GPU-Accelerated Conversion of RGBD Images to Textured Triangle Meshes

Collin Boots
University of Pennsylvania

Dalton Banks
University of Pennsylvania

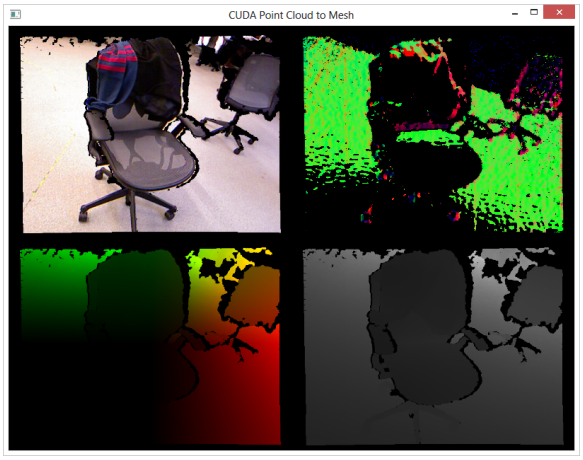

Figure 1. Depth data (placeholder)

Abstract

Mesh reconstruction from low-cost RGBD cameras, as opposed to simple point cloud storage, holds many possibilities, such as ease of object recognition and greatly reduced memory requirements. A modular framework was developed for extracting and synchronizing RGB and depth data from an RGBD camera, performing CUDA-based data processing on a GPU, and rendering the transformed data with OpenGL via CUDA-OpenGL interop on the GPU. In particular, data was gathered from the Microsoft Kinect using OpenNI, CUDA kernels were used to perform neighborhood-based estimation of point normals and dense surface mesh reconstruction in camera-space, and a variety of GLSL shaders were implemented to visualize the processed data. Further work will extend the framework to perform image registration of successive RBGD frames for reconstruction from camera movement, and will perform adaptive mesh resampling in order to further decrease memory usage.

1. Introduction

Previous work has demonstrated the diverse capabilities of RGBD cameras, from generating highly accurate 3D surface models [Newcombe et al. 2011] to reliably estimating 3D pose [Endres et al. 2012; Taguchi et al.]. However, many algorithms attempt to store the generated environment as a RGB 3D point cloud, which is not easily adaptable to dynamic environments, requires enormous quantities of memory to store large environments, and provides no intuition to higher perception processes about distinct objects beyond a volumetric approximation. Other approaches have been able to store and merge the surface data more efficiently, but still regard the environment as a unified whole rather than discrete objects. By instead extracting meaningful geometry from the RGBD data in the form of triangle meshes, many advantages can be realized:

1. High storage efficiency
2. Natural low level object segmentation
3. Easy to manipulate, modify, and render in real time
4. Efficient and easy to process intuition of geometry that higher cognitive functions can use for object recognition and manipulation tasks.
5. Straightforward tradeoff between simplicity and accuracy with mesh resolution

2. Code Overview

2.1. Image processing pipeline

The overall image processing pipeline is shown in Figure 3. First, an RGB frame and a depth frame are pulled from the Kinect and shipped to the GPU for processing. A world-space point cloud is then generated from the RGBD data, and a neighborhood-based estimate of the point normals is then extracted for later processing. Finally, the point cloud is triangulated and the generated mesh is passed to OpenGL where a variety of rendering options are implemented.

The underlying architecture is very modular, and can be easily extended to handle input RGBD streams other than the Kinect (as demonstrated in the implementation of log streams). A generic RGBD frame format is used, allowing computation and visualization to be performed without regard to how the data was obtained.

A more detailed view of the program flow is shown in Figure **??**. Note that after the RGB and depth frames are synchronized and shipped to the GPU, all computation and rendering is performed on the GPU, enhancing performance and allowing the CPU to be free for other tasks. The ComputeNormalsFast kernel supplants an earlier

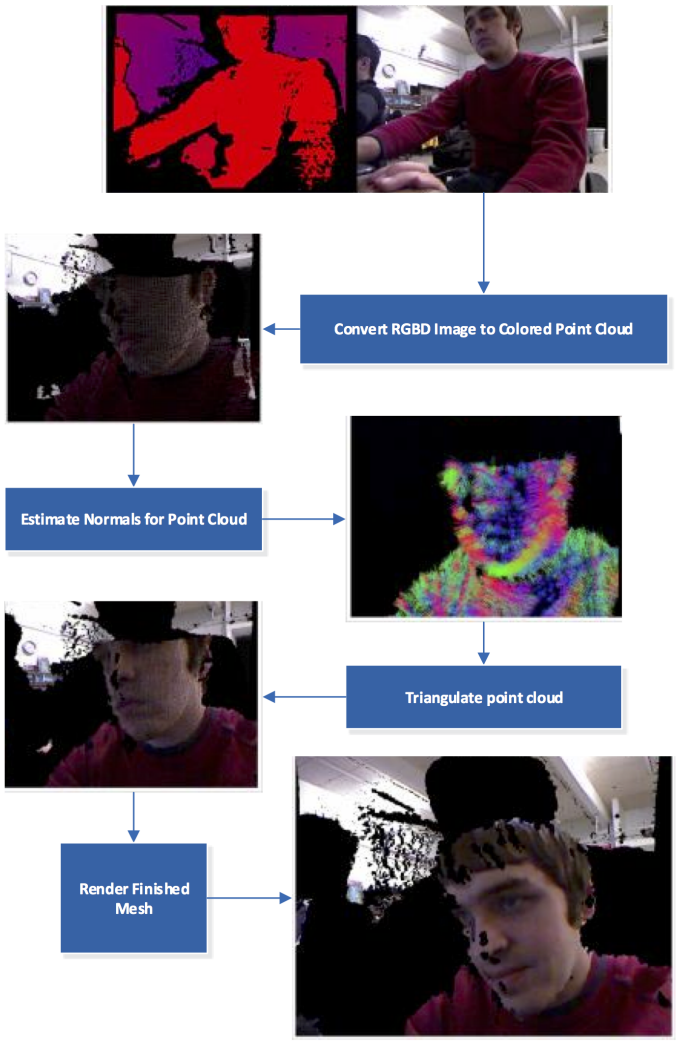

Figure 2. Image Processing Pipeline

iteration, ComputeNormals, which was written for estimation quality at the cost of a significant performance penalty.

Finally, Figure 6 shows a more detailed view of the OpenGL rendering pipeline. The rendering pipeline is also written in a very modular manner, allowing both for rapid code modification to experiment with different visualazation techniques, as well as hooks (note the black diamonds) for keypresses to completely change the render output on-the-fly.

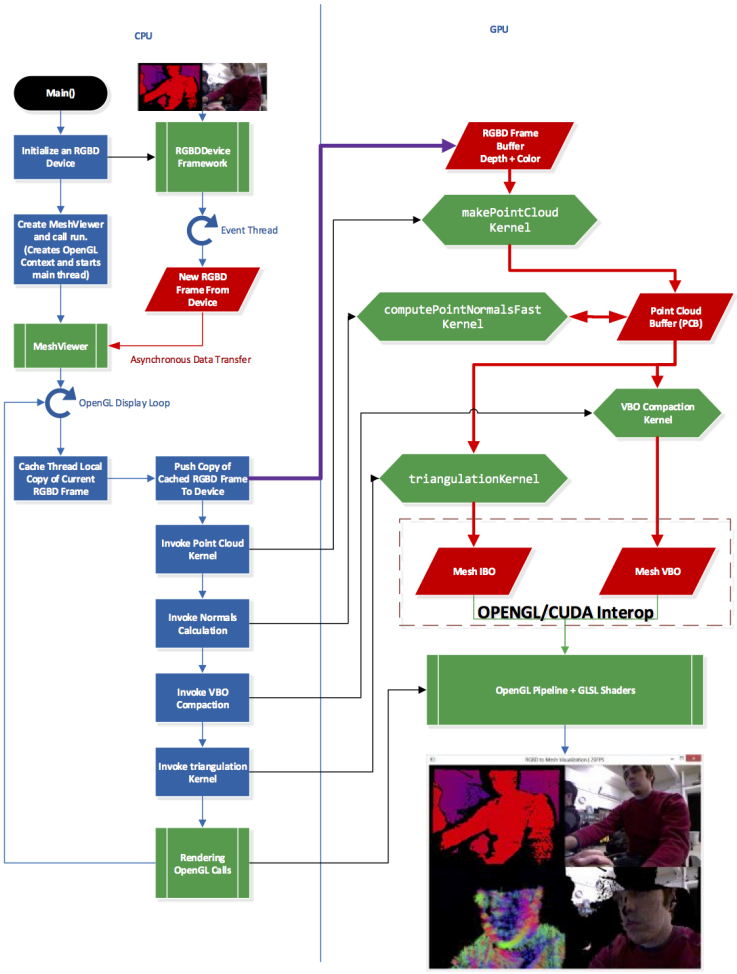

Figure 3. Overall Program Flow

3. Performance Analysis

Our point normals kernel was implemented as follows. A window radius is first specified as an algorithm parameter. For each point, we loop through its neighboring points in screen space in the square window specified by the radius, and pair it with a screen-space orthogonal point at the same radius. If both points are within a specified radius from the center point in world space, we take the cross product to compute the normal, which is then flipped if pointing away from the camera. If sufficiently many valid normals are found, we average them to produce the final normal estimate, otherwise we discard the point.

To improve the runtime of the point normals kernel, we reimplemented the algorithm using shared memory. In the shared memory implementation, all points in given

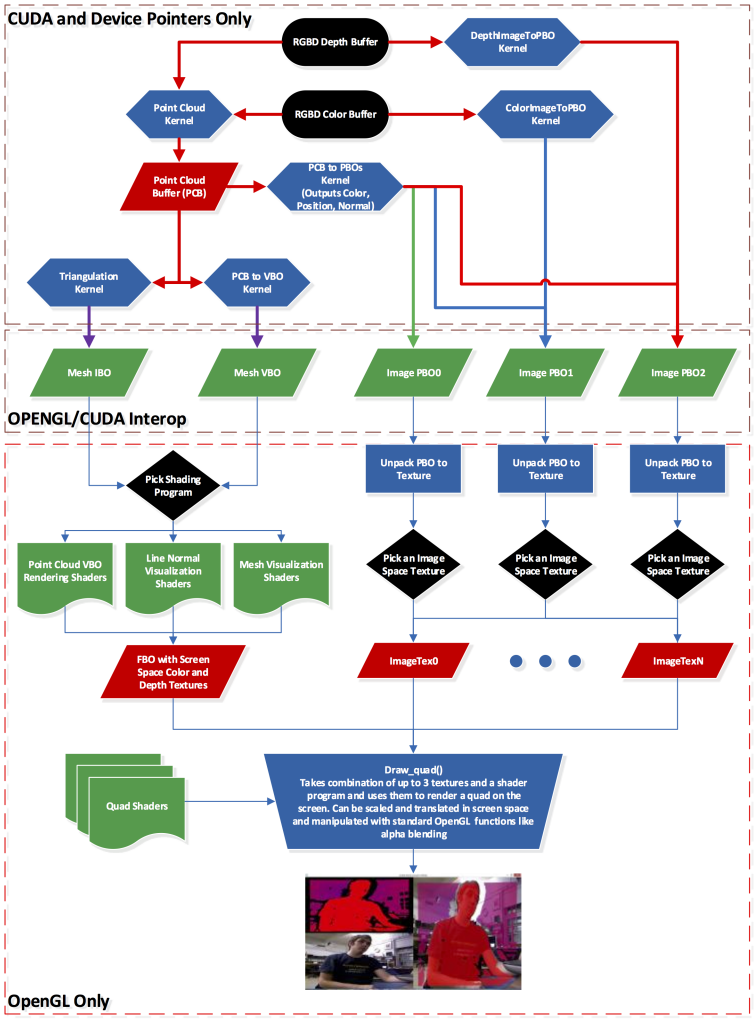

Figure 4. OpenGL Pipeline

thread block are first loaded into shared memory, along with the points lying within the specified neighborhood radius of the edges of the thread block, and the distance and cross product calculations are then performed using shared memory access. The results of the shared memory optimization on kernel runtime are shown for a range of window radii using a thread block size of 8x8.

As demonstrated, the shared memory optimization reduced the kernel runtime by approximately a factor of 2. The impact on the overall FPS was less dramatic, though still pronounced, due to the time spent in the rendering pipeline. All testing for this project was conducted on an Intel Core i5-2450M CPU, 2.5GHz 6GB (Windows 8, 64-bit OS) and an NVIDIA GeForce GT 525M GPU.

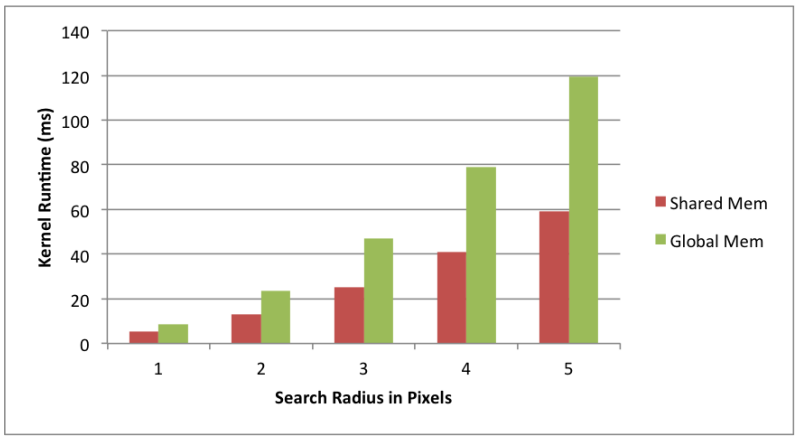

Figure 5. Global vs. shared memory access: kernel runtime

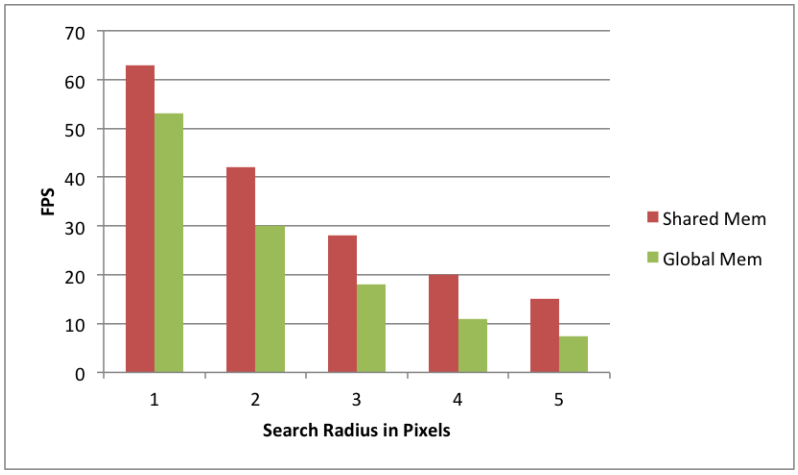

Figure 6. Global vs. shared memory access: frame rate

References

ENDRES, F., ET AL. 2012. An Evaluation of the RGB-D SLAM System. *Robotics and Automation (ICRA), 2012 IEEE International Conference on.*. 2

NEWCOMBE, R. A., ET AL. 2011. KinectFusion: Real-time dense surface mapping and tracking. 2

TAGUCHI, Y., JIAN, Y.-D., RAMALINGAM, S., AND FENG, C. Point-Plane SLAM for Hand-Held 3D Sensors. Tech. rep. 2