform these low level functions with only minimal intervention from higher cognitive functions. Such technology also has potential uses beyond robotics. Applications may include easily modeling indoor

environments for interior design concepts, generating 3D tours or maps for various buildings, or low resolution rough mapping of archaeological excavations.

This thesis is intended to work towards such a system based on RGB-D cameras, triangle meshes, and the powerful parallel computing capability modern Graphics Processing Units (GPUs) offer. RGB-D cameras like Microsoft's Kinect provide a great low cost solution for capturing 3D environments. Triangle meshes are efficient to store, simple to manipulate and refine, and very versatile. Meshes have the added benefit of being well suited to GPU hardware (which was originally designed for just that purpose). This thesis lays out a robust, high-speed GPU based pipeline that converts raw RGB and depth frames into a 3D textured mesh representation of large planar surfaces in the field of view.

1.2 Related Work

A great variety of methods have been applied to RGB-D camera data in an effort to construct a coherent worldview. Each has advantages and disadvantages.

1.2.1 Worldview Storage Models

Point Cloud Models The simplest approach to storing RGB-D data is as a raw point cloud. Each point is stored in a self contained data structure containing at least the point's position and color information. E.g.

$$P_i = \{pos_x, pos_y, pos_z, red, green, blue\}$$

This approach allows a complete record of the raw data to be stored very easily, but the size of the data stored will grow very quickly, and simply storing the data linearly results in very slow queries.

Nearest neighbor approximations have recently become a popular approach for speeding queries on large point clouds[21]. However, achieving reliably fast queries usually requires some form of hierarchical tree structure like K-d trees[24] or octrees[44]. K-d trees are much more adaptable and usually more efficient in terms of memory storage, but octrees have a significant advantage when it comes to incrementally building a point cloud because points can very easily be inserted into the appropriate octree leaf node with no duplication or restructuring. K-d trees are better suited for compressing point clouds offline.

Voxel Space Models Perhaps the most robust and impressive real-time surface reconstruction algorithm to date is Microsoft's Kinect Fusion[22, 14]. An open source implementation called KinFu is also available[29]. Kinect Fusion uses a bounded 3D voxel space where each voxel stores the distance to the nearest detected surface or empty space (the default). Figure 1.1 shows the workflow of the Kinect Fusion system. The point cloud generated by each RGB-D frame is projected into the voxel space and each point updates nearby voxels' distance to nearest surface metric. Implicit surfaces can then be rendered by raycasting through the voxel space and detecting the distance sign crossover point. This results in very high resolution and fidelity reconstructions of the implicit surfaces in the environment. The primary disadvantage of this approach is the workspace size is limited by memory and compute resources which scale with

the resolution and dimensions of the voxel space.

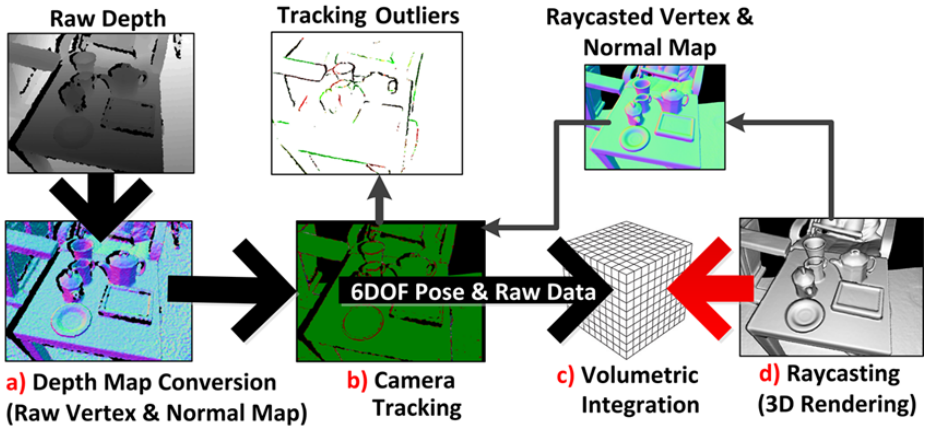

Figure 1.1: Kinect Fusion Tracking and Reconstruction Pipeline. Reproduced from[14]

A spatial extension of this system called Kintinuous was introduced in 2012[43, 42]. Kintinuous uses a mobile voxel space which periodically relocates based on camera motion. Voxels that move outside the space are converted to a triangular mesh using a greedy mesh triangulation procedure described by Marton *et al.*[20].

Others have attempted to use sparse voxel octrees (SVOs) to represent larger unbounded spaces using voxels of varying resolution[17, 32]. This approach is very amenable to human environments like buildings which are dominated by free space. However, all of these approaches are still limited to storing discrete data points and contain no inherent information about coherent objects or regions.

Point Cloud Offline Processing Point clouds can also be processed offline to extract surfaces and objects. These systems are not burdened by the strict requirements of real-time computation so they generally can produce more globally optimal solutions than their online counterparts. These systems generally target unordered

point clouds. Marton *et al.* proposed an incremental approach to triangulation of noisy point clouds[20]. The approach was amenable to introduction of new registered RGB-D frames by only triangulating points that did not correlate with existing mesh models, but the implementation was far too slow for real-time applications. Ma and Others have introduced a system for planar simplification of dense point clouds using a QuadTree based algorithm[19, 18] and texture maps. This thesis will rely heavily on this innovation. Other approaches worked towards implicit surfaces like parameterized smooth surface fits [13] or Poisson surfaces[15, 5].

1.2.2 Real-Time Processing

A crucial part of this thesis is achieving high speed plane segmentation. Many different researchers have created efficient parallel GPU implementations of common segmentation approaches like Markov Random Fields[2], the Potts model[1], parallelized graph-based approaches[41], seeded region growth (SRG)[28], and region growth based on local smoothness constraints[31]. However, this thesis parallelizes an algorithm by Holz *et al.* [11] based on clustering in normal space. This approach is superior for application in this thesis because it can readily detect large planes and solve them globally in a very data parallel manner.

1.2.3 Additional Resources

Although this thesis focuses on detecting and representing only planar elements, the envisioned worldview generating pipeline would need several additional components

and capabilities to be successful. Additionally, many data processing technologies exist that can improve the quality of the input data through more sophisticated filtering.

RGB-D Filtering The Kinect sensor provides its own set of filtering challenges. Khoshelham and Elberink provided a very useful in depth analysis of Kinect accuracy and resolution specifically with indoor mapping applications in mind[16]. Without some amount of preprocessing and filtering, quantization of the depth data and the fact that resolution and accuracy decrease with increasing distance from the sensor would completely prevent this thesis's pipeline from producing any usable results.

This thesis uses bilateral filtering of the depth image[27, 38]. A simple Gaussian filter is used to smooth local point normals, but other methods exist that could improve results if they could be implemented efficiently in parallel. One such method is adaptively computing filter windows using integral images[12]. Kinect data also is notoriously full of holes from washed out areas, shadows and other noise related effects. Some research makes an effort to fill these holes[6, 34, 45], but this thesis is geared towards progressive improvement of the world model and hopes that any holes will be patched by other viewing angles.

3D SLAM This thesis only deals with processing the current frame, but it was designed with an eye towards creating a closed loop system to integrate data from multiple frames. To accomplish this, a Simultaneous Localization and Mapping (SLAM) system will be needed. SLAM has been practically implemented using a

Kinect using a GPU [36, 35]. Whelan *et al* did fantastic work in comparing multiple SLAM methods combining RGB feature tracking and full point cloud registration algorithms[42]. Other research has evaluated pose graph optimization [7] and point-plane alignment[37] techniques for hand-held 3D sensors like the Kinect.

Mesh Processing and Modification One future direction for this work is to incorporate new information into existing meshes through efficient topology changes and resolution modification. The graphics community offers a wide range of insight into these methods, ranging from tracking surfaces through complex topology evolution[4], modeling deformable solids[33], Mesh subdivision and simplification approaches[30], and texture re-mapping or the effects of image warping[10, 9, 40, 26, 8].

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 will review some basic precepts of parallel algorithm design, along with specific optimization considerations for programming GPUs with CUDA. Chapter 3 will provide an overview of the system design as well as break down the pipeline into sub-modules to be explored in much more detail in Chapter 4. Finally, Chapters 5 and 6 will provide performance analysis, conclusions, and areas of potential improvement for future work.

Chapter 2

Parallel Programming Paradigms

This chapter will provide and introduction to the fundamentals parallel programming. The remaining chapters will assume a working knowledge of parallel algorithms, GPU architecture, and CUDA optimization techniques.

2.1 Principles of Parallel Programming

As the physical limits of transistor size are being reached, the trend in computing has been away from making processors faster and towards parallel processing. The introduction of General Purpose Graphics Processing Units have made data parallel algorithms a very attractive approach to accelerating computationally intensive tasks. However, migrating from sequential computing to parallel platforms is not always simple or even wise. The following are some principles to keep in mind when considering using GPU acceleration.

Keep data parallel Parallel computing is best suited to performing simple operations on large numbers of independent data elements, such as n-body simulations and per-pixel image processing operations. Aside from these trivial cases, adapting sequential algorithms to a GPU architecture efficiently can be challenging. Not all loops are easily parallelizable.

Memory is a bottleneck Computing with thousands of cores in a GPU makes memory I/O into a major bottleneck. Usually the data transfer to and from the parallel device will take much longer than the actual compute time. Be extremely conscious or memory usage.

Minimize code divergence On most architectures, parallel code runs in batches for increased efficiency. If the code is full of if/then/else statements, the divergent execution paths can dramatically impact performance.

Avoid sequential operations and synchronization Atomic operations that must be performed in order or synchronizations between concurrent threads of execution can be very slow. The more the threads can be independent of each other, the more efficiently the hardware will be able to execute them in parallel.

Be mindful of the hardware When writing performance critical code, understanding the underlying architecture can be a big help. Understanding how the code will actually be distributed and executed across multiple processors will make optimization much easier.

2.2 Parallel Algorithm Building Blocks

Just as there are fundamental building blocks of sequential computing like search, recursion, and iteration, many parallel algorithms make use of primitive operations like reduce, scan, and stream compact.

2.2.1 Reduction

Reduction is any operation on an array of elements that computes a single result. Common examples include sums, average, min, max, and product. In sequential terms, implementing reduction is very simple. Using sum as an example, algorithm 1 shows how this might be implemented.

Data: n element array X
Result: sum of elements in X
i = 0;
sum = 0;
while $i \ ; \ n$ **do**
    sum += X[i];
    i++;
end

Algorithm 1: Sequential Sum

Notice that in the sequential algorithm, the accumulator's value at each iteration depends on the value in the previous iteration. To perform a parallel sum, the associative property of the operator can be exploited to break group the operation into a structured set of binary operations that can be performed in parallel (Figure 2.1).

This parallel reduction method will work for any associative binary operator.

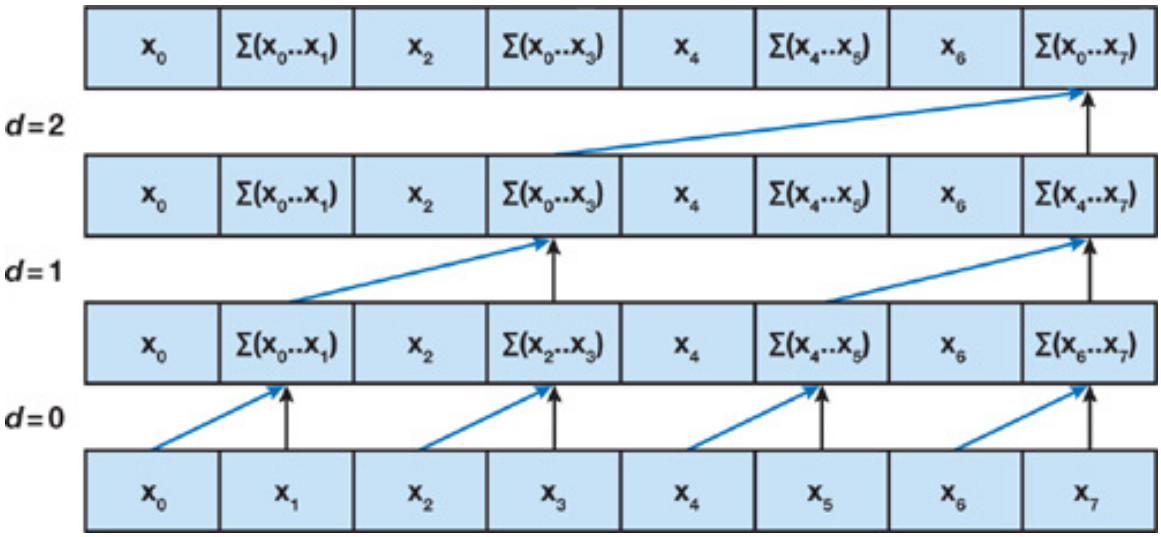

Figure 2.1: Parallel Reduction. Reproduced from[23]

2.2.2 Scan

The scan operation is similar to a performing a cumulative sum operation on an array. There are two forms of scan: exclusive and inclusive. Usually the term scan refers to an exclusive scan.

Given an associative binary operator $\oplus$ with identity value I and an array of n items $[a_0, a_1, \ldots a_{n-1}]$, an exclusive scan returns the array:

$$[I, a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

An inclusive scan returns the shifted result:

$$[a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

The sequential implementation of a sum scan is trivial and outlined in Algorithm 2.

Data: n element array X
Result: array Y contains exclusive scan of X
i = 1;
Y[0] = 0;
while i ¡ n **do**
  Y[i] = Y[i-1] + X[i-1];
  i++;
end

Algorithm 2: Sequential Sum

To perform a parallel scan in a work efficient manner, first a reduction is performed on the array as in Figure 2.1. Then, the last element of the array is set to 0, and a series of swaps and sums are performed as shown in Figure 2.2.

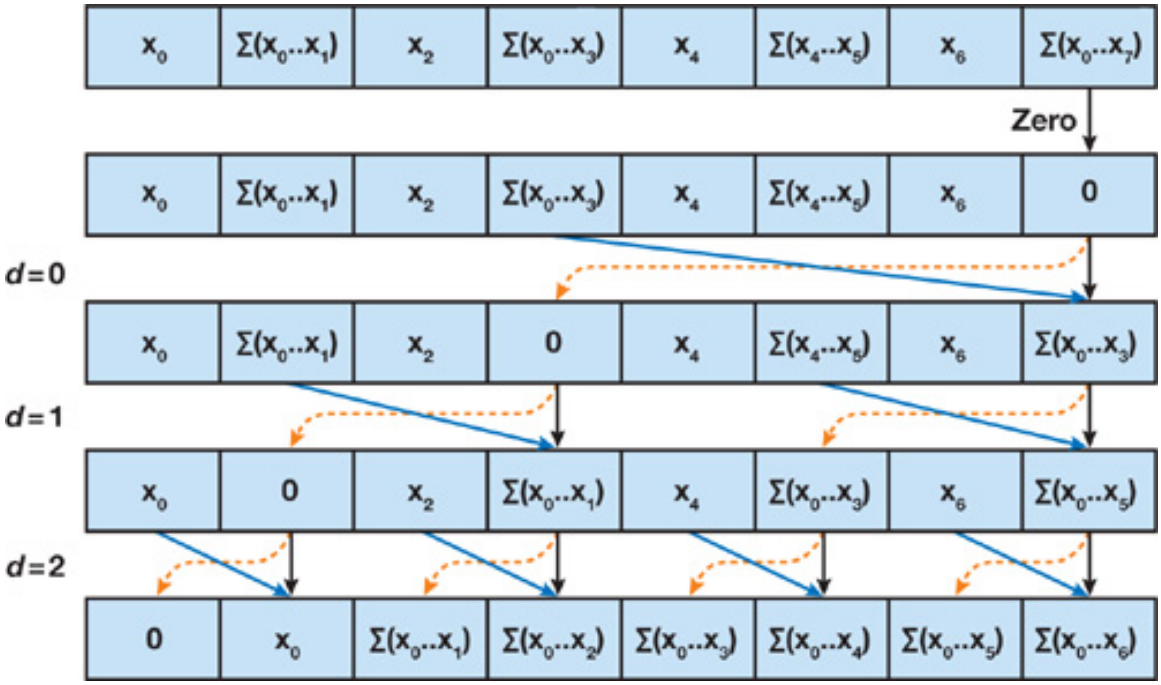

Figure 2.2: Down-sweep phase of parallel sum scan. Reproduced from[23]

2.3 Stream Compaction

In parallel computing, stream compaction is a useful tool for selecting a subset of data and compressing it into a coherent memory block. For example, stream compaction could be used to transfer all of the odd elements in array A of Figure 2.3 and place them in order compressed at the start of array B.

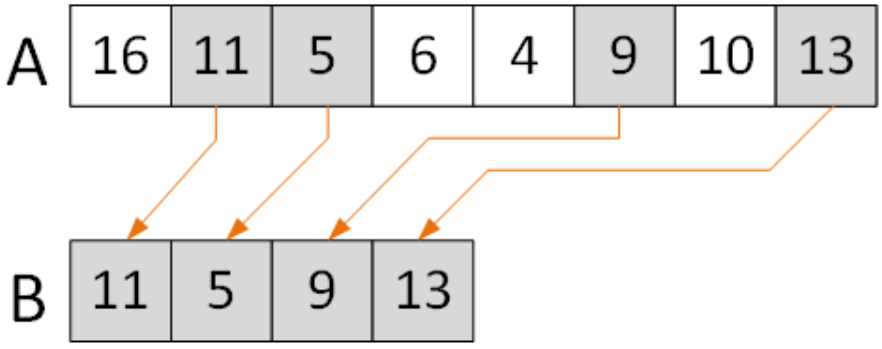

Figure 2.3: Example of stream compacting odd valued integers

Accomplishing this is surprisingly simple. As shown in Figure 2.4, a secondary array of flags is created. This array has a 1 for every odd number in array A and 0 for all others. An exclusive scan is performed on the flag array. Notice that every flagged element now has a unique zero-based index associated with it. The flagged elements are then "scattered" by the computed index. The scatter function is outlined in Algorithm 3.

2.4 Programming with CUDA

CUDA is a C/C++ based programming language created by NVIDIA to expose the general compute functionality of their GPU processors.

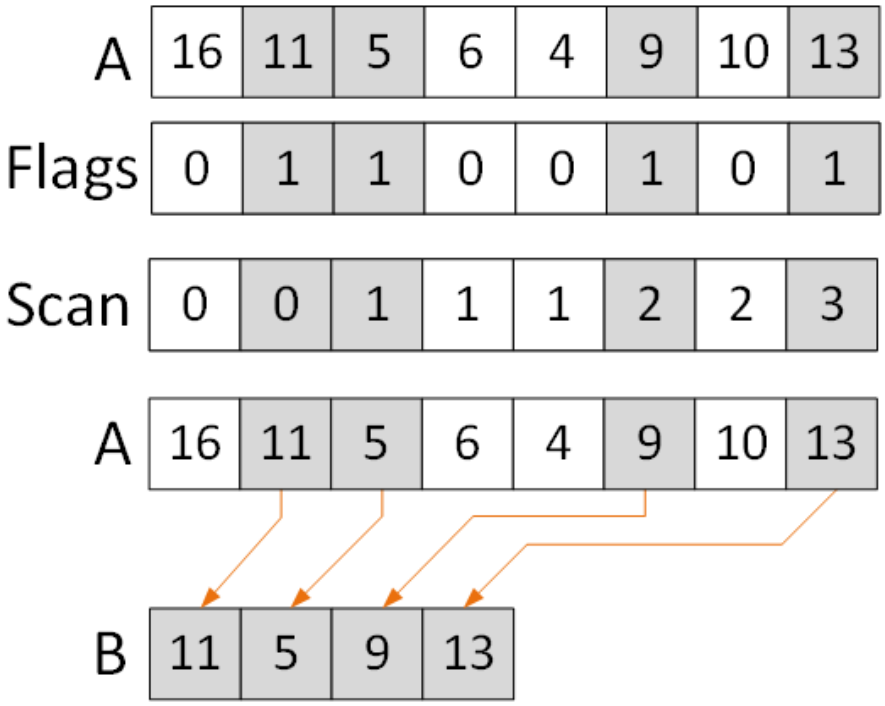

Figure 2.4: How stream compaction works (Scan and Scatter)

Data: input array A and scanned flag array F, both of length n
Result: output stream compacted array B
foreach *i=0...n-1 in parallel* **do**
  if *isFlagged(A[i])* **then**
    B[F[i]] = A[i];
  end
end

Algorithm 3: Sequential Sum

2.4.1 CUDA GPU Architecture

CUDA divides the computer into host and device. The host refers to the CPU and all memory it has access to, and the device refers to the GPU and its memory banks. Memory transfers between the host and GPU are started and managed by the host.

A piece of code that runs in parallel on the device is called a kernel. Figure 2.5 shows how kernels are organized. Each individual running process is called a "thread". Threads are organized into "blocks" of up to 1024 threads on modern architectures. Blocks are then grouped into "grids". Within each block, threads are executed in groups of 32 threads called "warps". Every thread in a single warp executes simultaneously on a single processor core. The device memory architecture is depicted in Figure 2.6. Global memory, constant memory, and texture memory are all accessible by any thread in the kernel. The primary difference between these memories . Every thread also has access to a limited amount of memory which is shared between all threads in a block. Shared memory is much faster than global memory access, and can be used to accelerate kernels that need to share data between threads locally. Finally each thread has access to its own local register memory.

2.4.2 Optimizing CUDA Code

Optimizing CUDA kernels is very application dependent and difficult to provide a universal manual for. However, there are several key underlying GPU architecture features that should be highlighted which can have a dramatic impact on performance.

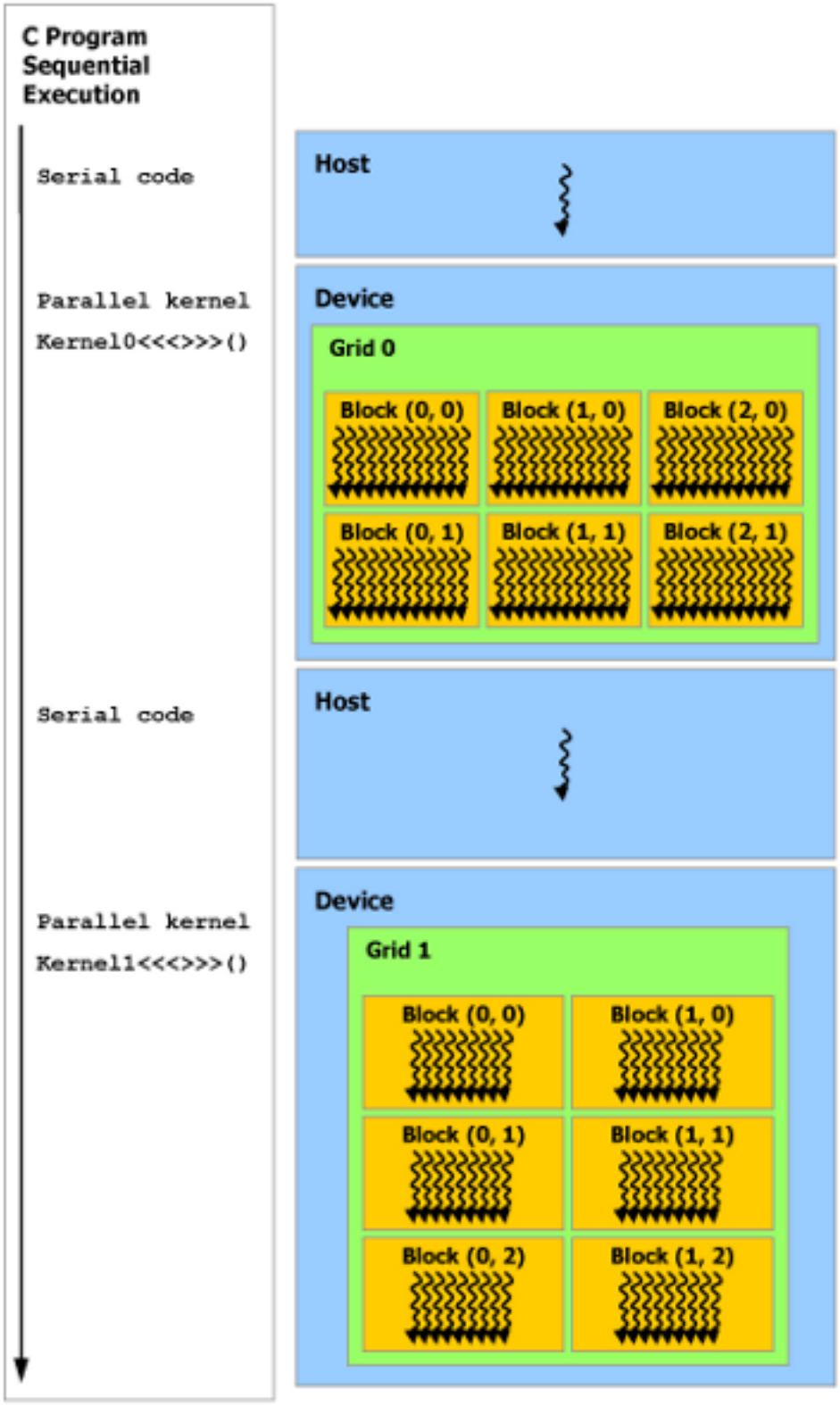

Figure 2.5: CUDA Kernel Organization. Reproduced from [25]

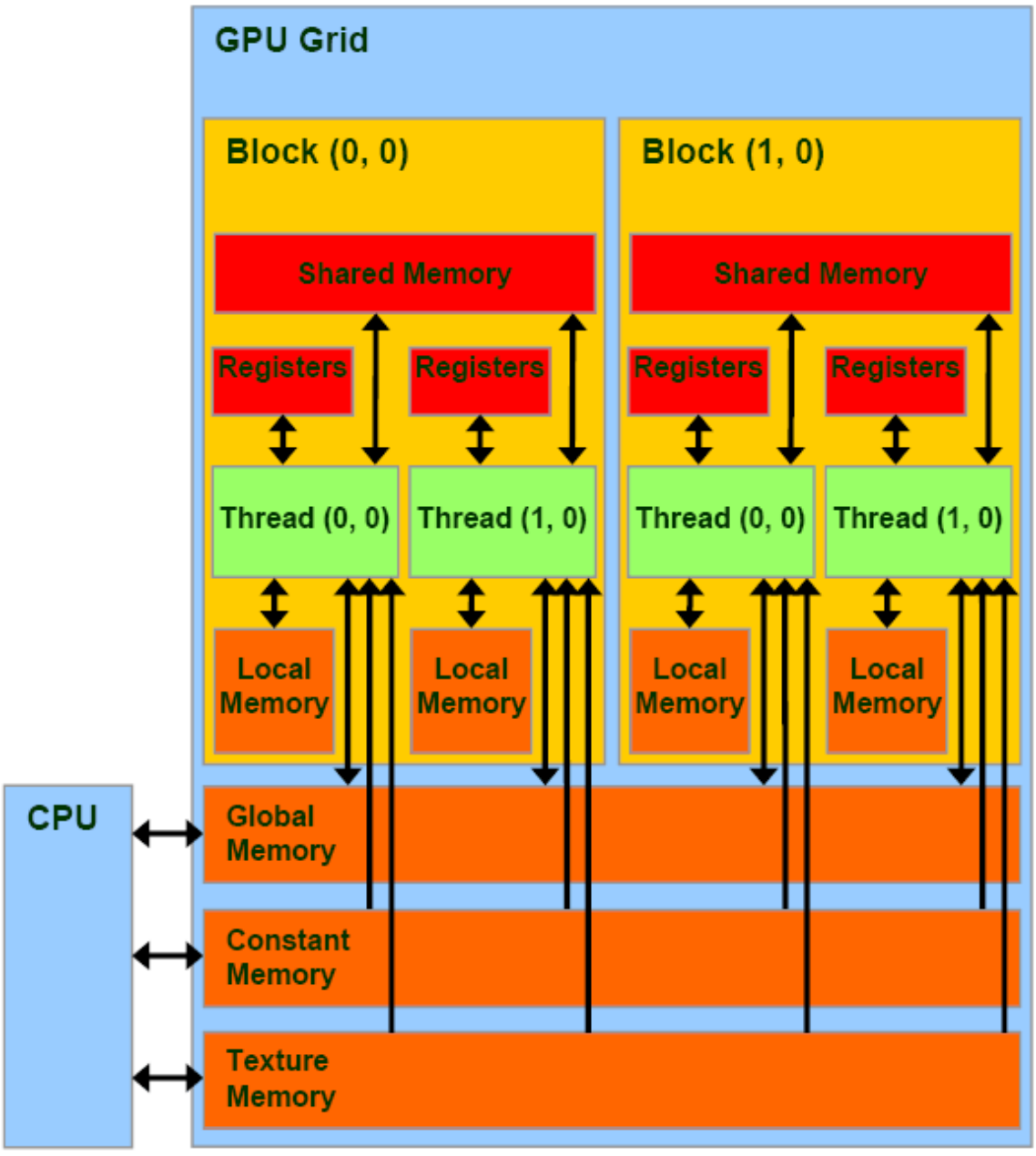

Figure 2.6: CUDA Memory Organization. Reproduced from [39]

Global Memory Coalesced Access Global memory is designed in such a way that each half warp can load 64 consecutive bytes in memory as a single parallel read, provided the memory chunk is also 64-byte aligned. This is just enough memory to load an array of 16 consecutively floats at once (one per thread per half warp). To achieve peak memory bandwidth, reads and writes from global DRAM should be grouped into large consecutive chunks of memory. Because of this, it is sometimes extremely beneficial to utilized structure of array (SoA) data storage schemes as opposed to the more typical array of structures model (AoS).

Shared Memory Optimization Shared memory is much faster than global memory, so it can be instrumental in optimizing kernels. For example, instead of performing a reduction or scan in global memory, the entire array can be loaded into shared memory in a single read operation. The operation would then be performed in shared memory and the results written back in a single coalesced DRAM write. However, shared memory is limited. If too much is used, the number of blocks that can be simultaneously executed on a multiprocessor could go down, preventing the kernel from fully utilizing the GPU hardware. Also, this provides motivation for restricting the size of input data. If all of the data to be processed can be loaded into a single block, shared memory becomes much easier to utilize.

Shared memory is divided into banks, where successive 32-bit words are assigned to successive banks. Shared memory access is done per bank, so if every thread in a warp reads from a different bank, regardless of the ordering, no bank conflicts occur. If two or more threads try to read from the same bank simultaneously, the reads are

serialized which reduced the shared memory bandwidth utilization. The exception to this rule is when all threads read from the same bank. This triggers a special broadcast mode that has no bandwidth penalty.

Warp Partitioning and Divergent Code As mentioned above, threads are actually executed in groups of 32 threads called warps. All threads in a warp take the same amount of time to operate. So if only one thread in a warp is actually doing any work, 31 threads will be idle and waste GPU hardware. Minimizing intra-warp code divergence in will go a long way towards fully utilizing hardware. Also, if possible retire entire warps early to free up resources for warps in the execution queue.

Chapter 3

Problem and Approach

Ultimately my goal is to create a software suite that will take RGB-D data from a Kinect or equivalent sensor and add the new information from each successive frame into a mesh-based representation of the world (ideally in real time). The target use case for the system is indoor environments. Human built environments tend to be dominated by planar surfaces, which if segmented out from the image can be processed and stored very efficiently. Since even planar segments with tattered edges require very few triangles to accurately represent them, the hope is that the majority of the scene can be stored with relatively few triangles.

To accomplish all of this in real time, most of the processing will be maintained on the GPU and host-device memory transfers will be minimized. Keeping the bulk of the pipeline on the GPU has many challenges. Existing algorithms have to be heavily modified or re-imagined to run optimally in a data parallel environment. To avoid costly synchronization delays, most of the pipeline must also be designed to accommodate anomalous data without CPU intervention or even acknowledgement.

GPU architecture does not offer a simple way to return results from a kernel, so returning flags for success or failure is difficult to accomplish efficiently.

This thesis presents an implementation of a large portion of the planned pipeline. The pipeline segments out planar segments and converts them to textured meshes using a QuadTree based triangulation scheme.

3.1 High Level System Design

Figure 3.1 shows the planned pipeline at the highest level of organization. The haloed blocks represent the portions of the pipeline that are implemented in this thesis. The other blocks are provided for context, as some of the design features of this thesis would make little sense without the end goal in mind. For the remainder of this chapter I will describe the function of each module in Figure 3.1. The specific implementation of each module will be explained in much greater detail in Chapter 4.

3.1.1 New RGB-D Image

This thesis presents a custom open-source highly modular framework which allows the pipeline to take its RGB-D data from a variety of sources without knowing the implementation details. Currently the framework can only read from a Kinect via OpenNI or from a log generated by the logging tool I developed with the same framework. However, the modular nature of the code allows new data sources to be very easily incorporated.

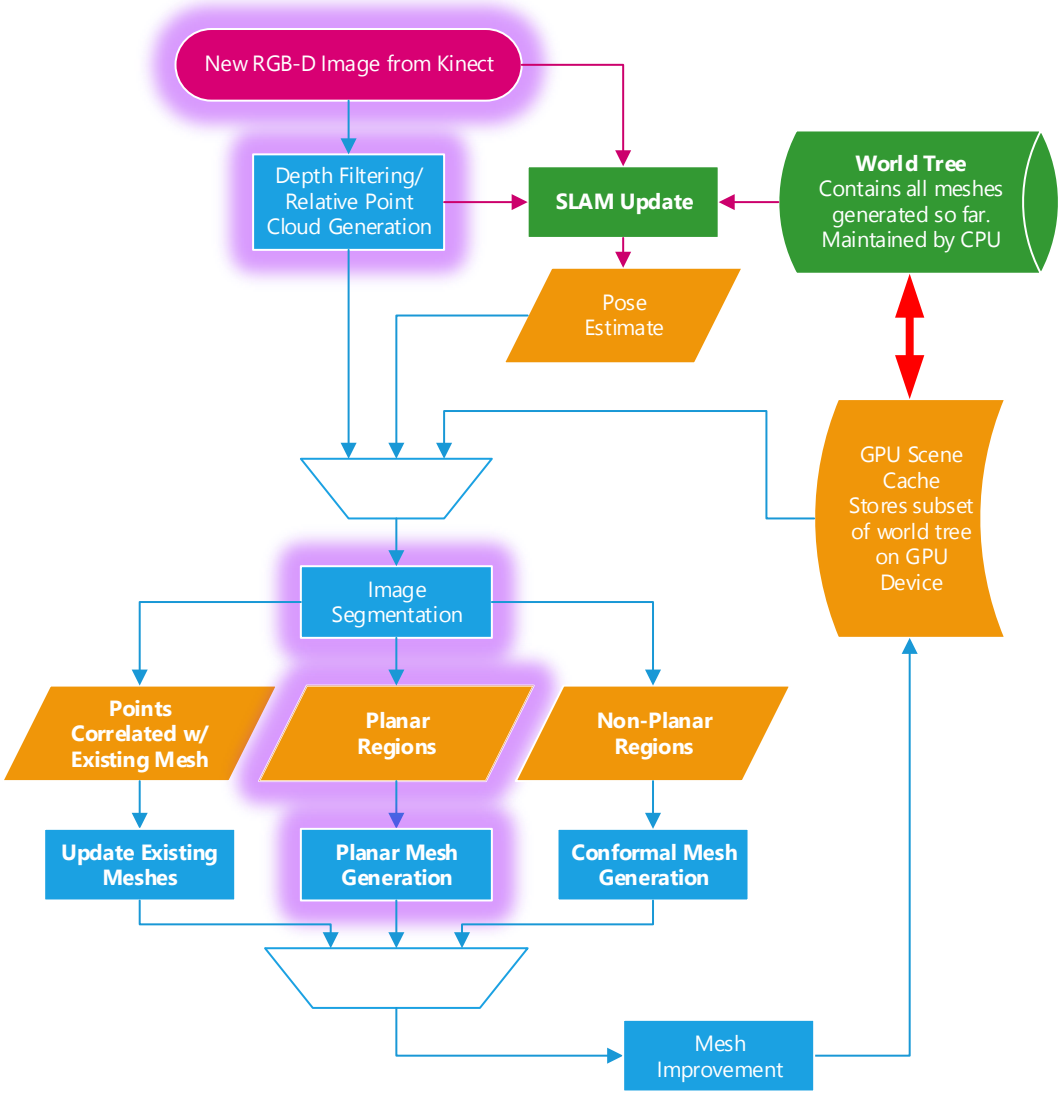

Figure 3.1: Planned pipeline design. Highlighted blocks represent this thesis.

3.1.2 World Tree

The world tree will be maintained in main computer memory by the CPU. It will store the complete mesh representation of the world in a hierarchical tree data structure similar to a scene graph. Unlike a generic scene graph, this structure will be maintained as a strict tree. Each leaf node will contain a single mesh object. Each mesh's component vertices will be stored in an object local reference frame. Objects will be transformed into world space through a series of transformations stored in each node of the tree. This system allows objects to be easily grouped for efficient rendering and simple manipulation.

3.1.3 GPU World Cache

To reduce data redundancy and GPU memory usage, the GPU will only have direct access to a device cached subset of the world tree. The optimal heuristics for maintaining the integrity and efficiency have not been developed yet, but the goal is to maintain a GPU copy of all known meshes in the Kinect's current field of view and those likely to come into view in the next few frames. The cache synchronization with the world tree and subset selection will be handled by the CPU.

3.1.4 Input Processing and SLAM

The raw RGB-D image from the kinect sensor is first passed to the GPU so the data can be filtered and processed in parallel. Once the data is processed, the SLAM algorithm will complete its pose estimation processing, combining the current world

model as stored in the world tree and the 3D features from the newly generated point cloud.

3.1.5 Image Segmentation

Using a combination of information from the processed RGB-D frame, the current camera pose estimate, and the GPU cached world model, each point in the frame's point cloud will be segmented into three categories: points that correspond to parts of the existing mesh world, uncorrelated planar regions, and uncorrelated non-planar regions. Each category will be processed very differently at the next stage, so generating a robust segmentation will be crucial. Within each category, individual regions will be indexed so they can be easily processed in a region parallel manner for the meshing stage.

For this thesis, only planar regions are detected.

3.1.6 Meshing

Each region generated by the segmentation stage will be processed in parallel and in a very different way.

Correlated Points

Points determined to correspond to existing meshes will be used to update the existing meshes. Points will be used to add new vertices to a mesh as needed for added detail or to increase confidence in existing vertex estimates. This process allows the pipeline to take full advantage of higher resolution/closer capture frames of previously seen

meshes. RGB data will also be used to update model texturing, allowing for texture quality improvement.

Planar Regions

Planar regions are projected to a best-fit true plane and triangulated using a QuadTree-Based (QTB) triangulation algorithm inspired by previous efforts[18, 19]. The method has been demonstrated to efficiently decimate points clouds and dramatically simplifies triangulation and texture generations. In the future, if a planar region is determined to overlap with an existing plane, the existing quad-tree will be expanded to incorporate the new information.

Non-planar Regions

The remaining regions will be triangulated and converted into a new non-planar mesh object using simple polygon based triangulation method. Since conforming texture generation with arbitrary surfaces is a difficult problem to solve in real time with minimal distortion, non-planar meshes will be vertex colored. With a detailed enough triangulation, the visual quality of the of vertex shading is favorably comparable to a textured mesh. The framework would allow for simple addition of textures later if a texture would be a more efficient representation.

3.1.7 Mesh Improvement

Once individual mesh regions have been generated, meshes will be checked for easy optimizations and the potential for mesh merging. The goal of this stage is to reduce

the number of vertices and distinct objects. This step is optional and not required for the pipeline to function, but it can help to simplify the representation and avoid unnecessarily fragmenting the world map.

Chapter 4

Implementation

This chapter will explore in great detail the functionality and implementation of this thesis's pipeline. As a reminder, the pipeline consists of the highlighted elements in Figure 3.1. Each section of this chapter describes the implementation of one of these functional blocks, as well as a brief exposition on the visualization system. Each section will be accompanied by a figure illustrating the end product of that stage of the pipeline using the same example data.

4.1 RGB-D Framework Library

A crucial component of the pipeline is being able to easily collect RGB-D sensor data from a variety of sources in such a way that the origin of the data is hidden from the remainder of the pipeline. To that end, a highly modular and easily extensible event based framework library was built to seamlessly convert the native data formats and streaming behavior of different sensors. Figure 4.2 provides an overview of

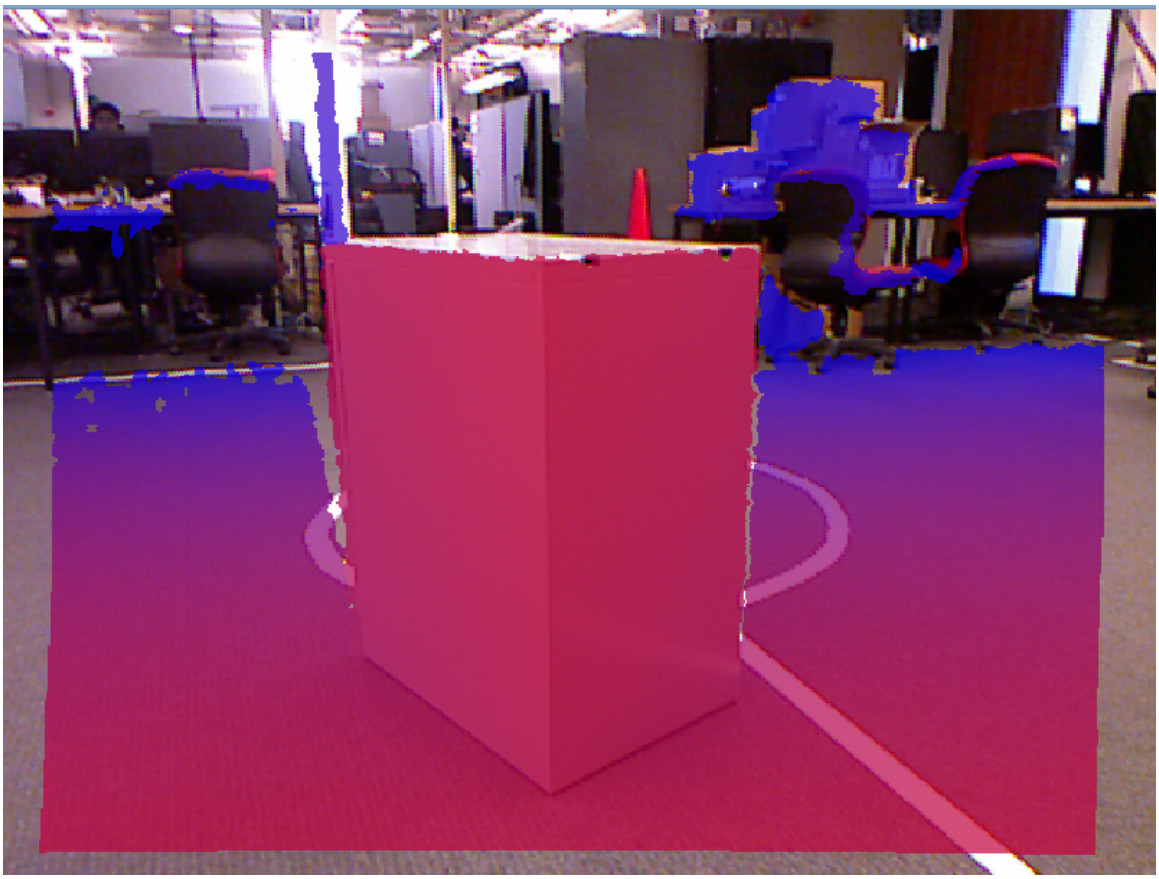

Figure 4.1: RGB image with depth overlay. Red is closer to the camera, blue is further away. Represents the output of the RGBD Framework

the framework organization. The application code deals directly with four primary classes: RGBDDevice, RGBDFrame, Event Listeners, and FrameLogger. The final output of this framework is a color image and a depth image that are registered in such a way that the pixels have a one-to-one correspondence as shown in Figure 4.1.

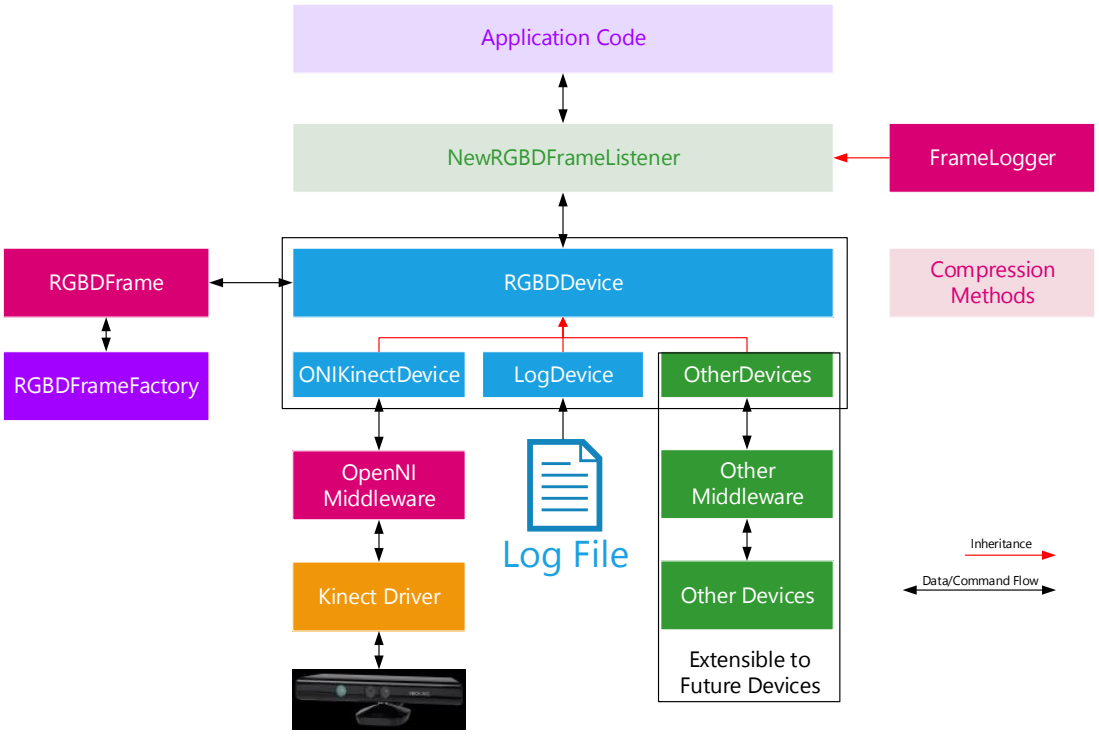

Figure 4.2: RGB-D framework architecture

4.1.1 RGBDDevice

The primary interface between the framework and the rest of the application is the abstract class RGBDDevice. RGBDDevice provides an abstract interface including a data stream management API, access to device properties like camera intrinsics and resolution, and event listener registration API. Interfaces for specific devices like the Kinect can be implemented by creating a subclass of RGBDDevice and implementing

the abstract methods. The application can then instantiate the desired subclass with device specific initialization parameters and the remainder of the pipeline can be completely agnostic to the underlying nature of the data source, since all RGBDDevices provide the same format RGBDFrame through the same event architecture regardless of the native device formatting. The current framework contains two subclasses: ONIKinectDevice and LogDevice.

ONIKinectDevice ONIKinectDevice implements a connection specific to the Microsoft Kinect using OpenNI as middleware. The implementation utilizes OpenNI's event-based interface to receive data from the sensor. Two streams for the color and depth data are created and registered with listeners internal to ONIKinectDevice. When a new depth or color frame is received from OpenNI, the data is copied and repackaged into an RGBDFrame format and a new thread is launched which passes the RGBDFrame to each registered listener.

LogDevice LogDevice replays a data log created using the FrameLogger class. Because of the way that the FrameLogger records the data, any device that can be implemented as an RGBDDevice can be recorded and played back using a LogDevice. The framework even allows logging data from a LogDevice. This class allows experiments to be performed very easily on prerecorded data without having to alter the behavior of the pipeline at all.

4.1.2 RGBDFrame

The RGBDFrame is a simple data structure that provides a consistent data formatting for users of the framework. Each frame consists of two managed shared pointers (implemented using boost::shared_array); one points to the color data and the other points to depth data. Color data is stored in a 24-bit RGB format (1 byte for each component red, green, blue). Depth data is stored in an unsigned 16-bit integer where the least significant bit represents 1mm of resolution (i.e. the value 1024 corresponds to a depth of 1.024m). The frame also has two flags indicating the validity of the depth and color arrays, since not every frame will have both.

RGBDFrames are created using a factory design paradigm. The RGBDFrameFactory generates managed pointers which refer to RGBDFrames of a given resolution. Because of the nature of shared pointers, when the last reference to the frames or its component arrays goes out of scope, the frame will delete itself. Because of this, the frames can easily be handed off to the application with no need for the application to be aware of the finer points of the RGBDFramework's memory management scheme. Since roughly 46MB of frame dedicated memory will be allocated every second by the framework, clean, simple memory management is essential.

4.1.3 Event Listeners

The RGBDDevice provides a suite of event listeners that the application code can register listeners for. Four events can be emitted by the RGBDDevice: DeviceConnected, DeviceDisconnected, DeviceMessage, and NewRGBDFrame.

DeviceConnected This event is triggered when the underlying device is successfully connected to the RGBDDevice. This usually happens upon success of the RGBDDevice::connect method.

DeviceDisconnected This event is triggered when the underlying device is disconnected. This happens either upon calling RGBDDevice::disconnect, or when a fatal communication failure occurs.

DeviceMessage This is a special event that allows the device to pass human readable text descriptors to the application for display in the appropriate place. This is useful for debugging or for registering more unique events for a specific device.

NewRGBDFrame This event actually transfers the latest RGBDFrame from the sensor to the application. It passes a shared_ptr to the all registered listeners.

4.1.4 FrameLogger

The FrameLogger class is actually just a special subclass of the NewRGBDFrameListener class. To record the output of any RGBDDevice to a directory, the FrameLogger just needs to be given an output directory using setOutputDirectory. Once that has been done, all that is required to log the output is to call the FrameLogger's startRecording method, which registers the FrameLogger as a NewRGBDFrameListener with the target RGBDDevice. As the logger receives new frames (which can happen in parallel with the primary pipeline), they are tagged with a counter id and saved to the output directory as raw binary files. Some options are also available for

data compression.

4.2 Filtering and Point Cloud Generation

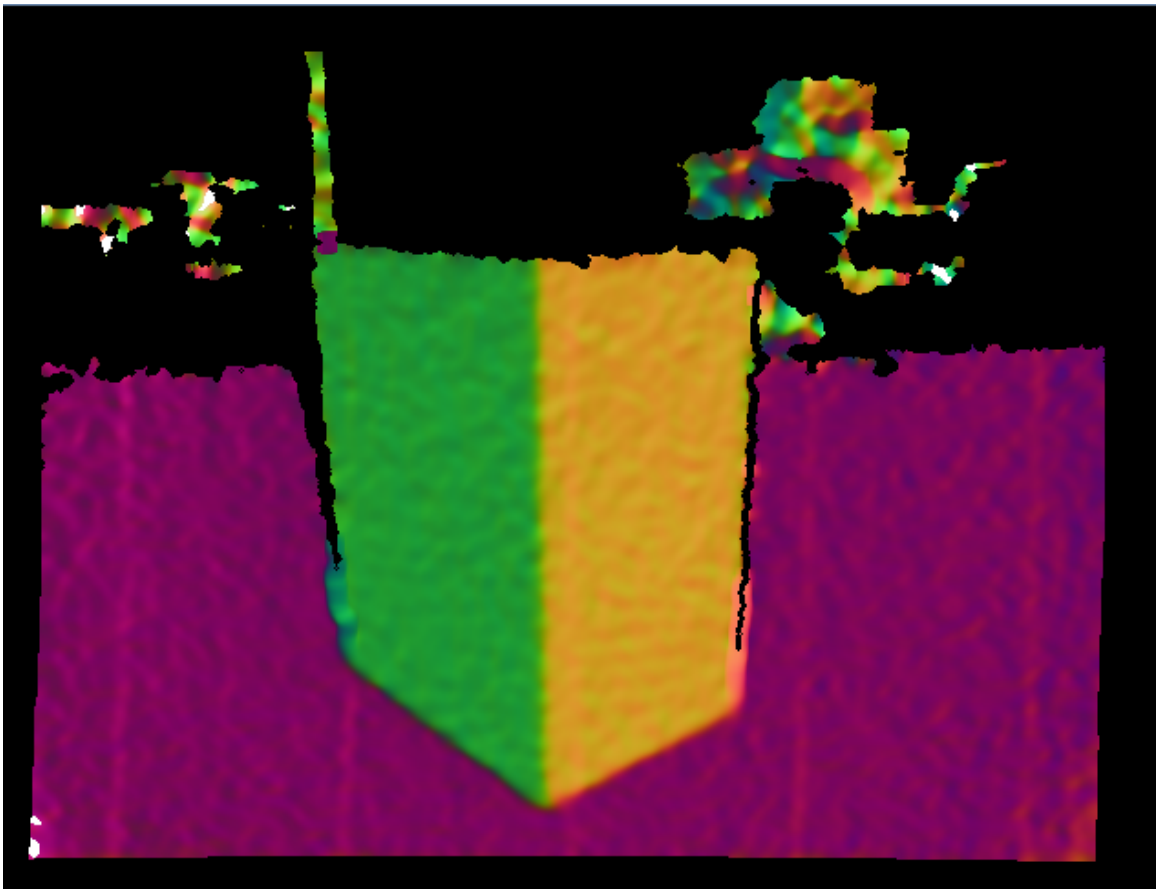

Figure 4.3: Visualization of surface normals, where components of the normal vector are mapped from to rgb. Example of output from preprocessing stage.

Once the raw RGB-D frame has been received, it is pushed to GPU memory. This is the only downstream memory transfer that happens during this entire thesis. Once the memory transfer is complete, the next step is to convert the RGBDFrame format into a floating point representation that is easier to manipulate and perform other preprocessing steps like depth filtering and normal estimation. The data is also

converted to a structure of arrays format to improve memory coherence. Figure 4.4 shows the entire preprocessing system's program flow.

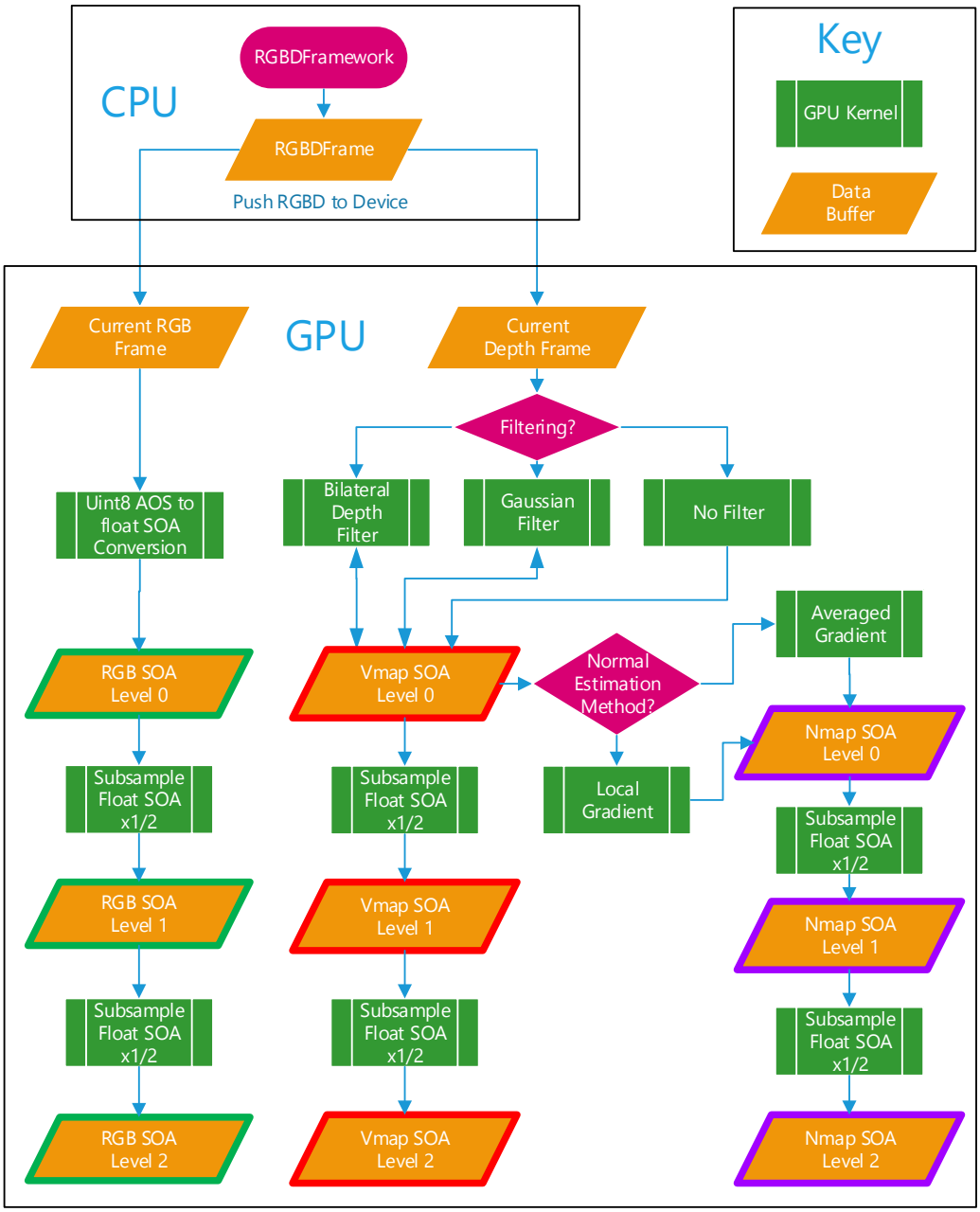

Figure 4.4: Preprocessing pipeline diagram.

RGB Data Processing The processing for the RGB data is trivial. A kernel converts the 0-255 integer representation of the color data into a 0.0-1.0 floating point representation stored in a structure of arrays (SOA) format for memory coherence. This SOA is then subsampled twice by a factor of two each time, resulting in two additional levels of image resolution. This is useful for later stages in the pipeline that do not require full resolution to function and can therefore be run much faster on the lower resolution image.

Depth Filtering The pipeline includes three options for pre-filtering the depth data: no filter, a separable Gaussian filter, and a separable bilateral filter approximation[27]. The no filter kernel simply performs the inverse camera projection to convert the depth data into an ordered point cloud stored in the vertex map (VMap) buffer. The formula for this projection is based on the camera intrinsic parameters $\{f_x, f_y, c_x, c_y\}$ and the pixel coordinates (u, v).

$$v_x = (u - c_x) * depth / f_x$$

$$v_y = (v - c_y) * depth / f_y$$

$$v_z = depth$$

Pixels with invalid depth data are stored as NaN to allow easy validation later in the pipeline.

The Gaussian kernel performs the same projection, but also applies a separated

implementation of Gaussian filter ($radius = 3pixels, \sigma = 2.0$) to the depth data before calculating the projection. This helps remove much of the depth quantization and sensor noise that makes normal estimation difficult. However, the Gaussian filter does not preserve edges, so crucial discontinuities in the image are blurred together.

To solve this problem, a bilateral filter is used instead. In addition to the spatial Gaussian term that weights pixels by their screen distance from the center pixel, bilateral filters also apply a Gaussian weight to the difference in intensity. In this way, pixels that have a vastly different depth value will have a lower weight and hard edges can be more easily preserved.

As with the color data, the vertex map is subsampled into a resolution pyramid.

Normal Estimation Surface normals are locally estimated for each valid point. Several methods exist for estimating point cloud normals, but the simplest to apply in an ordered image is a gradient based approach. For each point $\vec{p}(x, y)$ Two vectors $\vec{G_x}$ and $\vec{G_y}$ are created from the neighboring points.

$$\vec{G_x}(x, y) = \vec{p}(x + 1, y) - \vec{p}(x - 1, y)$$

$$\vec{G_y}(x, y) = \vec{p}(x, y + 1) - \vec{p}(x, y - 1)$$

The normalized cross product of these two vectors is then taken to be the point normal.

$$\vec{N}(x, y) = \frac{\vec{G_x}(x, y) \times \vec{G_y}(x, y)}{|\vec{G_x}(x, y) \times \vec{G_y}(x, y)|}$$

Since the sign of this normal is ambiguous, all normals are flipped so that they face the viewpoint. The normal faces the viewpoint when the following condition is met:

$$\vec{N} \cdot (\vec{p} - \vec{p}_{eye}) < 0$$

Since in the coordinate frame used at this stage has the eye at the origin looking along the $+z$ axis, this condition can be simplified to:

$$\vec{N} \cdot \vec{p} < 0$$

If the estimated normal violates this condition, the negative of the normal is stored.

While this simple normal estimate works reasonably well, it can be improved by applying a smoothing filter to the gradient images G_x and G_y. A separable Gaussian kernel is applied to each gradient image in turn, and then the cross product is performed. Figure 4.5 compares the various combinations of depth filters and normal estimation methods on the normal estimates. Notice how the Gaussian filter fills in the gaps between the cabinet and the floor, but the bilateral filter shows a very clean break. The final pipeline uses both the bilateral filter and the gradient smoothing techniques. And once again, the normals are subsampled and stored in a resolution pyramid.

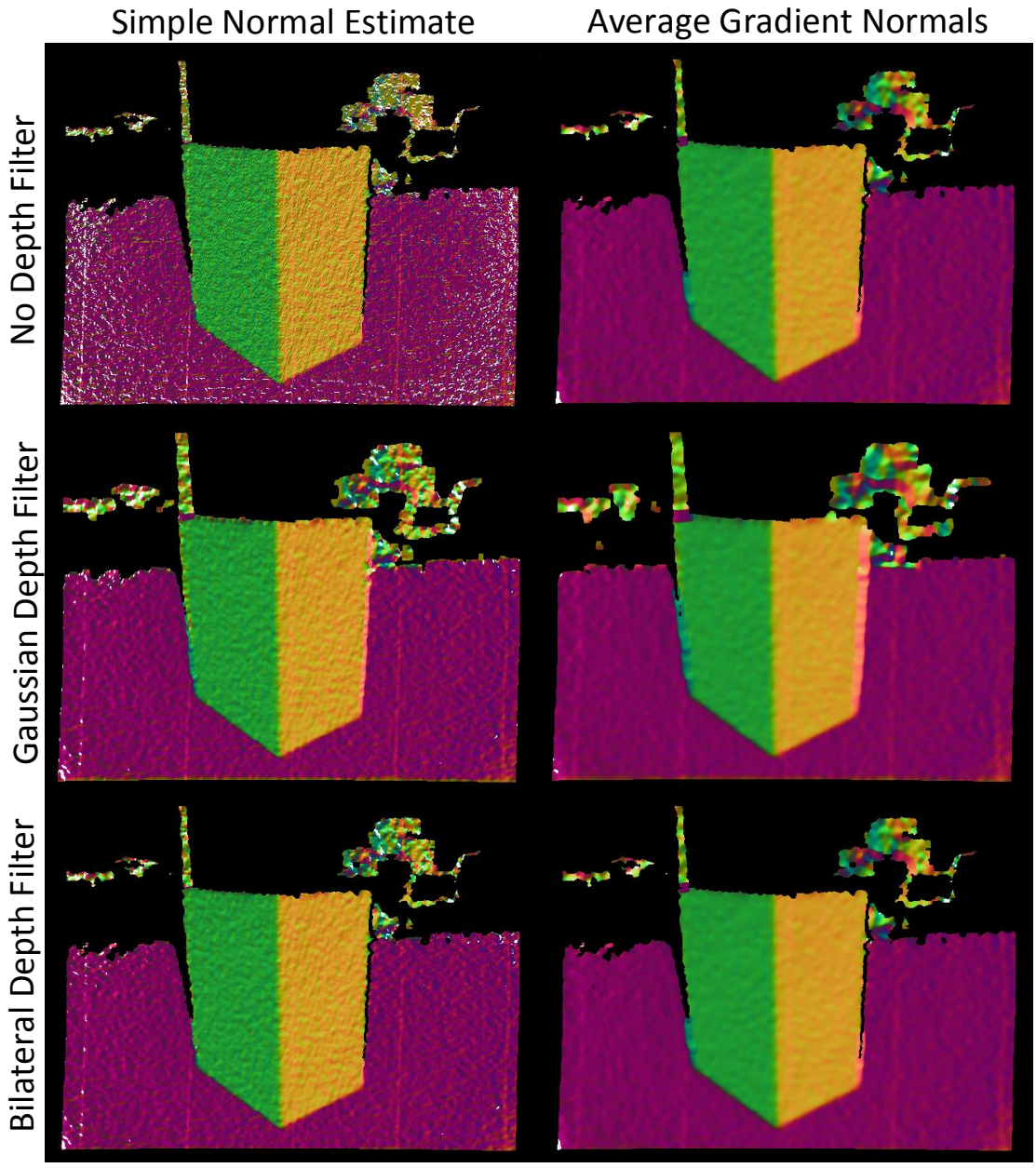

Figure 4.5: Comparison of the normals estimated using various methods and filters

4.3 Plane Segmentation

The plane segmentation portion of the pipeline is easily the most complicated (Figure 4.7). It takes the normals and positions generated by the preprocessing stage as input and outputs a buffer the same size as the original image where each pixel is the index of the detected plane the pixel belongs to, or -1 if the pixel doesn't belong to a plane. The results can be visualized with a random colorization as in Figure 4.8.

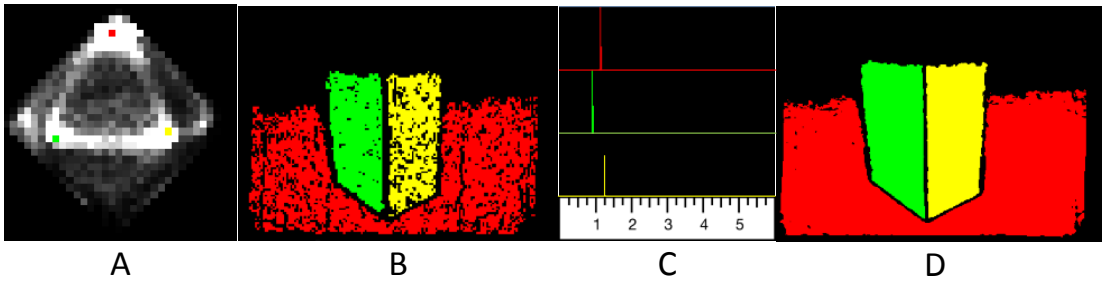

Figure 4.6: Buffers from various stages of the segmentation pipeline. A) The 2D normal histogram with 3 identified peaks. B) Normal Segmented Image. C) Distance histograms for refinement. D) Final Segmentation.

The overall approach to image segmentation is inspired by Holz *et al.*[11]. The initial step is to detect clusters of similarly oriented surface normals. A two-dimensional histogram of the normals is created, and a collection of dominant peaks is detected (Figure 4.6A). Once these peaks have been detected, the initial segmentation pass segments all normals that are very close to the peak (Figure 4.6B). Using only the points from the detected segments, a distance histogram is computed for each peak, where the distance value of each point is computed using the dot product of the peak normal and the point's position in camera space(Figure 4.6C). In addition to refining the segmentation in distance space, some basic statistics about the detected planes

are collected. Because the original normal estimate from the original histogram can be very coarse, the plane stats are used to realign the normal peaks more accurately. The entire inner loop is run again with the accurate normals, yielding cleaner results in complex scenes. After the second run through, the planes are finalized and the final segmentation buffer is updated (Figure 4.6D). Optionally, this entire process can be run multiple times excluding previously segmented pixels to detect planes that were previously obscured by larger noisy peaks.

4.3.1 Normal Histogram Generation and Peak Detection

Holz[11] used a 3D voxel grid to discretize normal space, although they also experimented with a two-dimensional (ϕ, θ) space. They then merged clusters with their neighbors to come up with their final segmentation. Unfortunately, this approach does not map well to the GPU. As an alternative, I created a very simple index mapping $[n_x, n_y, n_z] \rightarrow [\frac{\cos^{-1}(n_x)}{\pi} * numBinsX, \frac{\cos^{-1}(n_y)}{\pi} * numBinsY]$ where $numBinsX$ and $numBinsY$ are the resolution of the histogram. Instead of doing sophisticated clustering, I perform simple iterative global maximum detection using a high speed parallel reduction algorithm. After each peak is detected, a region of histogram space surrounding the peak is cleared to avoid just off max peaks from being recorded. This iteration continues until the maximum number of peaks have been detected or no peak is above the minimum threshold count. Each detected peak index is then reverse mapped to a peak normal vector. To improve accuracy slightly, the normal is computed using an average of the neighboring normals weighted by the histogram

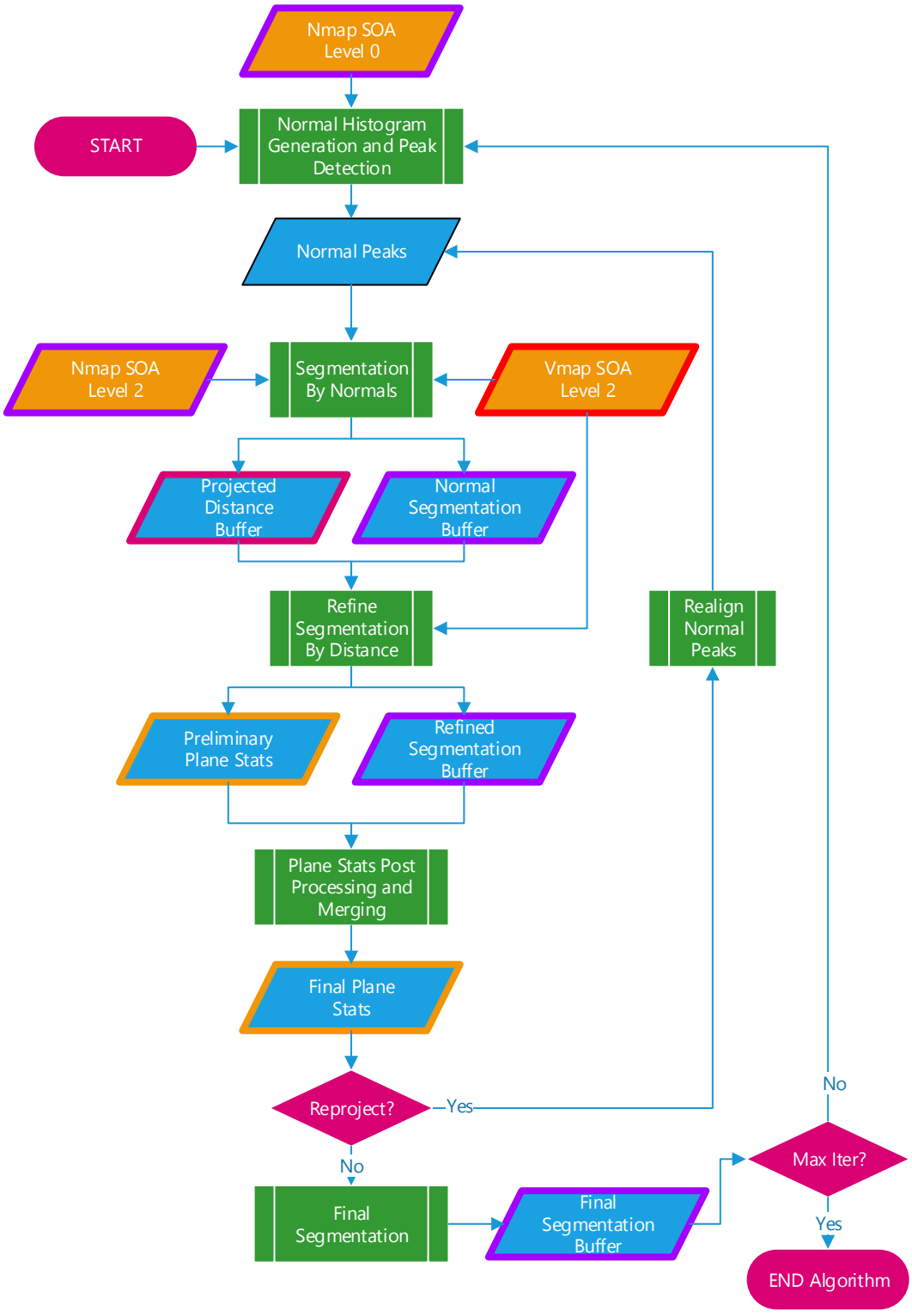

Figure 4.7: Plane segmentation pipeline diagram.

Figure 4.8: Random colorization of detected planes. Example of output from segmentation stage.

count.

One important thing to note about histogram generation in parallel is the runtime performance is heavily data dependent. If two threads running concurrently try to increment the same histogram bin, a race condition occurs and the result is corrupted. This requires using atomic operations to serialize these additions. As a result, the worst case scenario is a single noiseless plane that fills the entire image. By contrast, calculating the histogram on the CPU would be constant runtime. In my experiments, the CPU runtime plus memory transfers took longer than most GPU scenarios, but the ultimately it is an engineering tradeoff. Alternatively, the resolution of the histogram can be increased, which statistically decreases the number of write conflicts. However, this makes peak detection slightly more complicated to implement efficiently. Again, tradeoffs to be considered another time.

4.3.2 Segmentation By Normals

This step is fairly self explanatory. Using the peaks detected in the previous step, if the normal of a point deviates from a peak normal by less than 5 degrees, the pixel is assigned to that peak. This step also computes the projected distance of the point for use in the next step. The distance calculation is derived from the plane equation:

$$a * x + b * y + c * z = dist$$

Using the corresponding peak normal as the constant parameters $[a, b, c]$ and the point's camera space position as the $[x, y, z]$ values, $dist$ is simply the dot product

$$dist = \vec{N}_{peak} \cdot \vec{p}_{cam}$$

4.3.3 Refine Segmentation by Distance

This stage serves two functions. First, multiple planes can have the same normal but different distance offsets. A door recessed into a wall is a very common example in human environments. Using the projected distance along the normal to a parallel plane passing the origin and creating a histogram of these distances (Figure 4.6C), each plane should stand out as a unique peak. However, if the plane normal estimate from the peak detection step was too far off from the ground truth plane normal then these peaks will be smeared and difficult to detect. This is the primary reason that the inner loop is run twice, once to have a shot at detecting some portion of a plane to get a truer normal, and a second run with the true normal to find the actual planes. Without this step, large planes were often broken into several smaller strips or planes that were not offset by much were merged incorrectly.

Once the distance peaks are located, the previous segmentation is refined with only points corresponding to distance peaks remaining. For efficiency, the same kernel that updates the segmenation buffer also collects the first pass plane statistics. Each plane's pixel count, centroid, and scatter matrix are computed in this stage, though the values are just stored as accumulated sums. The data collected here will be

processed in the next stage.

4.3.4 Plane Statistics Processing and Plane Merging

This stage calculates various defining features of the planes. The statistics collected were inspired by Biswas*et al.* work on a RANSAC based plane filtering algorithm[3]. The plane centroid is computed as:

$$\bar{p} = \frac{1}{n} * \sum_{p_i \in P} p_i$$

where n is the number of points in the planar segment. To compute the plane normal and tangent, the scatter matrix of the plane could be computed as:

$$S = \sum_{p_i \in P} (p_i - \bar{p})(p_i - \bar{p})^T$$

. However, merging the planes becomes much simpler if the matrix can be decoupled from the centroid[3]. So let

$$S_1 = \frac{1}{n} \begin{bmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i z_i \\ \sum y_i x_i & \sum y_i y_i & \sum y_i z_i \\ \sum z_i x_i & \sum z_i y_i & \sum z_i z_i \end{bmatrix}$$

and

$$S_2 = \begin{bmatrix} \bar{x}\bar{x} & \bar{x}\bar{y} & \bar{x}\bar{z} \\ \bar{y}\bar{x} & \bar{y}\bar{y} & \bar{y}\bar{z} \\ \bar{z}\bar{x} & \bar{z}\bar{y} & \bar{z}\bar{z} \end{bmatrix}$$

then the matrix

$$S' = S_1 - S_2$$

is a normalized representation of the scatter matrix that can be used to compute the plane normal and tangent. Planes can then be conditionally merged by the procedure outlined by Biswas[3]. The eigenvector corresponding to the smallest eigenvalue of S' is taken to be the plane normal, while one of the other eigenvectors is chosen to be the tangent vector. The tangent vector is chosen such that all planes are oriented in roughly the same way in screen space, with the tangent vector roughly corresponding to the $+y$ axis.

4.3.5 Final Segmentation

Once all the planes have been finalized, all points in the image are considered and evaluated to find the best fit. Points are assigned to planes using the following minimization procedure:

$$PlaneId_i = \arg\min_{plane}[p_i \cdot \vec{N}_{plane}]$$

s.t.

$$|\vec{N}_i \cdot \vec{N}_{plane}| > angleThreshold$$

and

$$p_i \cdot \vec{N}_{plane} < distThreshold$$

The angle threshold can generally be set very wide, as much as $cos(\pi * 15/180)$, since the dominating factor will be distance to the plane. A typical distance threshold should be around 0.015m to allow for noise.

4.4 Planar Mesh Generation

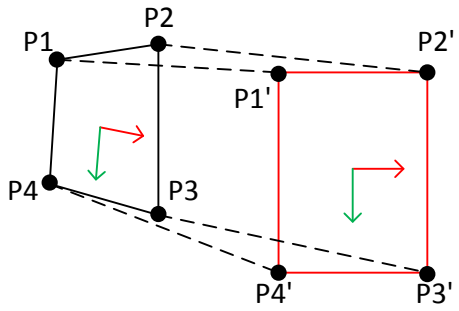

Figure 4.9: Projective transform example

Once the individual planes have been segmented, each plane is reduced using a parallelized version of QuadTree Based triangulation[18]. The first step is to calculate the inverse projection of the plane onto a flat surface that is uniformly scaled in X and Y as in Figure 4.9. The first step is to compute an axis-aligned bounding box (AABB) for each plane in the 2D coordinate frame defined by the plane's tangent and bitangent vectors $\vec{T}$ and $\vec{B}$. The plane's normal and tangent vectors are stored in the plane statistics, and the bitangent can be computed from the cross product $\vec{B} = \vec{N} \times \vec{T}$.

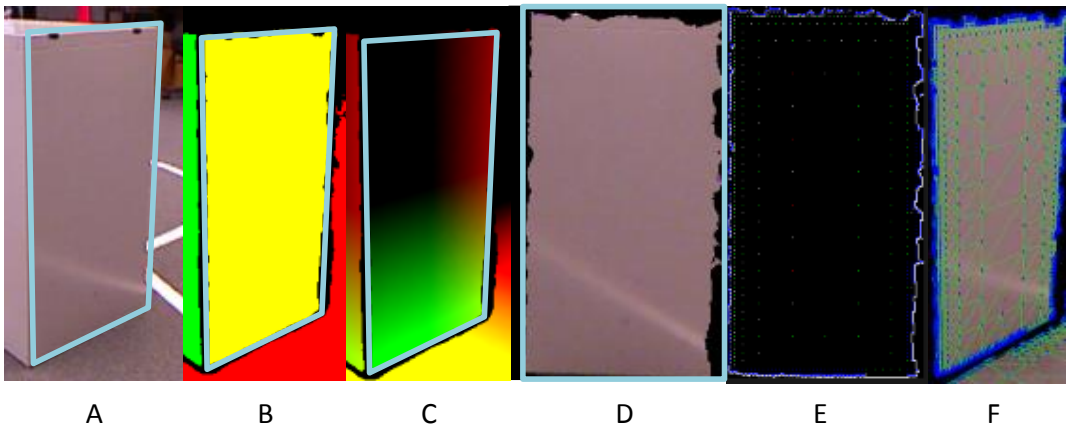

Figure 4.10: Stages of QuadTree Generation: A) Original RGB Image with AABB imposed, B) Plane Segmentation with AABB imposed, C) (S_x, S_y) coordinate system visualized. Green intensity corresponds to $+S_y$ value and red intensity to $+S_x$, D) Flat projected RGB texture with projected AABB, E) QuadTree verticies visualized in flat projection space, F) Final QuadTree mesh visualized with virtual camera

The position of a camera space point p in this coordinate system (S_x, S_y) can be computed as:

$$S_x = \vec{B} \cdot (p - \bar{p})$$

$$S_y = \vec{T} \cdot (p - \bar{p})$$

Using a parallel reduction algorithm, the minimum and maximum values of S_x and S_y for each plane are computed. This provides an AABB in meters for each segment (Figure 4.10C). To compute the projection matrix from camera space to flat space, the screen space coordinate pairs $(u_{source,i}, v_{source,i})$ are computed for each corner using the bounding box and camera intrinsics:

$$p_1 = (AABB_{x,min} * \vec{B}) + (AABB_{y,min} * \vec{T}) + \bar{p}$$

$$p_2 = (AABB_{x,max} * \vec{B}) + (AABB_{y,min} * \vec{T}) + \bar{p}$$

$$p_3 = (AABB_{x,max} * \vec{B}) + (AABB_{y,max} * \vec{T}) + \bar{p}$$

$$p_4 = (AABB_{x,min} * \vec{B}) + (AABB_{y,max} * \vec{T}) + \bar{p}$$

$$u_{source,i} = \frac{p_{i,x} * fx}{p_{i,z}} + cx$$

$$v_{source,i} = \frac{p_{i,y} * fy}{p_{i,z}} + cy$$

The resolution of the final image is selected such that the final projected image will be less than 1024x1024 and resolution given in pixels/meter is a power of 2. This makes textures from various meshes much easier to compare directly by simple up-sampling or down-sampling. Once a resolution has been selected, the destination width and height w_d and h_d in pixels can be computed directly from the AABB dimensions and the pixels/m resolution:

$$w_d = resolution_{px/m} * AABB_{x,max} - AABB_{x,min}$$

$$h_d = resolution_{py/m} * AABB_{y,max} - AABB_{y,min}$$

Now the transform matrix T can be calculated by the following procedure. Let:

$$\begin{bmatrix} u_{s,1} & u_{s,2} & u_{s,3} \\ v_{s,1} & v_{s,2} & v_{s,3} \\ 1 & 1 & 1 \end{bmatrix} * \vec{x} = \begin{bmatrix} u_{s,4} \\ v_{s,4} \\ 1 \end{bmatrix}$$

Solve for $\vec{x} = A^{-1} * b$ and then multiply through the original matrix by $\vec{x}$ such that:

$$A = \begin{bmatrix} x_1 * u_{s,1} & x_1 * u_{s,2} & x_1 * u_{s,3} \\ x_2 * v_{s,1} & x_2 * v_{s,2} & x_2 * v_{s,3} \\ x_3 & x_3 & x_3 \end{bmatrix}$$

This matrix represents a transformation from the source coordinate system to an orthogonal basis vector set. Next the transform from the destination coordinate frame to the basis vector set is computed in a similar fashion.

$$\begin{bmatrix} 0 & w_d & w_d \\ 0 & 0 & h_d \\ 1 & 1 & 1 \end{bmatrix} * \vec{x} = \begin{bmatrix} 0 \\ h_d \\ 1 \end{bmatrix}$$

Solve for $\vec{x} = A^{-1} * b$ and then multiply through the original matrix by $\vec{x}$ such that:

$$A = \begin{bmatrix} 0 & x_1 * w_d & x_1 * w_d \\ 0 & 0 & x_2 * h_d \\ x_3 & x_3 & x_3 \end{bmatrix}$$

Finally, the total transform can be computed

$$T = A * B_{-1}$$

Multiplying this transform by a pixel location in the destination image will compute the corresponding pixel location in the original RGB image after dehomogenization,

allowing for a very simple projection algorithm implementation that is parallel by destination pixel.

$$s = T * [d_x, d_y, 1]^T$$

$$x_{source} = s_x / s_z$$

$$y_{source} = s_y / s_z$$

Now that the texture has been projected to flat space (Figure 4.10D), the parallel QuadTree reduction algorithm is run. To simplify the problem, the QuadTree is aligned to the pixel grid of the flat image. This allows for coherent memory access, simple indexing, and very fast parallel decimation. A degree buffer is initialized with -1 for invalid pixels and 0 for pixels that are a part of the plane segment (Figure 4.11). The degree number of a pixel indicates size of the quad with the upper left corner at that pixel. Algorithm 4 shows how the QuadTree decimation works. As an optimization, the kernel was divided into two phases, one for reduction steps 1 through 16, and another for 32 through 256. This breakdown allows the entire process to be done in 16x16 blocks of shared memory, greatly accelerating the process.

Once the degree buffer has been filled, the vertices are stream compacted and a triangle index buffer is populated linking the compacted vertices. Each vertex is a 4 float vector containing the plane coordinate system (S_x, S_y) coordinates of the point and the (u, v) texture mapping coordinates. Each quad consists of two triangles using CCW winding order, the OpenGL default. These buffers can be directly drawn to the screen using OpenGL triangle elements (Figure 4.12).

Each mesh is then pulled from the GPU along with their generated textures. The transformation matrix H_{plane} from plane space to camera space is also generated at this point so that the plane can be drawn in the correct location on the screen.

$$H_{trans} = \begin{bmatrix} 1 & 0 & 0 & \bar{p}_x \\ 0 & 1 & 0 & \bar{p}_y \\ 0 & 0 & 1 & \bar{p}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_{rotate} = \begin{bmatrix} \vec{B} & \vec{T} & \vec{N} & \vec{0} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_{plane} = H_{trans} * H_{rotate}$$

```
Data: Degree array D with 0 indicating valid pixels, -1 otherwise
Result: Degree array D with QuadTree decimation results
foreach pixel (x,y) in parallel do
    /* First step is special case                                    */
    if D[x][y] == 0 and D[x+1][y] == 0 and D[x][y+1] == 0 and D[x+1][y+1]
    == 0 then
        D[x][y] = 1;
    end
    /* Quadtree decimation loop                                      */
    step = 1;
    while step < 256 do
        /* If pixel correctly aligned for this step                  */
        if x mod (2*step) == 0 and y mod (2*step) == 0 then
            /* If the four corners of this quad all have the correct
               degree                                                */
            if D[x][y] == step and D[x+1][y] == step and D[x][y+1] == step
            and D[x+1][y+1] == step then
                /* Increase the degree of the upper left corner     */
                D[x][y] *= 2;
                /* Clear the other three mid points                 */
                D[x+1][y] = -1;
                D[x][y+1] = -1;
                D[x+1][y+1] = -1;
            end
        end
        step <<= 1;
        syncthreads();
    end
    /* Patch holes.                                                  */
    if D[x][y] > 0 then
        if D[x+D[x][y]][y] < 0 then
            D[x+D[x][y]][y] = 0;
        end
        if D[x][y+D[x][y]] < 0 then
            D[x][y+D[x][y]] = 0;
        end
        if D[x+D[x][y]][y+D[x][y]] < 0 then
            D[x+D[x][y]][y+D[x][y]] = 0;
        end
    end
end
```

Algorithm 4: QuadTree Decimation

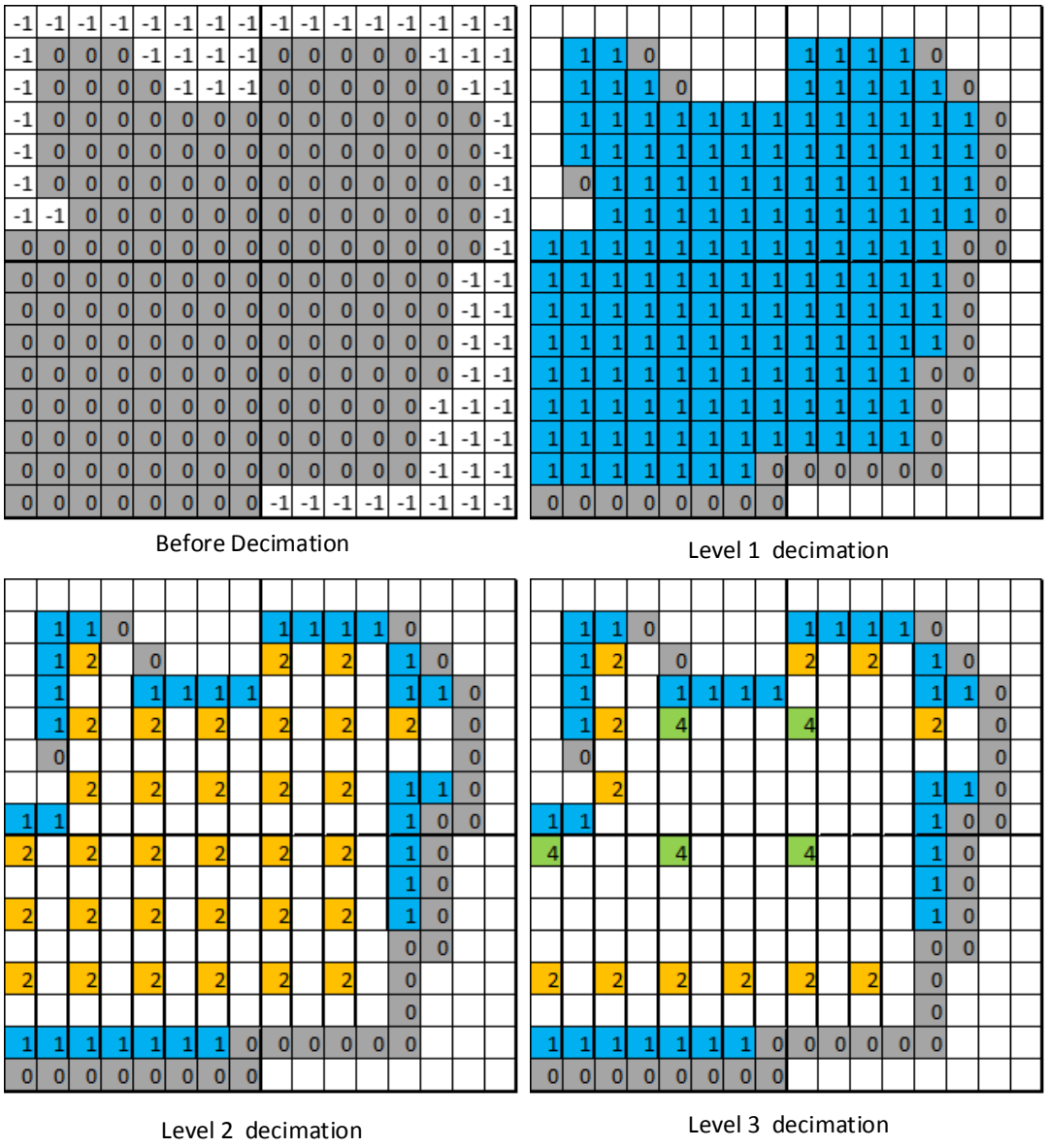

Figure 4.11: Example contents of the QuadTree degree buffer at various stages of decimation

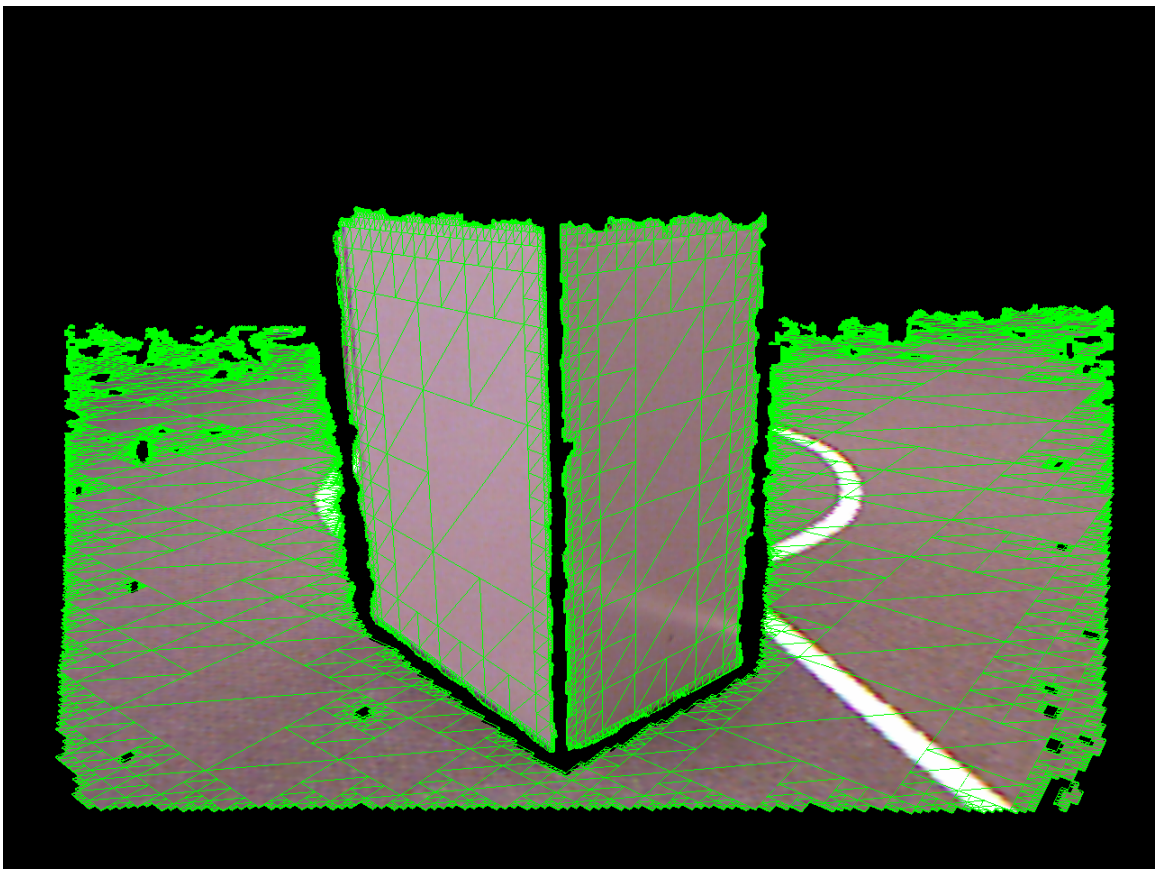

Figure 4.12: Wireframe visualization of the textured mesh. Rendered using the intrinsic parameters of the original camera for direct comparison. Example of output from mesh generation stage.

Chapter 5

Results and Analysis

This chapter provides some example results from the finished pipeline as well as some performance analysis. The runtime of some stages of the pipeline is heavily dependent on the target scene complexity. All measurements in this chapter were performed on a workstation with an Intel i5-3570K CPU @3.40GHz, 8.00 GB of RAM, Windows 8.1 64-bit, and a GeForce GTX 760 GPU.

5.1 Results

The pipeline is tuned to look for large planar surfaces like walls, furniture, ceilings, and floors. Figure 5.1 shows an example of a high quality segmentation result. The mesh reconstruction is visualized in the lower left corner through a virtual camera with the same intrinsic parameters as the original sensor for direct comparison. In this image, almost the entire scene is composed of planes, so the reconstruction is nearly complete.

Because the Kinect is based on structured infrared light, the result is robust to varying lighting conditions. Figure 5.2 shows a darker version of the same scene with the QuadTree mesh overlaid in green.

5.1.1 Successes

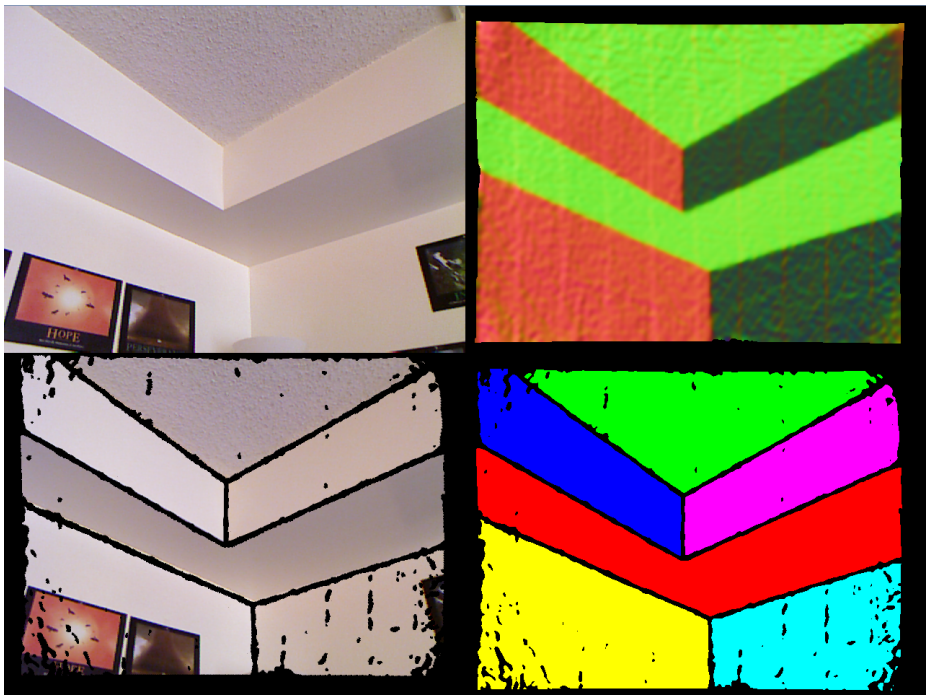

Figure 5.1: Example results of the upper corner of a room, 3 sets of 2 parallel planes detected. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

As it turns out, the pipeline is very good at floor plane detection. Figure 5.3 shows a reconstruction of an empty lab floor with a tape soccer field pattern. Because the algorithm is looking for global plane solutions, the result is very robust to occlusion (Figures 4.12 and 5.4). These images also demonstrate how adaptable the QuadTree representation is. Note that around the hole left by the robot in Figure 5.4 the QuadTree almost perfectly reconstructs the silhouette almost perfectly.

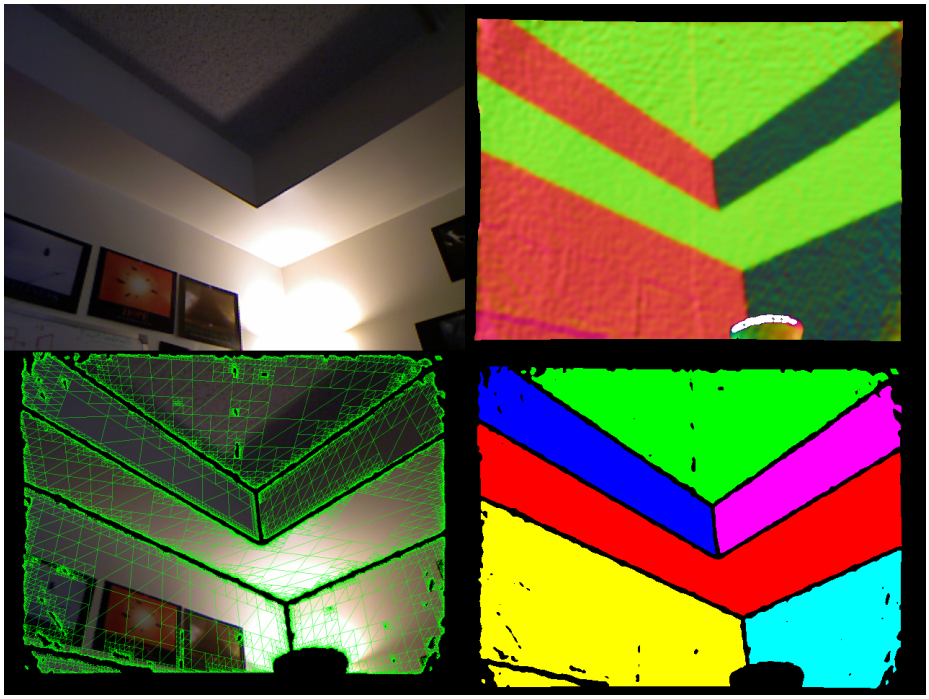

Figure 5.2: Example results of the upper corner of a room in low light conditions with wireframe. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

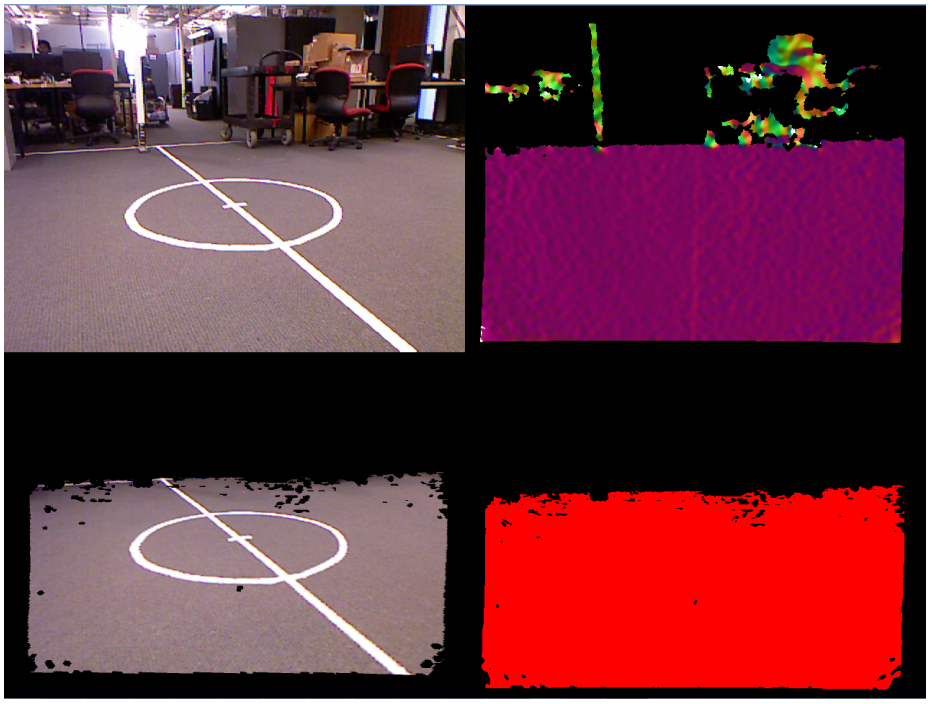

Figure 5.3: Effortless floor plane detection. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

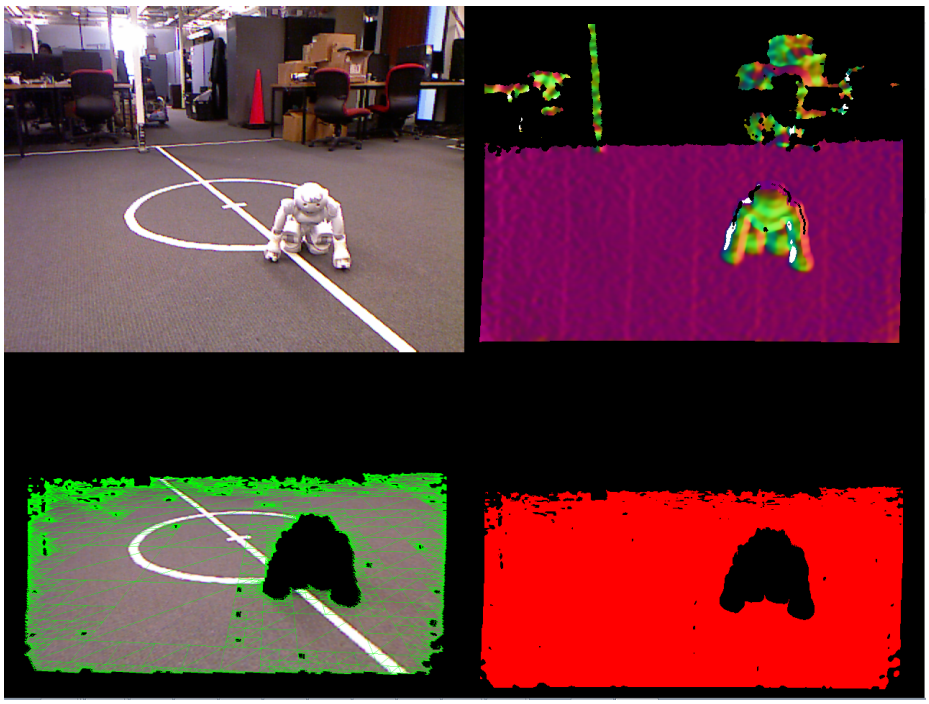

Figure 5.4: Lab floor with small obstruction to demonstrate QuadTree flexibility. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

5.1.2 Failures

However, the pipeline does have several issues and common failure cases. Figure 5.5 shows an example of pseudo-planar surfaces that are partially detected. Because the box sides are multifaceted, the algorithm detects a broken surface, if it detects the surface at all. A more serious issue is shallowly curved surfaces like lampshades or large building columns are sometimes detected as a series of small plane segments as in Figure 5.6. Not only does this represent a false positive on a clearly non-planar surfaces, but it also creates a set of false planes that can cause segmentation errors in other parts of the scene.

Figure 5.7 shows an object that is clearly a plane 2.5 meters from the sensor that is only partially detected due to noise and decreased resolution. This is likely because

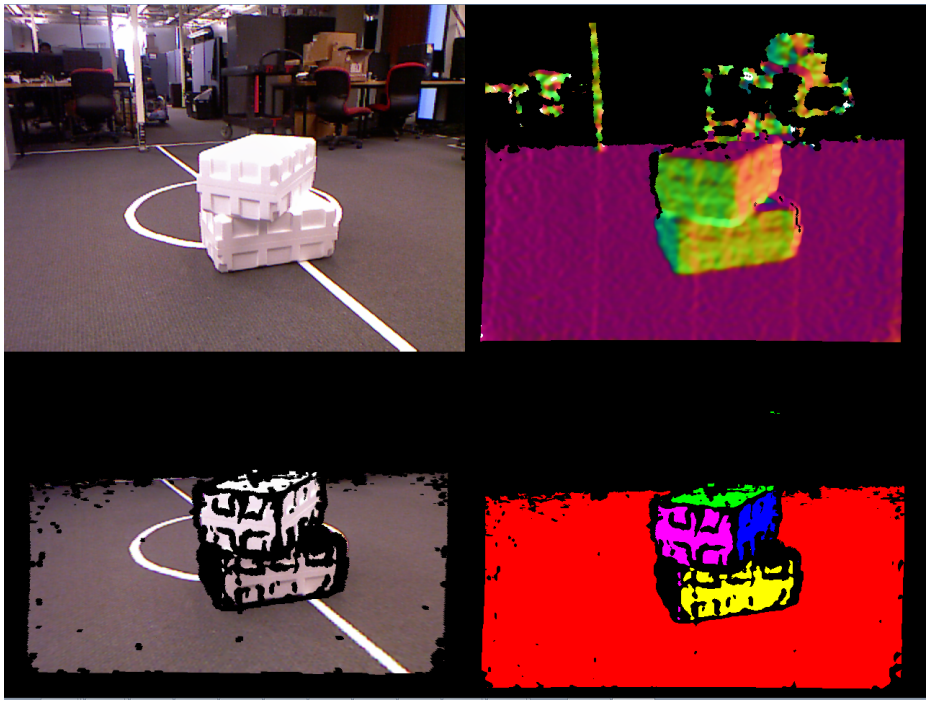

Figure 5.5: Algorithm has difficulty with pseudo-planar surfaces. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

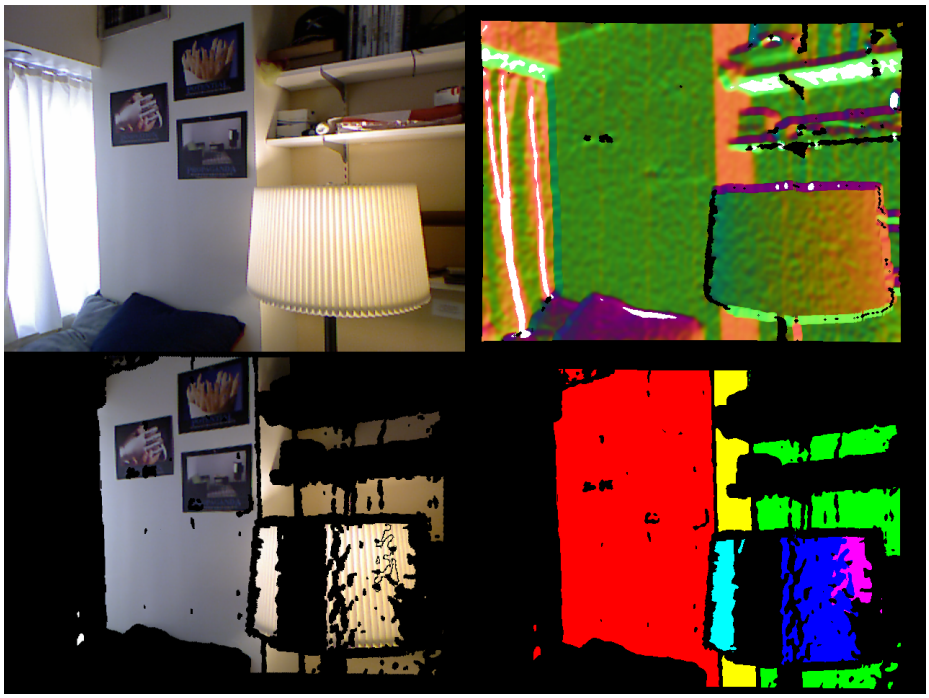

Figure 5.6: Algorithm incorrectly detects large smooth curves as multiple planes. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

the segmentation thresholds are constant across the full range of sensor accuracy and resolution, so surfaces at the extreme of the sensor's usable range are very likely to be broken up or poorly segmented. Adding resolution dependent thresholds is not a trivial task, because large planes can vary in resolution greatly from end to end.

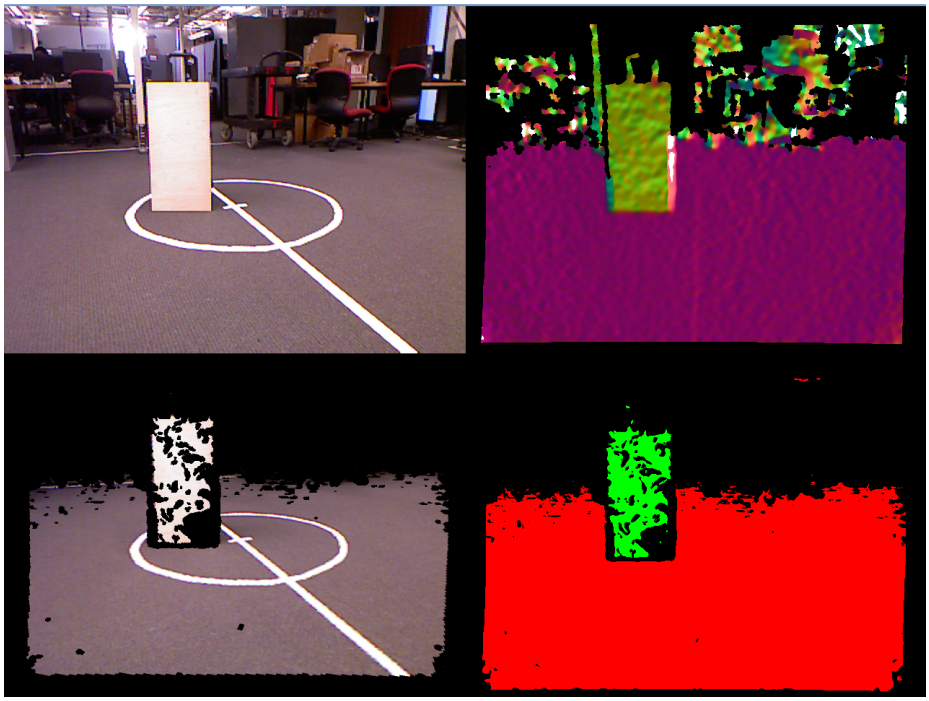

Figure 5.7: Fails to cleanly detect obvious plane 2.5m away. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

Figure 5.8 demonstrates a failing of the simple segmentation system I developed. If two large planes in the image (the wall and the whiteboard in this case) are very close to being parallel with each other, sometimes one of the planes will be completely missed. Also, when these closely related planes intersect, as is also the case in this image, an area tracing the plane intersection can be mislabeled. This is represented in Figure 5.8 by the horizontal strip of red spots extending from the upper right corner of the whiteboard segment extending towards the floorlamp. The plane intersection

problem also shows up even when both planes are correctly detected (Figure 5.9).

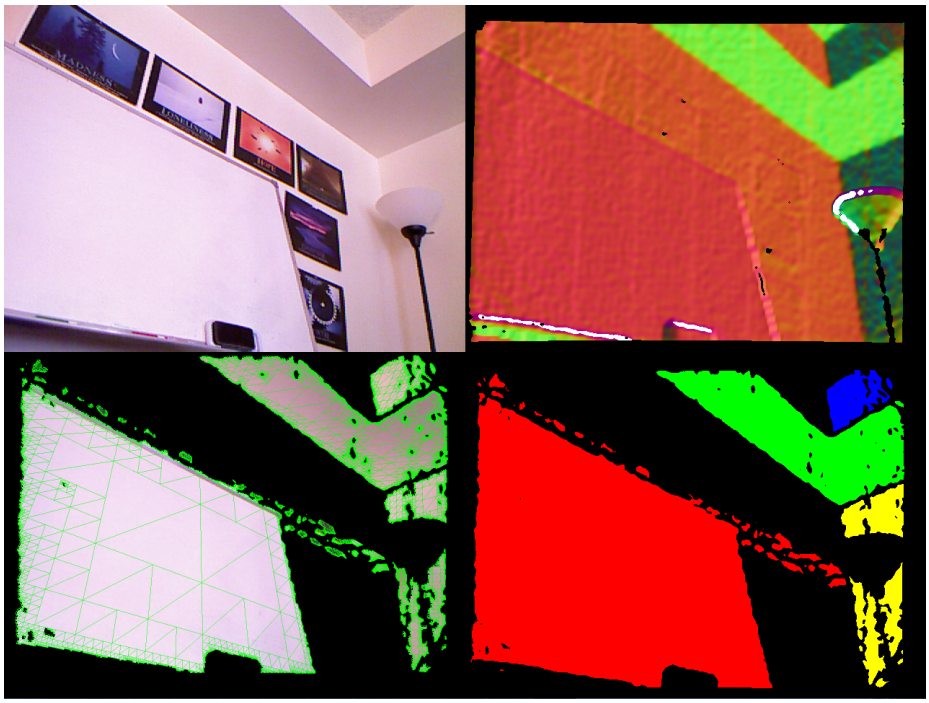

Figure 5.8: Two planes with similar normals, only one is detected. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

5.2 Performance Analysis

5.2.1 Runtime Analysis

One of the primary goals of this thesis is to ensure that the pipeline runs in near realtime. For a frame rate of roughly 30FPS, the entire pipeline must run in less than 32ms per frame. Most of the pipeline was designed to have roughly the same runtime regardless of the scene being imaged. However, some segments of the pipeline are heavily data dependent. In particular, the number of planes detected in the image has a huge impact on the runtime of the mesh generation module.

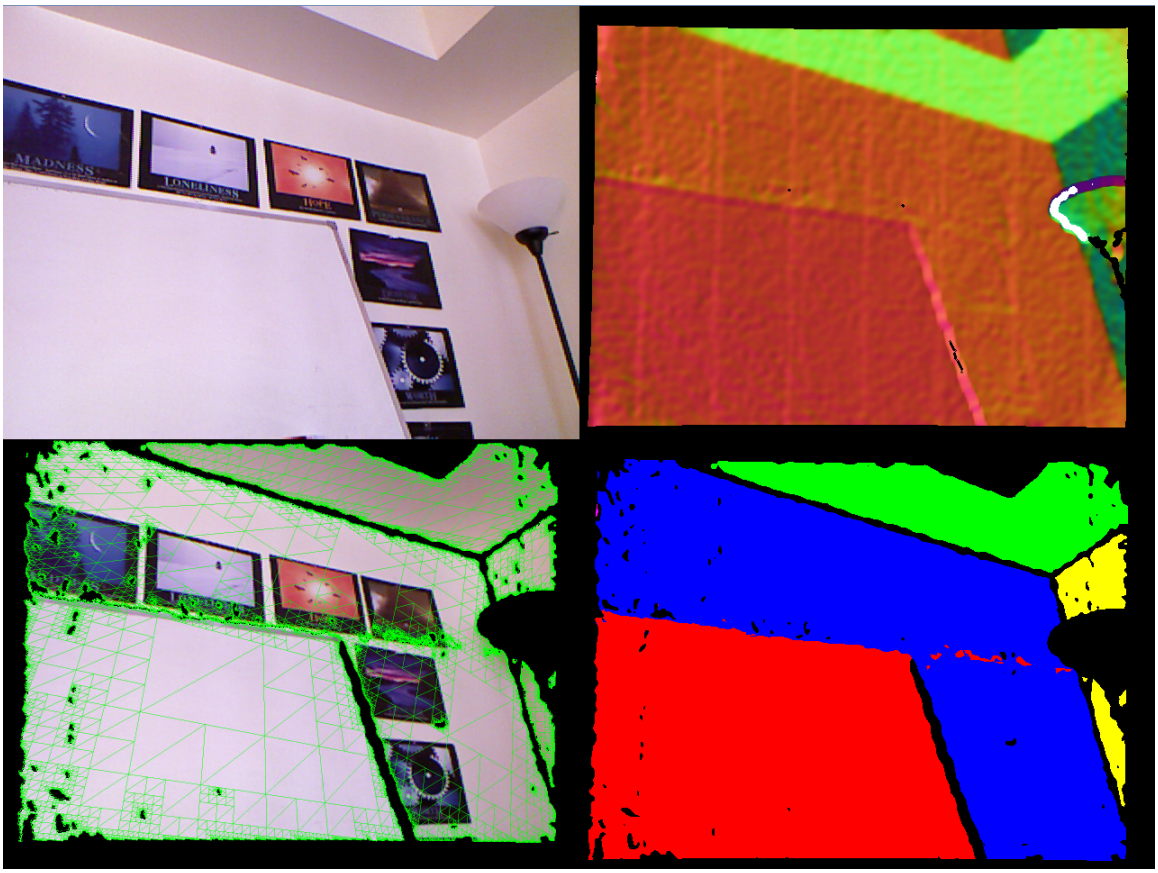

Figure 5.9: Two nearly parallel planes that intersect show some segmentation errors. Upper-left) original color image. Upper-right) surface normal estimates. Lower-left) mesh reconstruction. Lower-right) Random colorization of plane segments

Figure 5.10 shows the runtime of each major pipeline module outlined in Chapter 4 based data collected on a variety of scenes (n=2541). The memory management section has not been previously discussed and is responsible for deleting the results of the previous iteration.

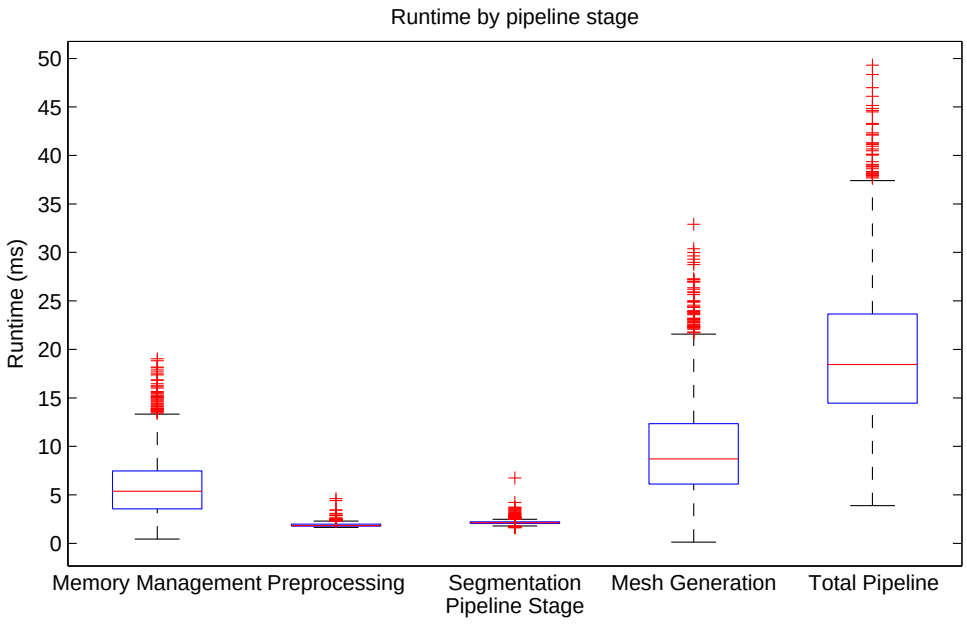

Figure 5.10: Pipeline runtime broken down by stage.

Note that both the preprocessing and segementation stages have very consistent runtimes with medians of 1.83 ms and 2.13 ms respectively. Figures 5.12 and 5.13 show that apart from the case where no planes were detected, the number of planes in the image had practically no effect on either stage's runtime. However, the mesh generation stage varies greatly in runtime. Figure 5.14 clearly demonstrates a positive correlation between the number of planes detected and the total runtime of the stage. Likewise, the memory management stage shows a slight increase in runtime as the number of planes increases.

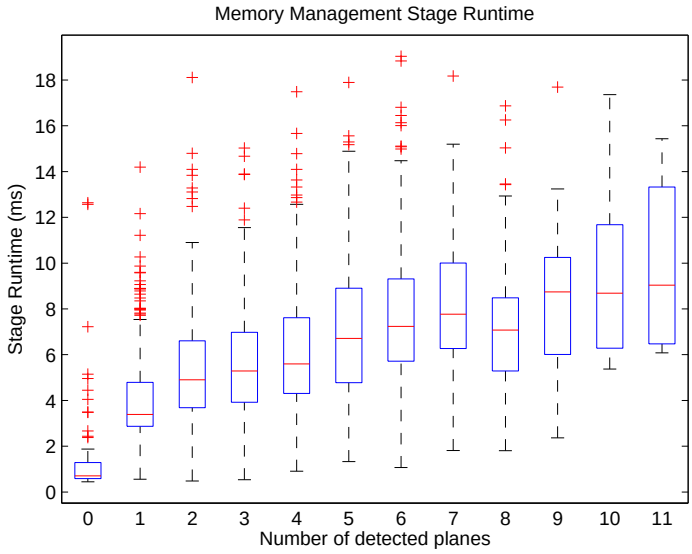

Figure 5.11: Runtime of the memory management stage by number of detected planes.

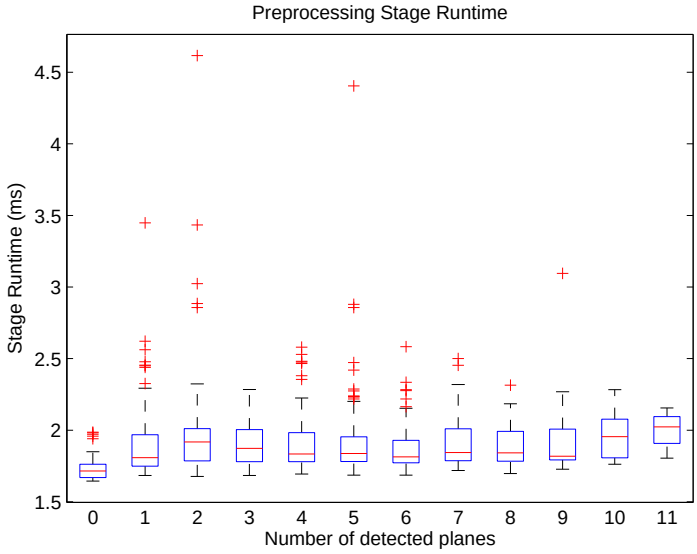

Figure 5.12: Runtime of the preprocessing stage by number of detected planes.

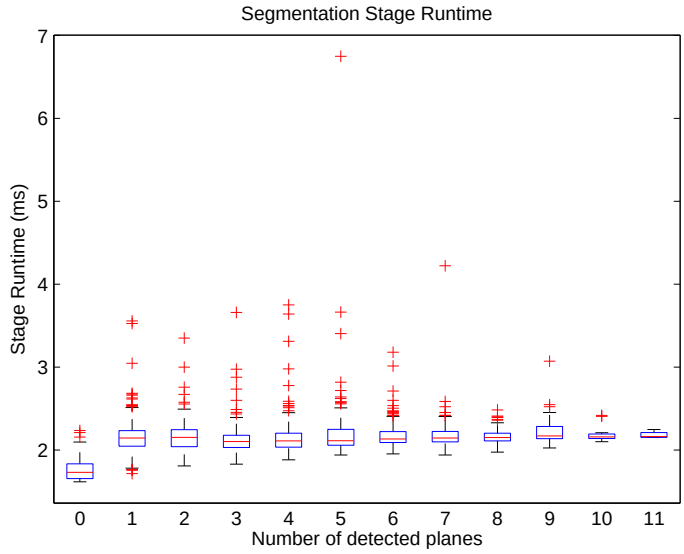

Figure 5.13: Runtime of the segmentation stage by number of detected planes.

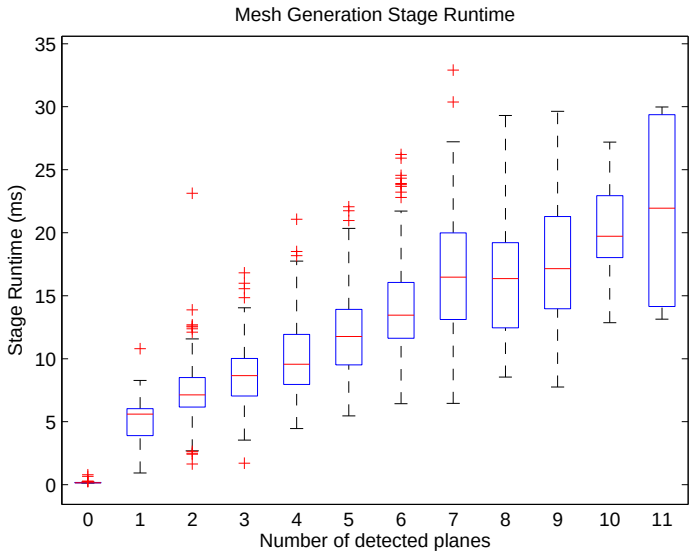

Figure 5.14: Runtime of the mesh generation stage by number of detected planes.

How has this affected the ability of the system to run in real-time? Figure 5.15 shows the total pipeline runtime for varying number of detected planes. The pipeline reliably achieves real-time performance when less than 6 planes are detected, and continues to achieve near real-time performance up to the most complicated scene with 11 planes.

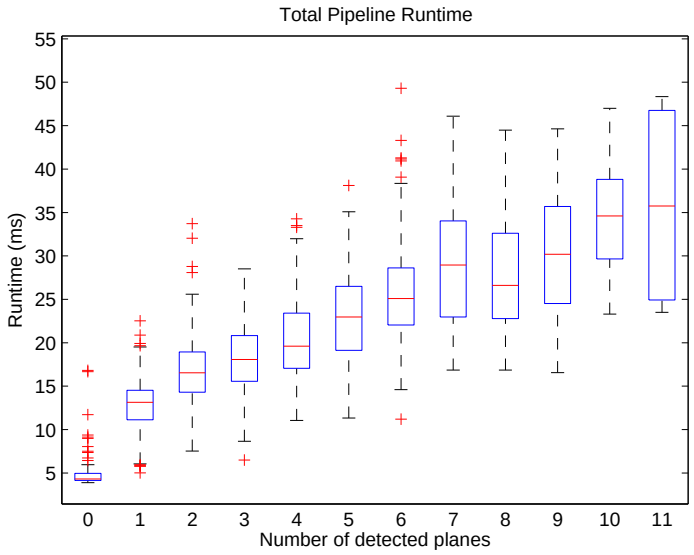

Figure 5.15: Runtime of the entire pipeline by number of detected planes.

5.2.2 Timeline Breakdown

The compute timeline of the pipeline is shown in Figure 5.16. This view of the pipeline helps identify some key areas of the pipeline that could benefit from optimization. The GPU is idle for the vast majority of the memory management section of the timeline, during which the CPU performs memory cleanup tasks. These tasks could be run in parallel with the next two pipeline stages, saving approximately 3ms per run.

The preprocessing timeline (Figure 5.17) shows a significant gap in the compute stream as a small chunk of data is copied to the device. That data is the precomputed spatial kernel for the bilateral filter (the pale blue boxes right after the gap). This copy was performed every time to make changing the filter parameters at runtime simpler. However, once those parameters are fixed, this gap could easily be engineered away.

The segmentation kernel has been highly optimized and offers few potential sources of performance improvement (Figure 5.18).

In contrast, the mesh generation pipeline (Figure 5.19) could potentially be improved in several ways. First, notice that the GPU alternates between compute and memory transfer operations. CUDA devices are capable of multi-streaming, where the results from a previous iteration of compute are transfered to the host while the next compute iteration begins to run. Implementing multi-streaming would make much better use of the GPU's available resources. Also, the pipeline does not need to be iterative at all. Every plane could be processed in parallel given enough available GPU memory. Finally, the primary impact on performance comes from the large memory transfers required to transfer the mesh textures to the CPU. Newer versions of CUDA than used by this thesis have the capability to write directly to GPU texture memory, which may eliminate this transfer completely.

5.3 Normal Estimation Accuracy

To assess the accuracy of the point normal estimation system, two different metrics were applied to the test set in Figure 5.20. First, the normals of the three planes

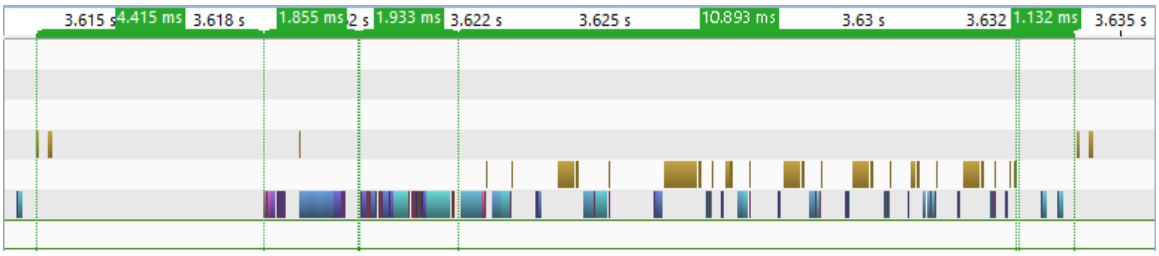

Figure 5.16: Timeline overview of pipeline captured by NVIDIA Visual Profiler. Each colored block indicates a different kernel. The MemCpy rows indicate memory transfers between the device and host. The time periods delineated in green indicate the stages of the pipeline. From left to right: Memory Management, Preprocessing, Segmentation, Mesh Generation, and OpenGL Visualization

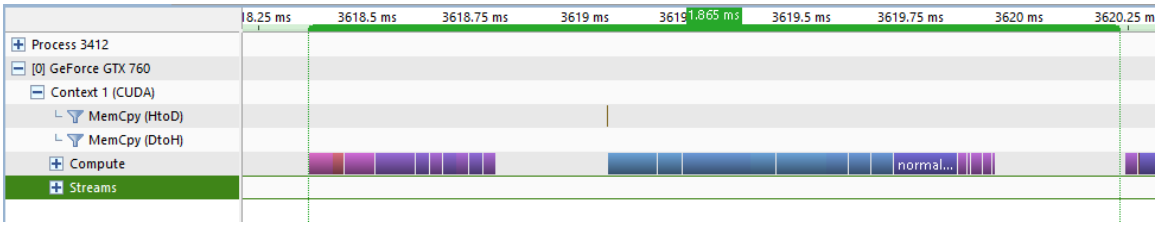

Figure 5.17: Timeline of the preprocessing stage of the pipeline captured by NVIDIA Visual Profiler. Each colored block indicates a different kernel. The MemCpy rows indicate memory transfers between the device and host.

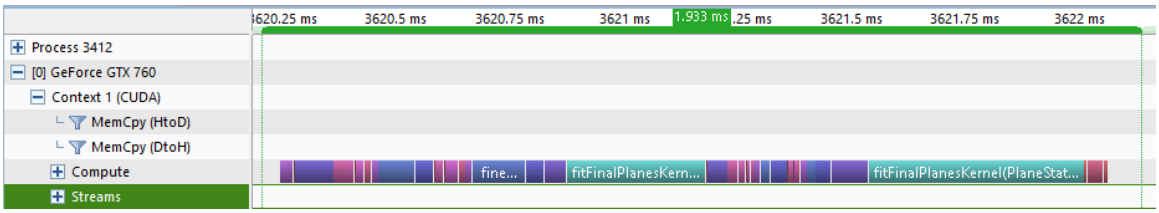

Figure 5.18: Timeline of the segmentation stage of the pipeline captured by NVIDIA Visual Profiler. Each colored block indicates a different kernel. The MemCpy rows indicate memory transfers between the device and host.

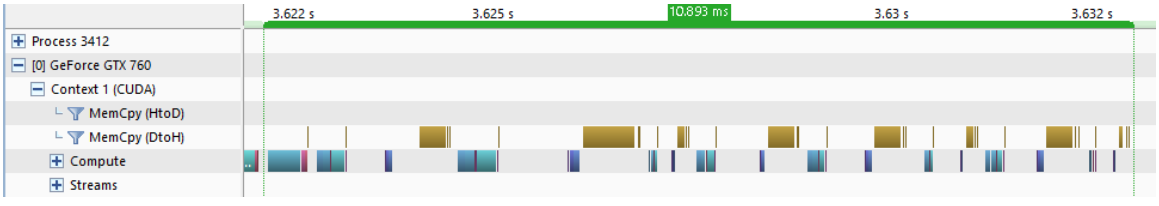

Figure 5.19: Timeline of the mesh generation stage of the pipeline captured by NVIDIA Visual Profiler. Each colored block indicates a different kernel. The MemCpy rows indicate memory transfers between the device and host.

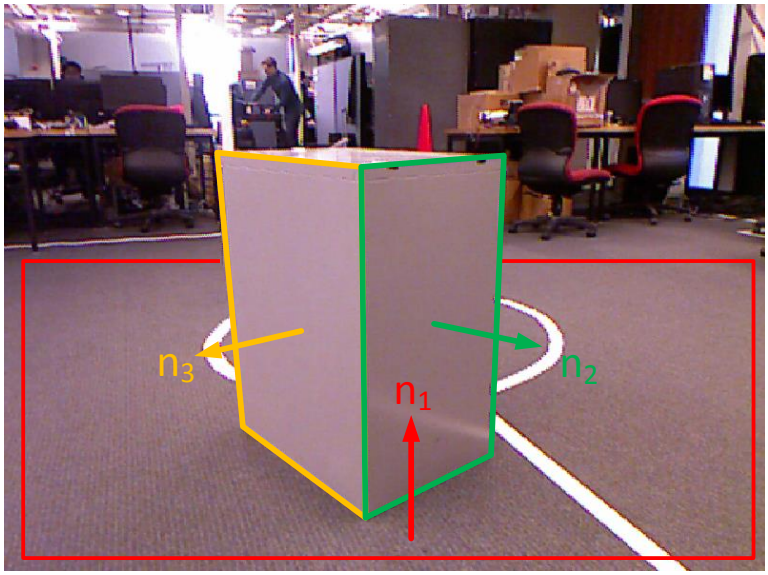

Figure 5.20: Normal accuracy analysis test set with true planes and normals drawn on top.

generated by the pipeline $(\vec{n_1}, \vec{n_2}, \vec{n_3})$ should should be perfectly orthogonal. By measuring the angle between the vectors pairwise, we have a good surrogate metric for the accuracy of the plane fits. Table 5.3 shows the relative error between each pair of vectors. The largest error is between $\vec{n_2}$ and $\vec{n_3}$, and even then it's only 1.443°.

Vector pair	Angle between vectors	Error from orthogonal
$\cos^{-1}(\vec{n_1} \cdot \vec{n_2})$	89.274°	0.726°
$\cos^{-1}(\vec{n_1} \cdot \vec{n_3})$	89.955°	0.045°
$\cos^{-1}(\vec{n_2} \cdot \vec{n_3})$	88.557°	1.443°

The second metric is to measure each segmented point's local normal with the final plane normal estimated by the pipeline. Figure 5.21 shows several heat maps of this error metric for the various depth filters and normal estimation methods. With the simple normal estimator and no depth filtering, the pipeline completely fails, so no data is available for that combination. Clearly the normal averaging has a significant effect on the normal error. Also, the largest errors tend to occur around sharp edges,

which is to be expected.

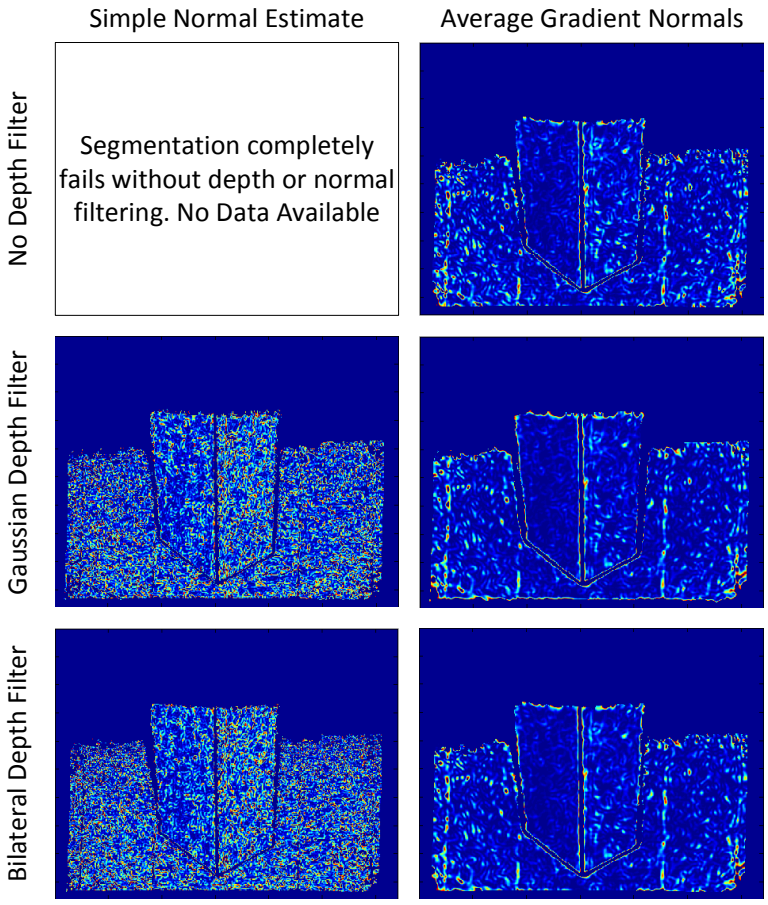

Figure 5.21: Heat map of local normal estimation errors compared to the found planes for various filters and estimation techniques. Values range from 0° off-angle (dark blue) to 2° off-angle (bright red). Unsegmented pixels appear as dark blue.

5.4 QuadTree Efficiency

The effectiveness of the QuadTree mesh simplification algorithm is difficult to measure fairly, since it was designed to optimize memory storage over multiple frames not frame by frame. So first, I will offer an apples to apples comparison between a set of meshes generated with and without QuadTree decimation on the same dataset. Fig-

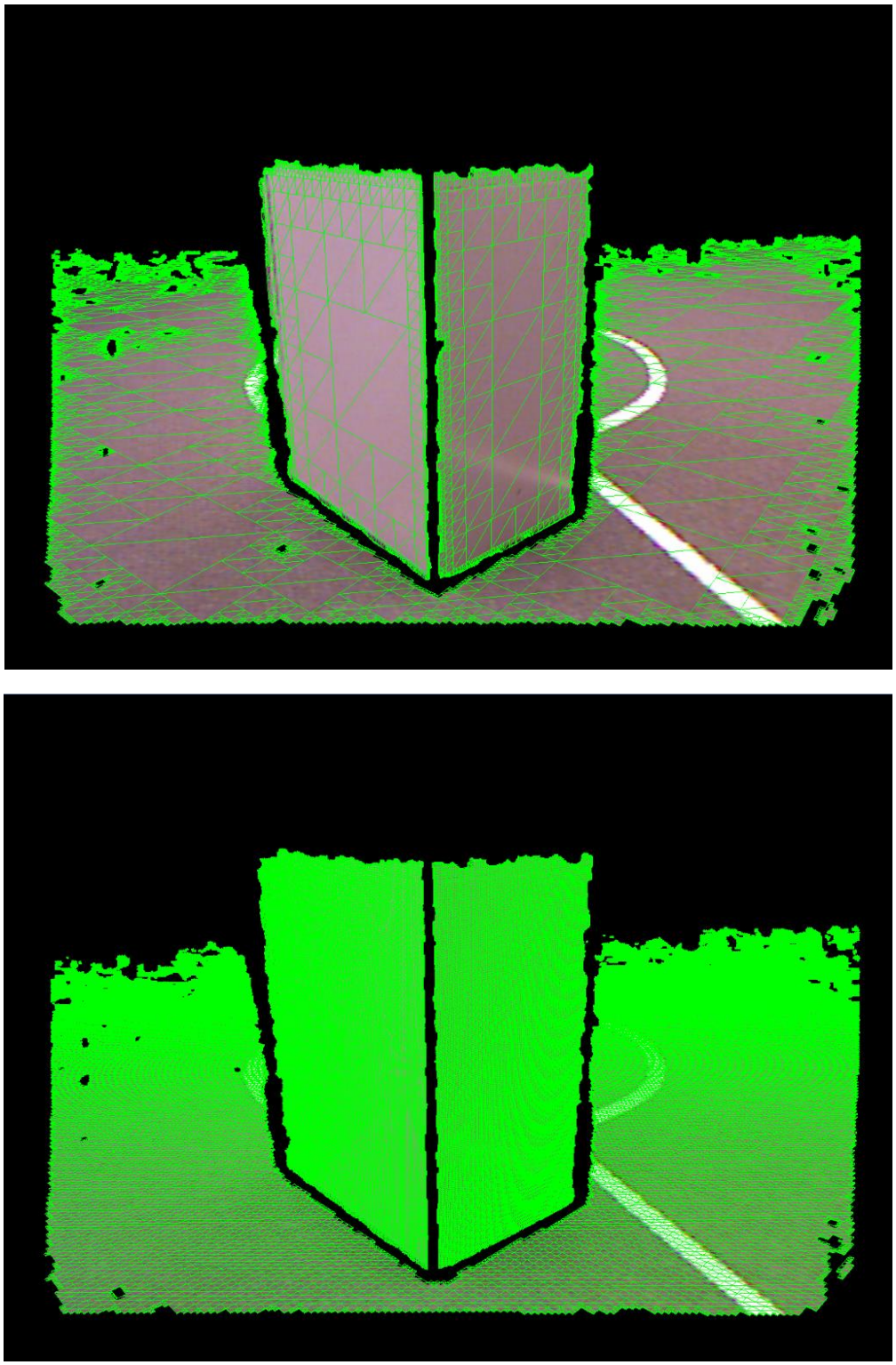

Figure 5.22: Comparison of textured mesh generated with (top, 76,698 vertices) and without (bottom, 18,573 vertices) QuadTree optimization.

ure 5.22 shows a visual comparison of the triangulations. The original mesh consists of 76,698 vertices. The simplified QuadTree mesh uses only 18,573 vertices, roughly 25% of the original mesh size.

However, the space these figures are computed in is a projection of the original image into a flat space. Depending on the size of the plane in the image, this space may be downscaled or upscaled to fit the texture efficiently. To show the practical effect this has on compression ratio, Figure 5.23 compares the measured compression ratio for multiple scenes plotted against the number of planar points in the original image. Note that only for some very small planes does the ratio go above 1, and for most they are at or below 0.25. This suggests that even if the mesh had been generated in camera space with a greedy meshing algorithm, the QuadTree representation would still be more space efficient in representing the geometry.

In addition, because the QuadTrees are restricted to a plane, each vertex only requires 2 degrees of freedom, whereas the greedy camera space triangulation requires 3 per vertex. Therefore, the storage requirements of the QuadTree are reduced by roughly another third over an equivalent 3D free mesh.

However, adding in the color texture representation results in a net gain in memory usage due to the higher resolution offered by the texture. Also, texture dimensions are fixed to be powers of 2, so many textures are larger than their contents. Figure 5.24 shows the theoretical memory usage for the same data set as Figure 5.23. Note the distinct curves that emerge from the data due to the rounding up of texture dimensions to powers of two.

So, the conclusion to draw from this data is that the memory savings from storing

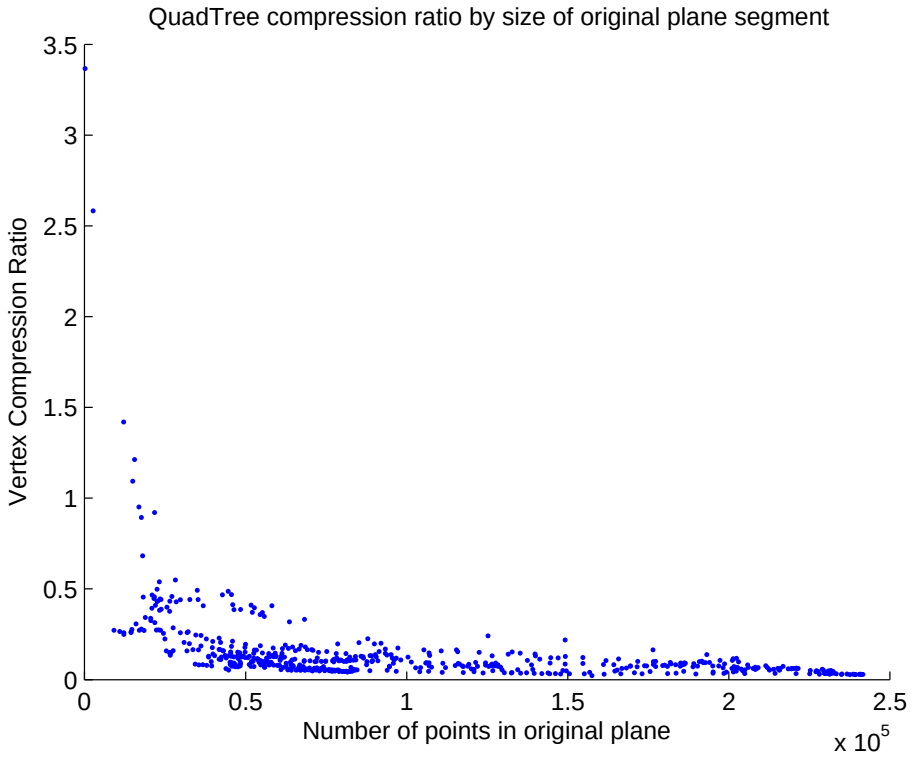

Figure 5.23: QuadTree vertex compression ratio measured for many different scenes. Compression is measured as number of pixels in the original plane segment over number of vertices in the final mesh.

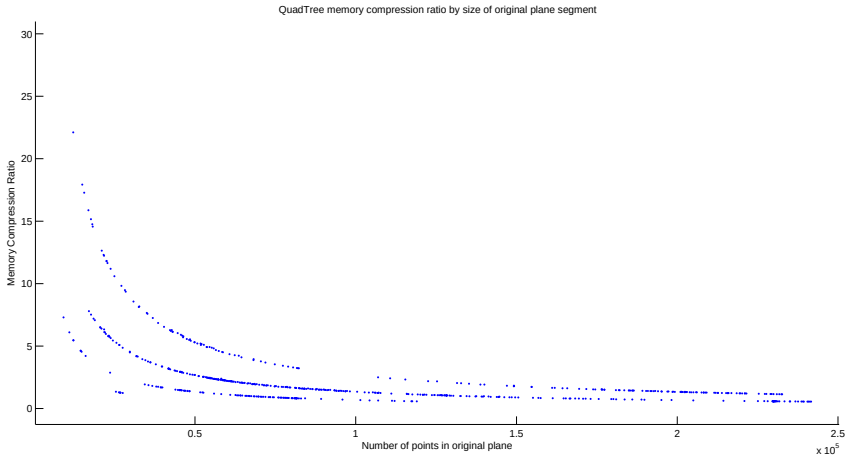

Figure 5.24: QuadTree total memory compression ratio measured for many different scenes. Compression is measured as number of bytes required by quadtree+color texture over number of bytes required by greedy triangulation with vertex coloring.

the mesh geometry in the QuadTree structure are quite significant, but at least for a single frame of data, including color information imposes a hefty additional memory burden. The hope is that as new frames are collected and integrated, this additional memory will be reused and the new color information will be trivial to integrate with the preexisting texture.

Chapter 6

Conclusions

This thesis describes a real-time plane segmentation and triangulation system based on RGB-D cameras and enabled by GPU parallel computing. Human environments tend to be dominated by planar surfaces, so it is quite useful to be capable of rapidly and reliably detecting them. By writing all of the image processing code in CUDA, the system is able to very quickly segment planar surfaces. The pipeline heavily leverages the algorithms optimization techniques outlined in Chapter 2 to maximize performance. The pipeline is fully capable of creating and rendering a triangular mesh-based reconstruction of planar surfaces in or near real-time with full color textures.

While the pipeline as a whole is just a component in a planned larger world model creation system outlined in Chapter 3, individual components created in this thesis could easily be leveraged for other applications. The segmentation module could easily be leveraged for other applications like floor plane detection, SLAM algorithms, and obstacle or object detection. Likewise the RGBDFramework library provides an

easily extensible and highly modular system that can make RGB-D technology easier to work with. The parallel mesh generation algorithm outlined in this thesis offers a very fast method for optimizing QuadTrees on the GPU. These components are all highly optimized, so they can be added to a new system with little impact on performance.

Future work will be focused on two areas. First, as discussed in Chapter 5, the mesh generation stage leaves a lot of room for re-engineering and optimization. Since this work targeted NVIDIA's Fermi architecture, many of the advances introduced by the newer Kepler GPU architecture such as dynamic parallelism could not be utilized. The pipeline could be improved in several key locations by utilizing Kepler's more advanced technologies.

The second and more significant research direction to pursue is completing the fully integrated mesh generation system this thesis was designed to enable. Being able to generate a mesh representation of the world in real-time with a single low cost sensor opens up many possibilities both inside and outside of robotics. Mesh representations combined with scene graph data structures offer methods to easily plan object manipulation tasks and enable robots to build robust but dynamic pictures of their environments. The ability to readily scan an environment into a manipulable 3D model in real-time could be extremely useful for rapidly prototyping interior designs, generating virtual tours or maps for buildings of interest, or low resolution 3D object scanning.

Full source code for this project can be found at: http://github.com/cboots/RGBD-to-Mesh/

Bibliography

[1] Alexey Abramov, Tomas Kulvicius, Florentin Wörgötter, and Babette Dellen. Real-time image segmentation on a gpu. In *Facing the multicore-challenge*, pages 131–142. Springer, 2011.

[2] Pilar Arques, F Aznar, M Pujol, and Ramón Rizo. Real time image segmentation using an adaptive thresholding approach. In *Current Topics in Artificial Intelligence*, pages 389–398. Springer, 2006.

[3] Joydeep Biswas and Manuela Veloso. Planar polygon extraction and merging from depth images. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 3859–3864. IEEE, 2012.

[4] Morten Bojsen-Hansen, Hao Li, and Chris Wojtan. Tracking surfaces with evolving topology. *ACM Trans. Graph.*, 31(4):53, 2012.

[5] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Parallel poisson surface reconstruction. In *Advances in Visual Computing*, pages 678–689. Springer, 2009.

[6] Abdul Dakkak and Ammar Husain. Recovering missing depth information from microsofts kinect, 2012.

[7] Felix Endres et al. An Evaluation of the RGB-D SLAM System. *Robotics and Automation (ICRA), 2012 IEEE International Conference on.*, 2012.

[8] Yanwen Guo, Hanqiu Sun, Qunsheng Peng, and Zhongding Jiang. Mesh-guided optimized retexturing for image and video. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):426–439, March 2008.

[9] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, pages 56–67, November 1986.

[10] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, CA 94720, June 1989.

[11] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Sven Behnke. Real-time plane segmentation using rgb-d cameras. In *RoboCup 2011: Robot Soccer World Cup XV*, pages 306–317. Springer, 2012.

[12] Stefan Holzer, Radu Bogdan Rusu, M Dixon, Suat Gedikli, and Nassir Navab. Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2684–2689. IEEE, 2012.

[13] Kai Hormann. From scattered samples to smooth surfaces. *Proc. of Geometric Modeling and Computer Graphics*, 2003.

[14] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM, 2011.

[15] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, 2006.

[16] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[17] S. Laine and T. Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, Aug 2011.

[18] Lingni Ma, R. Favier, Luat Do, E. Bondarev, and P.H.N. de With. Plane segmentation and decimation of point clouds for 3d environment reconstruction. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 43–49, Jan 2013.

[19] Lingni Ma, Thomas Whelan, Egor Bondarev, Peter HN de With, and John McDonald. Planar simplification and texturing of dense point cloud maps. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 164–171. IEEE, 2013.

[20] Z.C. Marton, R.B. Rusu, and M. Beetz. On fast surface reconstruction methods for large and noisy point clouds. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3218–3223, May 2009.

[21] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[22] Richard A Newcombe, Andrew J Davison, Shahram Izadi, Pushmeet Kohli, Otmar Hilliges, Jamie Shotton, David Molyneaux, Steve Hodges, David Kim, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[23] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[24] A. Nüchter, K. Lingemann, and J. Hertzberg. Cached k-d tree search for icp algorithms. In *Proceedings of the 6th IEEE international Conference on Recent Advances in 3D Digital Imaging and Modeling (3DIM '07)*, pages 419–426, August 2007.

[25] NVIDIA. Cuda toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[26] Masaaki Oka, Kyoya Tsutsui, Akio Ohba, Yoshitaka Kurauchi, and Takashi Tago. Real-time manipulation of texture-mapped surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 181–188, July 1987.

[27] Tuan Q Pham and Lucas J Van Vliet. Separable bilateral filtering for fast video preprocessing. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 4–pp. IEEE, 2005.

[28] Juan C Pichel, David E Singh, and Francisco F Rivera. A parallel framework for image segmentation using region based techniques.

[29] Michele Pirovano. Kinfu – an open source implementation of kinect fusion. http://homes.di.unimi.it/ pirovano/pdf/3d-scanning-pcl.pdf, 2012. PhD student in Computer Science at POLIMI.

[30] E. Puppo and D. Panozzo. Rgb subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):295–310, March 2009.

[31] Tahir Rabbani, Frank van den Heuvel, and G Vosselmann. Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(5):248–253, 2006.

[32] Julian Ryde and Jason J Corso. Fast voxel maps with counting bloom filters. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4413–4418. IEEE, 2012.

[33] Eftychios Sifakis, Tamar Shinar, Geoffrey Irving, and Ronald Fedkiw. Hybrid simulation of deformable solids. In *2007 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 81–90, August 2007.

[34] Luciano Spinello and Kai Oliver Arras. People detection in rgb-d data. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3838–3843. IEEE, 2011.

[35] F Steinbrucker, Jürgen Sturm, and Daniel Cremers. Real-time visual odometry from dense rgb-d images. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 719–722. IEEE, 2011.

[36] J Sturma, E Bylowb, C Kerla, F Kahlb, and D Cremersa. Dense tracking and mapping with a quadrocopter.

[37] Yuichi Taguchi, Yong-Dian Jian, Srikumar Ramalingam, and Chen Feng. Point-Plane SLAM for Hand-Held 3D Sensors. Technical report.

[38] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846. IEEE, 1998.

[39] Jeremiah van Oosten. Cuda memory model. http://3dgep.com/?p=2012, November 2011.

[40] Lifeng Wang, Sing Bing Kang, R. Szeliski, and Heung-Yeung Shum. Optimal texture map reconstruction from multiple views. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–347–I–354 vol.1, 2001.

[41] Jan Wassenberg, Wolfgang Middelmann, and Peter Sanders. An efficient parallel algorithm for graph-based image segmentation. In *Computer Analysis of Images and Patterns*, pages 1003–1010. Springer, 2009.

[42] Thomas Whelan, Hordur Johannsson, Michael Kaess, John J. Leonard, and John McDonald. Robust real-time visual odometry for dense rgb-d mapping. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5724–5731, May 2013.

[43] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John McDonald. Kintinuous: Spatially extended kinectfusion. 2012.

[44] K. M. Wurm, A. Hornung, M Bennewitz, C. Stachniss, and W. Burgard. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, USA, May 2010.

[45] Lu Xia, Chia-Chih Chen, and JK Aggarwal. Human detection using depth information by kinect. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 15–22. IEEE, 2011.