Real-Time Scene Reconstruction

Remington Gong
Benjamin Harris
Iuri Prilepov

June 10, 2010
Abstract

This report discusses the implementation of a real-time system for scene reconstruction. Algorithms for feature extraction, tracking, structure from motion and mesh generation are introduced. We present experimental results demonstrating the performance of the KLT feature tracker running on GPU. The report touches on the eight point algorithm for structure from motion and a solution to mesh generation, analyzing their fitness for a real-time system.
Contents

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
2 KLT Feature Tracking on GPU . . . . . . . . . . . . . . . . 3
   2.1 Background . . . . . . . . . . . . . . . . . . . . . . . 3
   2.2 Implementation . . . . . . . . . . . . . . . . . . . . 4
   2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . 4
3 Eight point algorithm . . . . . . . . . . . . . . . . . . . . . 6
4 Mesh Generation . . . . . . . . . . . . . . . . . . . . . . . 8
5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . 9

Bibliography 10
1 Introduction

Our team took on the challenge to create an application for capturing and recreating environments from photographs. Though the challenge was to reconstruct objects from photographs, we opted to use video sequences instead, and attempted to build a system to recreate 3D model of an environment in real-time. The problem can be broken up into three parts. The first step is to detect feature points in a frame and then track them in consecutive frames. Then numeric linear algebra techniques are used to infer the depth of the points in each camera view and generate a point cloud of a 3D model. Lastly, based on the point cloud an attempt is made to generate the best fitting surface. This is a long-term project that hasn’t been completed. Currently, only the first step - feature tracking - is fully functional and exhibits real-time performance.

The first chapter discusses the feature tracking on GPU, second chapter will briefly discuss the eight point algorithm used to solve for the point cloud and our current state of the implementation. The last chapter describes what mesh generation is and how we would go about implementing it.
2 KLT Feature Tracking on GPU

Feature tracking is the first major stage of 3D scene reconstruction. Given a sequence of 2D images, a feature tracker identifies 2D points in each pair of adjacent frames that correspond to the same real-world features. The features can have any type of geometry but in the application of scene reconstruction points are convenient so that the output of the tracker can be used to generate a 3D point cloud using structure-from-motion algorithms.

2.1 Background

There are many ways to go about tracking features across image frames, but the most common method for real-time applications is the KLT feature tracking algorithm. This algorithm, developed by Kanade, Lucas, and Tomasi in [4, 7], uses a simple motion model for an image sequence: it assumes the camera moves approximately linearly between adjacent frames. This assumption greatly simplifies the mathematical representation of the problem.

We model the input as a sequence of functions $I_1(\vec{x}), I_2(\vec{x}), \ldots, I_F(\vec{x})$ that relate the color intensities of each of $F$ frames to 2D pixel coordinates $\vec{x} = (x,y)$. The following function, defined with a slightly different formulation in [7], represents the constraint that the KLT tracking algorithm imposes on frames $1 \leq i < F$:

$$I_i(\vec{x}) = I_{i+1}(\vec{x} + \vec{\delta}_i(\vec{x})) + noise_i(\vec{x})$$

The function $\vec{\delta}_i(\vec{x})$ relates pixel coordinates in each frame to the linear displacement to the subsequent frame. The function $noise_i(\vec{x})$ accounts for arbitrary noise in the intensity measurements of the camera. As Tomasi and Kanade point out in [7], if this factor is too large then the above constraint becomes extremely difficult to solve for, and the KLT algorithm will likely fail.

The constraints now defined reduce the tracking problem to two simple steps. The first step selects $N$ points from the first image, say $\vec{x}_{1,1}, \vec{x}_{1,2}, \ldots, \vec{x}_{1,N}$, that are suitable for tracking. In [7], Tomasi and Kanade show that points with high intensity changes in both $x$ and $y$ directions (i.e. “corner points”) are the best features to select since the intensity change is used in the second step as the basis for tracking. To select these “good features,” gradients of the image are calculated in both $x$ and $y$ directions and the minimum eigenvalue of the $2 \times 2$ gradient matrix is used to rate each pixel’s trackability. The $N$ pixels with the highest trackability are chosen to be tracked.

The second step solves $\vec{\delta}_i(\vec{x})$ for each point in frame $i$, where $1 \leq i < F$. The location of each point in a subsequent frame can then be easily calculated as $\vec{x}_{i+1,j} = \vec{x}_i + \vec{\delta}_i(\vec{x}_{i,j})$, where $1 \leq j \leq N$. The common approach to solve for the delta function uses the Newton-Rhapson iterative method to
minimize the error in the intensity difference between windows of pixels from each frame. If the linear motion model from the KLT assumption holds, this method quickly converges to the correct delta function.

Putting these two steps together, it is possible to track \( N \) feature points across \( F \) image frames with only incremental computation. This property bodes well for real-time applications, which is likely why KLT tracking is the most common method of choice.

### 2.2 Implementation

Considering the massive parallelism in the above algorithm, we decided to develop a GPU implementation. Our first implementation attempt used the OpenCL API, but we later moved to NVIDIA’s CUDA API. The implementations involved several components, most of which mapped well to the GPGPU architecture:

1. [GPU] Convolution filter to smooth images and calculate gradients.
3. [CPU] Feature selector to find the best points to track.
4. [GPU] Feature tracker to solve for the displacement of feature points.

Performance in the OpenCL version was limited by the convolution component since we did not consider GPU-specific optimizations. In the CUDA version, we used several optimizations such as loop unrolling, memory access coalescing, and caching in shared memory. These lower-level optimizations, more accessible in the CUDA API, offered significant performance gains. Another performance bottleneck was the copying of eigenvalues to the CPU for component 3, the sorting of eigenvalues on the CPU, and the copying of feature points back to the GPU for component 4. Our final solution uses a selection-heap-sort technique to only sort and copy data that is needed since most the pixels of the input image are thrown out before the tracking stage.

We also considered several quality enhancements for the KLT algorithm. We calculated a pyramid of images from each input image, such that each level of the pyramid is one fourth the size of the previous level. With sufficient smoothing and subsampling these smaller pyramid levels provide the tracking component with initial estimates of the displacement function, allowing larger displacements to converge more quickly and avoiding convergence on local error minima.

### 2.3 Results

Our resulting feature tracker attains reasonable real-time performance. On an NVIDIA 8600 GT laptop GPU, the tracker takes an average of 20ms
per frame to track 100 features in a 512 by 512 video sequence. With more powerful desktop cards we noticed as much as 10 times this performance, clearly allowing the possibility for more features and larger video resolutions.

Figure 1: NVIDIA 8600M GT

<table>
<thead>
<tr>
<th>Total(ms)</th>
<th>Ave(ms)</th>
<th>Ave(%) : Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>25507</td>
<td>23.5</td>
<td>1.000: Frame</td>
</tr>
<tr>
<td>4789</td>
<td>4.4</td>
<td>0.188 : Frame Grabbing</td>
</tr>
<tr>
<td>10561</td>
<td>10.4</td>
<td>0.414 : Feature Tracking</td>
</tr>
<tr>
<td>10549</td>
<td>10.4</td>
<td>0.999 : Preprocessing</td>
</tr>
<tr>
<td>1657</td>
<td>1.6</td>
<td>0.157 : Copy to GPU</td>
</tr>
<tr>
<td>819</td>
<td>0.8</td>
<td>0.078 : Intensity Normalization</td>
</tr>
<tr>
<td>1610</td>
<td>1.6</td>
<td>0.153 : Smoothing</td>
</tr>
<tr>
<td>6426</td>
<td>6.3</td>
<td>0.609 : Pyramid</td>
</tr>
<tr>
<td>2001</td>
<td>2.0</td>
<td>0.311 : Downsampling</td>
</tr>
<tr>
<td>10143</td>
<td>9.4</td>
<td>0.398 : Rendering</td>
</tr>
</tbody>
</table>

Figure 2: NVIDIA GTS 250

<table>
<thead>
<tr>
<th>Total(ms)</th>
<th>Ave(ms)</th>
<th>Ave(%) : Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2843</td>
<td>2.3</td>
<td>1.000: Frame</td>
</tr>
<tr>
<td>1454</td>
<td>1.2</td>
<td>0.511 : Frame Grabbing</td>
</tr>
<tr>
<td>1384</td>
<td>1.1</td>
<td>0.487 : Feature Tracking</td>
</tr>
<tr>
<td>1382</td>
<td>1.1</td>
<td>0.999 : Preprocessing</td>
</tr>
<tr>
<td>427</td>
<td>0.4</td>
<td>0.309 : Copy to GPU</td>
</tr>
<tr>
<td>133</td>
<td>0.1</td>
<td>0.096 : Intensity Normalization</td>
</tr>
<tr>
<td>181</td>
<td>0.1</td>
<td>0.131 : Smoothing</td>
</tr>
<tr>
<td>637</td>
<td>0.5</td>
<td>0.461 : Pyramid</td>
</tr>
<tr>
<td>189</td>
<td>0.2</td>
<td>0.297 : Downsampling</td>
</tr>
</tbody>
</table>
3 Eight point algorithm

The eight-point algorithm (EPA) is a method for recovering a 3D point cloud from two views [1]. It is based on estimating the rotation and translate of the camera between the two views from the displacement of corresponding feature points in two frames. The notion of the fundamental matrix $F$ is central to the EPA. For every pair of points the following equation must hold

$$p'^T F p = 0 \quad (1)$$

where $(p,p')$ is a pair of corresponding feature points from two views

$$p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad p' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$  

After rearranging the terms the equation (1) takes the form $AF = 0$, with

$$A = \begin{bmatrix} x_1x'_1 & x_1y'_1 & x_1 & y_1x'_1 & y_1y'_1 & y_1 & x'_1 & y'_1 & 1 \\ x_2x'_2 & x_2y'_2 & x_2 & y_2x'_2 & y_2y'_2 & y_2 & x'_2 & y'_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}.$$  

A least squares approximation of $F$ can be found by computing a singular value decomposition of $A = SUV^T$ and taking the elements of $F$ from the last column of $V$. Depending on the size of the matrix, it might be beneficial to compute the svd on GPU using the CULA library. From the fundamental matrix, another important entity called the essential matrix $E$ can be extracted

$$E = K^T F K \quad (2)$$

where $K$ is a 3x3 camera calibration matrix. Rotation and translation vectors can then be extracted from $E$ and used in triangulation to find the 3D coordinates of the scene point. Current program to test the implementation of the code is made using the Windows SDL library for working with OpenGL and C++ TNT Jama linear algebra package for SVD decomposition. Further work will require some performance optimizations for this algorithm to be included in the real-time scene reconstruction system. Which frames from the video sequence to select as the view pairs for point cloud generation and how to register the partial point clouds that capture the same subsets of 3D points are some of the issues that need to be considered in the future. Additionally, other, more robust algorithms are to be researched and analyzed for performance.
An attempt to reconstruct a cube has so far resulted in the following point cloud.

Figure 3: Test structure from motion output
4 Mesh Generation

The first step was to get a 3D convex hull. We decided to use Timothy Chans 3D convex hull generator. His algorithm is a modified version of the divide-and-conquer algorithm. The approach is to look at the 3D convex hull problem as a kinetic 2D problem. To solve it as a kinetic 2D problem we must project the 3D points into 2D, find the convex hull, and then project the points back to 3D. Looking for the lower and upper hull is the same thing, so code need only be written for one half. The divide-and-conquer comes in for each of the hulls. Each half of the hull is split in half and the hull for each quarter is found. Then all the hulls are bridged together in the end. The input is a list of 3D points and the output is an array of events. The output array is basically represents the order in which the points are inserted into the convex hull.

Here is a sample output from the algorithm. The input was a point cloud of a birdhouse. Somehow our understanding of what the algorithm outputs is not up to par. The points that are outputted should be in the set of points used as input. However, after looking at the output we find that the resulting points are not of the point cloud data.

Figure 4: Test convex hull output
5 Conclusions

We have achieved only partial success in the implementation of our system. However, after analyzing the algorithms used in each of the stages, we have come to a conclusion that the real-time scene reconstruction system is feasible. The obvious trade-off between performance and quality dictates that the quality of the 3D models will suffer. But judging by the rapid development of the research in scene reconstruction, we strongly believe that in the nearest future it will be possible to have a high quality real-time 3D scene reconstruction system running on conventional hardware.
Bibliography


