Aural Imaging from 3D Vision

by

Jasper Fourways Vicenti

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[June 2006]
May 2006

Author ....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by...
David Demirdjian
Research Scientist
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Aural Imaging from 3D Vision

by

Jasper Fourways Vicenti

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Dense stereo maps have many useful applications in computer vision. They can help in performing such tasks as background separation, segmentation, and object recognition. Despite the continued exponential advances in the computational capacity of microprocessors, current personal computers are limited in their ability to adequately solve the dense stereo vision problem in realtime. Modern processors now contain a SIMD instruction set architecture that can provide substantial improvements in performance when working with specific data types. This program demonstrates a stereo vision algorithm that implements commonly known methods in computer vision and takes advantage of the Altivec instruction set to achieve realtime performance. Once the stereo data has been processed, the depth map that is produced can be used to provide an additional dimension of data to the user. The software demonstrates a possible use for this data, as an aural aid to people that are blind or vision impaired. The system uses the depth information to identify obstacles in a user's path and alert the user with aural feedback.

Thesis Supervisor: David Demirdjian
Title: Research Scientist

Contents

Chapter 1

Introduction

The goal of computer vision is to be able to develop an understanding of a scene from one or more sources of visual data. One of the greatest difficulties is to be able to transform that data into useful information. Vision algorithms can provide a description of a scene that can be of greater use to a user or when integrated into a larger computer system. This has the potential to significantly reduce the amount of data being provided to the user by eliminating redundant information.

As part of a security system, a vision system could have a number of uses. For example, a security system that has many more cameras than screens might use computer vision to display the most important cameras based on various priorities. They could show cameras which are exhibiting unexpected movement in the image, track pedestrians, or even highlight suspicious behavior. This system could intelligently archive and summarize visual data, only displaying and storing relevant parts of the video.

Of all of our senses, sight allows us to gain a detailed understanding of the world around us. Our eyes help to recognize objects from afar and pinpoint the motion of these objects and ourselves. The ability to see enables us to engage in action sports such as soccer or baseball where we must track a small, fast moving object. Additionally, our eyes allow us to interact with these objects using hand-eye coordination. All of this happens seamlessly with our other senses, enabling us to coordinate with other people as a team.

In the early evolution of man, without sight, we would be at a distinct disadvantage to predators and in hunting our prey. Without the aid of others, a vision-impaired person would likely have great difficulty surviving on their own for very long. Even in a modern, urban environment, navigating through unfamiliar territory would be a daunting effort without some form of outside assistance.

While having the use of one eye allows a person to see very well, the addition of the second eye helps our brain to generate a three-dimensional representation of the scene in front of us. This drastically improves our ability to track moving objects, estimate distances, and perform tasks requiring complex user interaction. With a stereo vision algorithm, providing a computer system with a similar capability can further enhance its utility.

1.1 Motivation

Stereo vision algorithms can be extremely computationally intensive. Existing algorithms tend to have the primary goal of accuracy. As a result, most of these algorithms are not suitable for use in real-time or near real-time applications. One of the goals of this program is to achieve real-time performance while still allowing for sufficient post-processing to occur on the same CPU.

Many existing desktop processors consume vast amounts of power when running at full speed. These CPUs were even put into laptop computers to act as desktop replacements, at the expense of battery life. While just recently there has been a trend to improve their performance per watt, many of the processors have not been adequate for embedded use. This has led to dual core chips to try to improve performance without significantly increasing power consumption. This is not a new idea, however. Most modern processors contain a special vector processing unit that is intended to vastly accelerate the processing of certain types of data. This program takes advantage of the vector processing unit of the PowerPC G4 chip.

An additional requirement is to ensure that the system is reasonably portable. A part of this constraint is that the system needs to be based on a computer that is both light and power efficient while still having adequate processing capability to achieve real-time performance. The program currently operates sufficiently on a laptop computer almost two years old.

Finally, the depth map is then post-processed in order to highlight an example of usage of the stereo information. For people that are blind or vision-impaired, they must rely on their other senses to navigate. The goal of this program is to assist in navigating unfamiliar areas by warning about obstacles and identifying the clearest path. By generating aural feedback to the user, they will be assisted in navigating a path that may contain obstacles.

1.2 Task

A pair of stereo video feeds alone are usually not very useful to a user. As an example, a remotely controlled robot may have stereo cameras mounted to aid in navigating difficult terrain. The Mars landers have stereo cameras installed, but there is no real-time component involved. Due to the computational requirements that a stereo algorithm would require, they are not used for real-time analysis. An embedded processor with the capacity to run a real-time or near real-time algorithm would likely exceed the power consumption limitations of that system. Neither are the stereo cameras actively used for human-controlled navigation due to the distance from the control center. In general, they are saved for further non-real-time analysis.

The program requires two standard CCD cameras to be attached to the computer. These cameras are positioned in a traditional stereo orientation, with both sensors in the same focal plane, offset by a fixed amount in the horizontal direction. The image data from the cameras is continuously captured by the program. This data will then be used as input by the algorithm in order to generate a depth map of the scene.

The program will take the analyzed depth map derived from the stereo vision algorithm and generate audio signals that the user will hear through stereo headphones. This will allow the user to pinpoint the source of the audio signal in the scene and hence understand the direction of the obstacle or path which caused the signal. More detail will be given in the subsequent chapters.

Chapter 2

Background

This chapter describes the concepts and terminology that will be necessary to understand the implementation of the system. It begins with an overview of the current research paths being pursued in the field of stereo vision, and will describe in further detail about existing real-time algorithms. Next there will be a description of how humans perceive sound. This will be integral in providing a useful soundscape. Finally, the chapter will conclude with details of the vector processing unit which will allow for a significant improvement in execution speed for the stereo algorithm and post-processing steps.

2.1 Stereo Vision

Significant advances in stereo vision algorithms continue to be made year to year. An objective comparison of the most recent algorithms are available for evaluation online[5]. Newly developed algorithms can be submitted to compare against those developed by others. The Middlebury website provides ranking based on the errors in all regions, non-occluded regions, as well as regions near depth discontinuities that would be close to occluded regions. Unfortunately the ranking does not have an additional column for runtime performance. Although this would be difficult to coordinate due to differences in platforms and hardware used, it is still important for the real-time subclass of stereo vision.

Stereo Vision Approaches

While there continues to be an outpouring of new or modified algorithms, the majority of these can be classified into one of two groups. Global optimization methods seek to minimize an energy function with a data term and a smoothness term (Equation 2.1). The E_{smooth} value measures the degree to which the mapping f is not piecewise smooth, whereas E_{data} measures the disagreement between f and the observed data. The smoothness term can be arbitrarily defined, but it generally is used to ensure that the solution is mostly continuous in nature.

$$E(d) = E_{data}(d) + \lambda E_{smooth}(d) \tag{2.1}$$

Generally there are a number of iterations that an algorithm performs, each time improving the results. A typical global optimization method represents the images as a directed graph and calculates the max-flow through the graph or applies graph cuts to find the minimum energy[3]. Graph cuts can model the effects of discontinuities well, since the occluded pixels in each image will be removed during the cut. In most of the algorithms for global optimization, the running time is far worse, ranging from tens of seconds to thousands of seconds in the worst case.

The second group of algorithms is based on local optimization methods, such as using normalized cross correlation is an example of a local optimization method. Most local methods use a window around each pixel, with either a fixed window, multiple windows of different sizes[2], or variable-sized window[1]. Matching cost computation is generally calculated using sum of squared differences [SSD] or sum of absolute differences [SAD], the latter usually being less computationally intensive. Equation 2.2. Most of the windows are rectangular to achieve a faster runtime, however arbitrarily-shaped window algorithms do exist[4].

$$\int\int (E_l(x + \partial x/2, y) - Er(x - \partial x/2, y))^2 dxdy \tag{2.2}$$

While global optimization tends to produce more accurate results than local optimization methods, their computational requirements are much higher. Because of this, the system described in this paper is based on the local optimization method of least sum-of-squares differences.

Although there are proposals that improve the behavior slightly around depth discontinuities, they still suffer from errors in low texture regions. The benefit to local optimization algorithms is that they are very fast, running in the order of seconds for quarter-vga (320x240 pixel) test images. The methods optimized for speed run in the order of hundreds of milliseconds.

Many of the more successful stereo vision algorithms to date use belief propagation to propagate areas where the algorithm has greater confidence in their value to areas of lesser confidence. This can greatly increase computation time due to the large number of iterations required to approach a steady state, however it is fairly successful due to some of the difficulties in stereo vision.

A summary of the latest approaches to stereo vision, including local optimization techniques such as SAD, SSD, and global optimization techniques such as graph cuts and belief propagation can be found in this paper and corresponding web site[5].

Difficulties in Stereo Vision

One of the difficulties in applying a stereo vision algorithm is that there are often areas that are not very textured and therefore have very low contrast. This could be due to many factors, such as an image region with a solidly painted wall or a patch of bright sky, or even areas of the image that are washed out due to bright lights or poor image calibration.

Similarly, one of the more common problems is that very dark regions will have low contrast and thus be subject to increased camera noise. This is especially a problem for low-cost CCD cameras. Because stereo vision algorithms rely on more than one camera, the noise is also amplified because each camera has their own unique noise signature independent from the other camera. Additionally, each individual channel may exhibit their own unique noise properties. The blue channel, for example, tends to exhibit greater amounts of noise than the other channel. For algorithms that only rely on a single channel to generate a depth map, there may be a tradeoff between using only one of the channels versus downmixing all three channels to generate a lightness channel to work from. Also, since most CCD cameras use a Bayer layout of the individual pixels, using only the green channel may result in reduced resolution.

Algorithms that only produce depth maps in areas of high contrast generate what is called a sparse depth map. Alternatively, an algorithm that tries to generate a depth value at each source pixel is called a dense depth map. Local optimization algorithms may produce a dense depth map, but they are more likely to be marred by errors in low contrast regions due to only being able to use local data for its calculations.

Another difficulty for stereo vision algorithms is when trying to resolve areas around depth discontinuities. This is due to the fact that a pixel in one camera may not have a corresponding pixel in another camera, as that pixel would have been occluded by some other image region that is closer to the camera.

Finally, there are issues that arise from the design of the stereo camera. A slight misalignment of the two cameras is almost inevitable. Additionally there is radial and other types of distortion in the lens which can impact the accuracy of the results. Some of this can be corrected with pre-processing, however any pre-processing will generally require calibration and an increase in processing time. Some problems cannot be corrected. Different lighting conditions could be detrimental to the behavior of the algorithm. A highly reflective surface like a mirror would appear to the algorithm to be further away than it actually is.

2.2 Aural Perception

Just as the use of both eyes enhances the ability to see, so does having two ears. With both ears, objects that produce sound can be pinpointed relative to the listener. Even in a room with complex acoustic properties that may cause reverberation and echoes, a person brain can generally compensate for all of these anomalies to identify the true source of the sound. This is effective even for sounds that are above or behind the listener. This property will be especially useful when trying to identify the specific direction of obstacles in the current path.

An experienced listener could listen to a symphony orchestra and effectively separate out and listen to each individual instrument or set of instruments from the rest of the group. Being able to separate multiple distinct sounds simultaneously could be useful in this stage of the program.

When the use of one of our senses becomes unusable, the other senses become more acute to compensate. For people that are blind or visually impaired, their sense of hearing and touch adapt to compensate for the lack of sight. A byproduct of this is that it becomes easy to train a blind person in techniques that take advantage of their heightened senses, such as reading braille. In this case, the increased sensitivity to sound will be taken advantage of.

The human ear is sensitive to sound with frequencies as low as 20 Hz to frequencies up to 20 KHz. As a person ages, high frequency sensitivity decreases to approximately 16 KHz. This still leaves a wide range of frequencies that are potentially usable for conveying information to the user. Also available as a variable is the amplitude of the sound, which can also be modulated as required. The range of amplitude in human hearing is 130 dB. Finally, with stereo headphones the left-right balance of the sound source can be adjusted to allow for differing sound sources, as well as the time delay between the left and right headphones. Because of the size of the human head, a sound coming from the left takes 0.63 milliseconds longer to reach the right ear than the left. Suppose two identical sounds are played back to a listener. If the two sounds are played back at the same time, the brain will assume that the two sounds are from the same source. As the delay between the sound increases, our brain will still associate the sound source as being the same. This principle is known as the Haas Effect. The first instance of the sound is taken as the primary source, while any subsequent similar sounds within 25 - 35 milliseconds are essentially ignored by the brain. The program will seek to take advantage of as many of these aspects as is feasible within the real-time processing requirements.

There have been other efforts to derive sound from visual data. In one instance, the sound is produced as follows. Every second, the image is swept from left to right. As the image is swept from left to right, the sound is also swept from the left ear to the right ear. Areas in the upper part of the image are played back at higher frequency than regions in the lower part of the image. Bright regions in the image are played back louder than darker regions. The resulting sound is unfamiliar at first, but after some time spent training, one can become accustomed to the new format. One of the downsides with this approach is that because the visual data is often so complex, the sound produced is literally a cacophony[13]. The approach presented in this paper will seek to use vision techniques to parse the visual data and using the most relevant information.

2.3 PowerPC AltiVec SIMD Execution Unit

The AltiVec Processing Unit is a 128-bit SIMD (Single Instruction, Multiple Data) execution unit that is capable on performing operations simultaneously on 4 32-bit integers or floating point numbers, 8 16-bit integers, or 16 8-bit integers. This degree of parallelism is especially useful in image processing, since the image can be decomposed into its red, green, and blue 8-bit channels, allowing for operations on up to 16 channels at a time, opening up the opportunity for significant improvement in calculation speed. The 128-bit data sets are referred to as vectors.

This parallelism does not come without a cost. As with most optimization at a low level, care must be taken to ensure that the CPU pipeline is sufficiently full and branches are reduced to a minimum. Also, because the instructions can operate on a large amount of data at once, performing operations on individual bytes of data can be wasteful and slow.

Some of the operations that will be used in this algorithm are described below. The AltiVec unit compares favorably with other SIMD implementations on personal computer chips. AltiVec has available 32 vector registers for use. For comparison, Intels Multimedia Extension (MMX) unit has 8 64-bit registers and Streaming SIMD Extension (SSE) unit has 8 128-bit registers. This degree of parallelism will be examined further.

2.4 AltiVec Operations

Similar to other execution units on the CPU, the AltiVec Unit has specific execution units that perform operations on its data. AltiVec has explicit instructions for loading and storing 16-byte aligned data from and to memory, which is performed by the Load-Store Unit (LSU). To work with unaligned data, the Vector Permute Unit (VPERM) allows for an arbitrary assignment of values from two vector registers into one using a third permute parameter. The Vector Simple Integer Unit (VSIU) performs many of the tasks of the ALU, such as add, subtract, min, max, as well as boolean and comparison operations. There is also a Vector Complex Integer Unit (VCIU), which performs more specialized integer calculations. As an example, the multiply-sum instruction can multiply two byte-word vectors together and add the resulting elements to a third 4 byte-word vector. This can be accomplished in one cycle, whereas for a standard ALU, this might take 30 to 60 or more cycles. Multiple instructions can also be dispatched to the different units, enabling even greater work per clock cycle to be performed.

Programming in AltiVec resembles the programming model for C. AltiVec uses standard variables defined in the form vector unsigned byte. AltiVec instructions can be interspersed with standard C code, including for loops and if statements. Detailed below are a subset of the AltiVec instruction set which are necessary for the vision techniques used in this paper. Besides the load and store operations, arithmetic operations make up the majority of the useful operations.

Load and Store

In AltiVec, it is necessary to perform loads and stores explicitly. These operations will be 16-byte aligned. To load data, a pointer to a specific data location is passed. Additionally a byte-offset amount can be specified from that pointer, although the load will still only be 16-byte aligned. One side effect of this behavior is that the AltiVec execution unit and integer execution units cannot work on the same data registers. Therefore in order to access one register from the other, both an integer load-store and a vector load-store must occur. This can be costly in terms of cycle latency as a roundtrip to memory may result in tens or hundreds of wasted cycles.

$$\mathrm{a} = \texttt{vec\_ld(0,datapointer);} \tag{2.3}$$

Similarly, the store instruction will push the data back to memory. The vector a is now passed in with the desired location to store the vector. This is stored in a 16-byte aligned location.

$$\texttt{vec\_st(a,0,datapointer);} \tag{2.4}$$

Addition and Subtraction

This instruction will add or subtract each corresponding element of b to or from the corresponding element of a and put the result into c. In other words, it performs $c = a \pm b$ for all elements.

$$\texttt{c = vec\_add(a,b);} \tag{2.5}$$

$$\texttt{c = vec\_sub(a,b);} \tag{2.6}$$

In order to combine data within a vector, there is an additional instruction which will sum across all elements in the vector. Given a vector of 32-bit integers, it will add all four integers together, placing them in the rightmost element. If the numbers are {5, 6, 7, 8}, the resulting vector will be {0, 0, 0, 26}.

$$\texttt{c = vec\_sums(a);} \tag{2.7}$$

Bitwise Shift

This instruction will shift each corresponding element of a left or right b bits and put the result into c.

$$\texttt{c = vec\_sl(a,b);} \tag{2.8}$$

$$\texttt{c = vec\_sr(a,b);} \tag{2.9}$$

Minimum and Maximum

This instruction will take the minimum or maximum of each corresponding element of a and the corresponding element of b and put the result into c. These instructions are equivalent to `c = a < b ?a:b;` and `c = a > b ?a:b;` respectively for each element.

$$\mathtt{c} = \mathtt{vec\_min}(\mathtt{a}, \mathtt{b}); \tag{2.10}$$

$$\mathtt{c} = \mathtt{vec\_max}(\mathtt{a}, \mathtt{b}); \tag{2.11}$$

Fused Multiply-Add

For our purposes, the 8-byte multiply-add is used. In this example, a and b are 8-bit vectors and c is a 32-bit vector. This instruction will take the product of the corresponding elements of a and b and add each set of 4 8-bit integers to the corresponding element of c.

$$\mathtt{d} = \mathtt{vec\_msum}(\mathtt{a}, \mathtt{b}, \mathtt{c}); \tag{2.12}$$

Permute Operations

Probably the most powerful part of the SIMD instruction set are the operations which perform permutations. The permute unit allows arbitrary reordering and selection of two vectors based on a third vector. In the equation below, each element in d is chosen from a or b based on the lower 5-bit value of c. To illustrate this, if the values of c are {0x0, 0x1, 0x2, 0x3, ... ,0xE, 0xF}, then d would be equal to a after the operation. These values of c are known as the *identity permute.*

$$\mathtt{d} = \mathtt{vec\_perm}(\mathtt{a}, \mathtt{b}, \mathtt{c}); \tag{2.13}$$

The identity permute can be very useful to align data to the desired position. The vector load-for-shift-left operation takes the identity permute and adds a given value to each element in the vector. For example, if datapointer is 16-byte aligned, then the equation below will add 1 to each value. Thus the resulting vector will be {0x1, 0x2, 0x3, ..., 0x10). When this vector *shift* is then passed in to the `vec_perm` operation, it will shift the concatenation of a and b by 1.

$$\texttt{shift} = \texttt{vec\_lvsl}(\texttt{1}, \texttt{datapointer}); \tag{2.14}$$

There is an equivalent instruction, `vec_sld`, which performs shifting in one step. For example, the following operations are equivalent.

```
shift = vec_lvsl(1, datapointer);
    c = vec_perm(a, b, shift);
```

and

```
c = vec_sld(a, b, 1);
```

The `vec_splat_u8` instruction, as the name implies, will place a scalar value into all elements of the vector. This particular version of the instruction works on unsigned byte vectors.

$$\texttt{allfives} = \texttt{vec\_splat\_u8}(5); \tag{2.15}$$

Chapter 3

Design

3.1 Stereo Vision Algorithm

The primary goal is to achieve a very fast algorithm that still achieves good results when compared to existing algorithms. It will be an implementation of a local optimization algorithm to assist in achieving this performance target, but an image pyramid will be used to avoid some of the negative aspects with using this type of optimization. Towards that goal, the runtime of each of the methods in the algorithm is analyzed to ensure full use of the parallel features of the AltiVec chip.

The intention is to present an algorithm that uses an image pyramid and runs recursively beginning at a very general level and finishing on a more detailed level. This is typically used in computer vision to separate out high frequency and low frequency changes in the image in order to analyze the levels separately.

In the analysis of the running time for each section of the algorithm, the performance improvement gained by taking advantage of the parallelism inherent in the SIMD machine is analyzed. As with external memory algorithms, where the algorithms are measured in disk blocks rather than data items[9], the algorithm presented here will take a similar approach. In this case, a vector represents a specific number of pixels. The variable B is defined as the level of parallelism provided in the program. Since the algorithm is working primarily in sets of 16 bytes, the value of B will generally be 16.

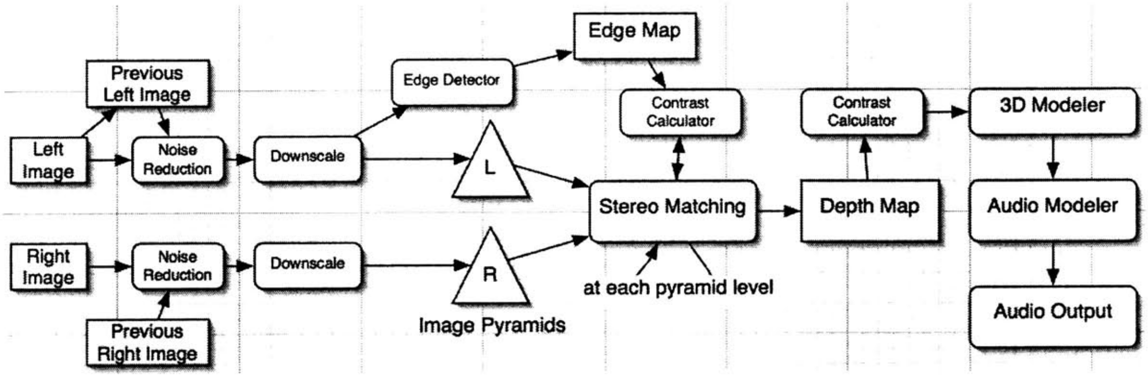

Figure 3-1: System Diagram

3.2 Local Methods

The following methods describe the algorithm in pseudocode for a straightforward understanding of the approach.

Comparison Calculation

In order to compare a source vector v with a vector w and disparity range from 0 to d, both vectors are loaded into registers and a permute function is used that shifts in one byte for every value of i. If d is greater than B, the level of parallelism gained by using AltiVec, then the vector (w + 1) is assigned to w and the adjacent vector (w + 2) is loaded. The difference can be calculated by using the vector commands for max, min, and subtract. This can be done since the operations are working with unsigned bytes. The sum the square of the differences is calculated using Multiply-Sum (vec_msum) as described above. Sum-Across-Vector (vec_sums) sums the across the remaining elements in the vector (now 32-bit numbers) so that it can be compared to the scalar value bestcorrelation.

```
COMPARE-VECTOR-OFFSETS($v, w, d_{min}, d_{max}$)
 1  $a \leftarrow$ LOAD($v$)
 2  $b \leftarrow$ LOAD($w$)
 3  $c \leftarrow$ LOAD($w + 1$)
 4  $bestoffset \leftarrow 0$
 5  $bestcorrelation \leftarrow 999999999$
 6  for $i \leftarrow d_{min}$ to $d_{max}$
 7      do
 8          $e \leftarrow$ PERMUTE($b, c, i$)
 9          $max \leftarrow$ MAX($a, e$)
10          $min \leftarrow$ MIN($a, e$)
11          $diff \leftarrow$ SUB($max, min$)
12          $sd \leftarrow$ MSUM($diff, diff, zero$)
13          $ssd \leftarrow$ SUMS($sd$)
14          if $sd \leq bestcorrelation$
15             then $bestoffset \leftarrow i$
16  return $bestcorrelation$
```

In terms of the amount of time required to perform pixel offset calculations, this loop requires d loops to calculate all d offsets for the pixels in a vector. Since there are B pixels in a vector, the runtime of Compare-Vector-Offsets for n pixels is O(n/B).

Downscaling

Downscaling of the image is necessary to ensure that there is good coverage of low-texture areas. The low-texture areas will be ignored for the high resolution parts of the image. Thus the low-resolution version of the image in the image pyramid is used. Given an array of vectors V, number of vectors in a row r, and height h, output an array W that is half the width and height. The values *mergefirst* and *mergesecond* and used to interleave the vectors into the correct format in order to take the average in the horizontal direction.

```
DOWNSCALE($V, W, r, h$)
 1  $i \leftarrow 0$
 2  $j \leftarrow 0$
 3  for $j \leftarrow 0$ to $h/2$
 4      do
 5          for $i \leftarrow 0$ to $r/2$
 6              do
 7                  $a \leftarrow V[2jr + 2i]$
 8                  $b \leftarrow V[r(2j + 1) + 2i]$
 9                  $c \leftarrow V[jr + 2(i + 1)]$
10                  $d \leftarrow V[r(2j + 1) + 2(i + i)]$
11                  $temp1 \leftarrow$ AVERAGE($a, b$)
12                  $temp2 \leftarrow$ AVERAGE($c, d$)
13                  $temp3 \leftarrow$ MERGE($temp1, temp2, mergefirst$)
14                  $temp4 \leftarrow$ MERGE($temp1, temp2, mergesecond$)
15                  $final \leftarrow$ AVERAGE($temp3, temp4$)
16                  $W[jr + i] \leftarrow final$
17  return $bestcorrelation$
```

The runtime of Downscale is O(n/B) where B is the level of parallelism gained by using the AltiVec instructions.

Calculating Contrast

Since we want to ensure that an area of low texture is not calculated upon with too small of a grid size, we want to calculate the contrast value of a particular vector. This is similar to the Compare-Vector-Offsets method as described above, except the vector is computed against itself.

```
CALCCONTRAST($v$)
1  $a \leftarrow$ LOAD($v$)
2  $b \leftarrow$ PERMUTE($a, a, 1$)
3  $max \leftarrow$ MAX($a, b$)
4  $min \leftarrow$ MIN($a, b$)
5  $diff \leftarrow$ SUB($max, min$)
6  $sd \leftarrow$ MSUM($diff, diff, zero$)
7  $contrast \leftarrow$ SUMS($sd$)
8  return $contrast$
```

Again, for the similar reasons as above, perform B calculations against n pixels, this method runs in O(n/B) time.

Dynamic Program

The dynamic program will be a recursion which will recursively call itself until reaching the most general level. Once there, it will calculate the contrast for each vector and corresponding vector in Q, returning the value back to an array of arrays which should allow for further analysis, areas of improvement.

```
CALCRECURSION($P, Q, r, h, minlevel, curlevel, maxlevel, d_{min}, d_{max}$)
 1 if $(curlevel \leq maxlevel) AND (curlevel \geq 0)$
 2    then DOWNSCALE($P, P'$)
 3         DOWNSCALE($Q, Q'$)
 4         CALCRECURSION($P', Q', r/2, h/2, minlevel,$
 5         $curlevel + 1, maxlevel, d_{min}/2, d_{max}/2$)
 6         for $v \in P$
 7             do if CALCCONTRAST($v$) $\geq x$
 8                   then $d[curlevel].[v] \leftarrow$
 9                        COMPARE-VECTOR-OFFSETS($v, w, d_{min}, d_{max}$)
10    else for $v \in P$
11             do $d[curlevel].[v] \leftarrow$ COMPARE-VECTOR-OFFSETS($v, w, d_{min}, d_{max}$)
```

We start by calling CalcRecursion with the following arguments. We set curlevel to 0, maxlevel to the highest level of recursion we want, and minlevel to the minimum recursion level.

3.3 Aural Representation

The source data consists of the depth map produced by the stereo vision algorithm. Using this data, appropriate sound can be produced that correlates to this data. There are numerous approaches to converting the visual data to aural data.

In this particular application, the goal is to provide a simple enough model that will provide the user with an indication of current obstacles and paths. The depth map will thus be modelled as a terrain, with peaks and valleys. The peaks correspond to obstacles and valleys correspond to paths that are unobstructed.

Whichever representation is chosen, the current limitation is the available processing power. After allocating a certain amount of computational time in order to capture the image data and run the stereo vision algorithm, there are limited resources available to both process the depth map, produce the sound representation, and output the sound to the headphones. As a result, a simpler transformation from image to sound is necessary to retain its real-time response.

Preprocessing

Similar to the windowed approach used in a sum-of-squares differences algorithm, this same type of algorithm can be used effectively within the processing constraints. The sum of the contents of the window can be calculated easily using vector operations. Using a preset window size, the window can be scanned across the depth map. A record of the largest and smallest sums and their relative locations in the image are tracked and prioritized.

```
AURAL-PREPROCESS($depthmap, width, height$)
 1  $pathlist \leftarrow \{\}$
 2  $obstaclelist \leftarrow \{\}$
 3  for $j \leftarrow 0$ to $height$
 4      do
 5          for $i \leftarrow 0$ to $width$
 6              do
 7                  $w \leftarrow$ WINDOW($depthmap, i, j$)
 8                  $x \leftarrow 0$
 9                  for $k \leftarrow 1$ to SIZE($w$)
10                      do $x \leftarrow x +$ CALCCONTRAST($w[k]$)
11                  if $x \leq paththreshold$
12                     then $pathlist \leftarrow pathlist \cup \{x, i\}$
```

```
13              if $x \geq obstaclethreshold$
14                then $obstaclelist \leftarrow obstaclelist \cup (x, i)$
15  AURAL-OUTPUT($pathlist, obstaclelist$)
```

Aural Output

With the highest priority obstacles and paths determined, the next task is to produce sound from this information. Using the industry standard MIDI protocol, the sounds can be generated using the built-in MIDI framework. This allows for a playback of a potentially large number of simultaneous preset music instruments with full control over amplitude, pitch, and balance. Unique instruments can be used to represent the derived visual information.

For the sounds produced, the amplitude will be used to indicate the confidence in the path or obstacle. This can be based on how close the obstacle is or how large it is. The balance of the sound outputted is based on the location of each respective sound source. Since the field of view of the input images are limited to not more than around 45 degrees, the sound output should also be limited to a similar range. While this does not take full advantage of the full left-right sound output potential, the sound output maps more naturally than it would otherwise. For example, as a user pans the camera from left to right, the sound output will also pan at the same rate.

```
AURAL-OUTPUT($pathlist, obstaclelist$)
1  SORT($pathlist, 0, ascending$)
2  SORT($obstaclelist, 0, descending$)
3  for $p \in pathlist$
4      do
5          $amplitude$ = NORMALIZE($p[0]$)
6          $balance$ = CALCBALANCE($p[1]$)
7          PLAYPATHSOUND($amplitude, balance$)
8  for $o \in obstaclelist$
```

```
    do
        $amplitude = \textsc{normalize}(o[0])$
        $balance = \textsc{calcbalance}(o[1])$
        $\textsc{PlayObstacleSound}(amplitude, balance)$
```

Chapter 4

Implementation

4.1 Camera Setup

The baseline distance between the two cameras is approximately 8 cm. The stereo rig is based on small and inexpensive fixed-zoom cameras. The cameras are connected in series using IEEE-1394 cables. The program uses the sequence grabbing feature of QuickTime in order to capture images from the two cameras. While the cameras support color, for the purpose of this program the images are captured in grayscale mode. This reduces the image capture overhead significantly. To also reduce processor usage, the cameras are configured to capture at 320x240 resolution. Although the cameras theoretically support 640x480, the image quality and noise are negatively affected. Image quality and noise also

4.2 Camera Calibration

Both cameras are mounted to a fixed bracket, which results in near parallel epipolar lines. As a result, minimal camera calibration is performed at this stage in development. The system assumes that the epipolar lines are parallel or nearly parallel. Two procedures are used to provide adequate results. First, calibration is performed over

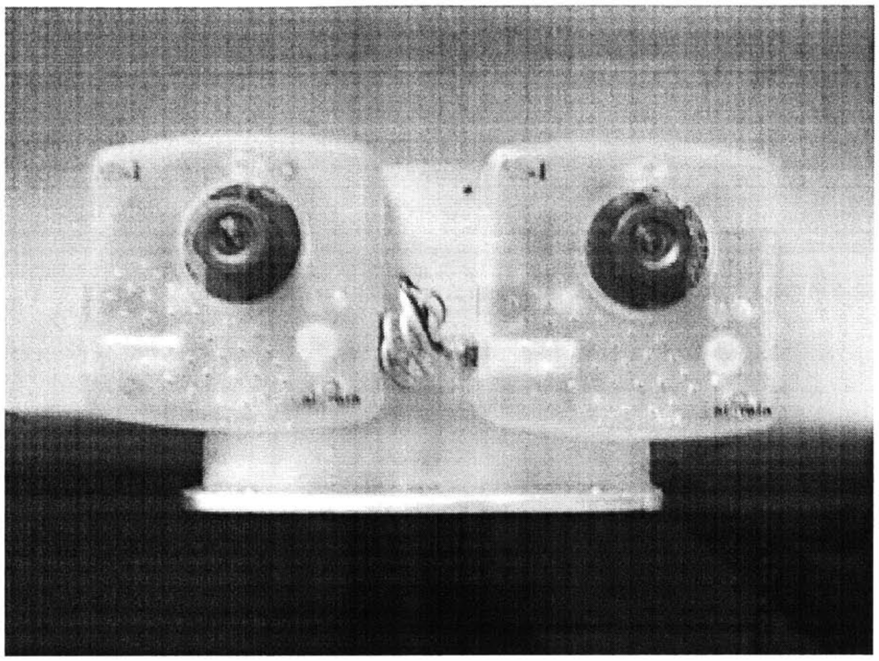

Figure 4-1: Camera Setup

a large support window by least sum-of-squares of differences. This window is shifted $\pm\partial y$ and $\pm\partial x$ a fixed amount based on the size of the images. Since the cameras can capture at 320 x 240 or up to 640 x 480, it is necessary to compensate for arbitrary sizes. A more robust implementation might choose a few large support windows in various locations of the image and choose the best matching y offset from these.

Additionally, during correlation matching, to compensate for some small amount of distortion, the algorithm will shift the window up and down a single pixel. While this triples the number of lines required to test over the ideal case, adequate performance is still maintained.

Another issue that is not addressed is that of lens distortion. Equation 4.1 can be used to eliminate radial distortion. Once k has been determined it need not to be recalculated for each frame. A suitable remapping from distorted to undistorted points, however, does. Further examination is required to determine the feasibility of implementing this in real-time. Since a graphics card is capable of doing this calculation very quickly, it may be best to offload this work to the GPU. This can be achieved quite easily in Mac OS X using a Core Image kernel[9], which is uses the OpenGL Shader Language to interact with the GPU.

$$x' = x(1 + kx^2) \tag{4.1}$$

4.3 Noise Reduction

The current system uses off the shelf IEEE-1394 cameras. These cameras exhibit significant amounts of noise in all regions of the image regardless of brightness level. Because this is a real-time system, complex math to remove this noise may not be possible. The noise exhibited in the image consists mostly of what appears to be uniform gaussian noise. Additionally there is often noise resulting from an unidentifiable interference source. To try to remove some of the noise, a simple technique is used to take the average of the current frame with the previous frame in Equation 4.2.

This adequately removes the noise at the expense of some blurring when the cameras are moving. In the static images shown, it is more difficult to see the reduction in noise. It is much more apparent in the video image. Multi-frame averaging further suppresses noise, however the blurring increases accordingly. Since the blurring occurs in both cameras equally, the effect is not that detrimental to the end result. General obstacle detection still works to a certain extent with slightly blurred images. In future versions, the noise reduction algorithm could be enhanced by using optical flow to assist in choosing an offset appropriate for the new frame prior to averaging.

$$E_{i,j,k} = 2(E_{i,j,k} + E_{i,j,k-1}) \tag{4.2}$$

```
grayin1 = vec_ld(0, indata);
grayout1 = vec_ld(0, outdata);
grayin1 = vec_avg(grayin1, grayout1);
```

```
   vec_st(grayin1, 0, outdata);
```

4.4 Edge Detection

Stencils, or computational molecules, can be used to provide discrete approximations to continuous equations[7]. The code below calculates the unsigned estimation of $\partial E/\partial x$ for all 16 pixels simultaneously. Note that as with the scalar discrete approximation, the estimation is for a point midway between each pixel location. In the current implementation, this detail may not have a pronounced effect. Alternatively, the function could use the difference between E_{i-1} and E_{i+1} with little loss in performance.

```
   first = vec_ld(0, data); // Load the first vector
   second = vec_ld(16, data); // Load the second vector

   temp = vec_sld(first, second, 1); // Shift the concatenation of vectors
left by 1

   max = vec_max(temp, first);
   min = vec_min(temp, first);
   final = vec_sub(max, min); // Calculate the unsigned difference from
the min and max

   vec_st(final, 0, outdata); // Store the vector to our edge buffer
```

Since the goal is to maintain computation with unsigned byte values, calculating the gradient using the squared differences would not be helpful. The current implementation calculates the unsigned brightness difference in the x and y direction. In order to prevent overflow or saturation, the x and y values are shifted prior to adding together. (Figure 4-2). If signed brightness difference is necessary, this can be performed as well using additional shifts. (Figure 5-5).

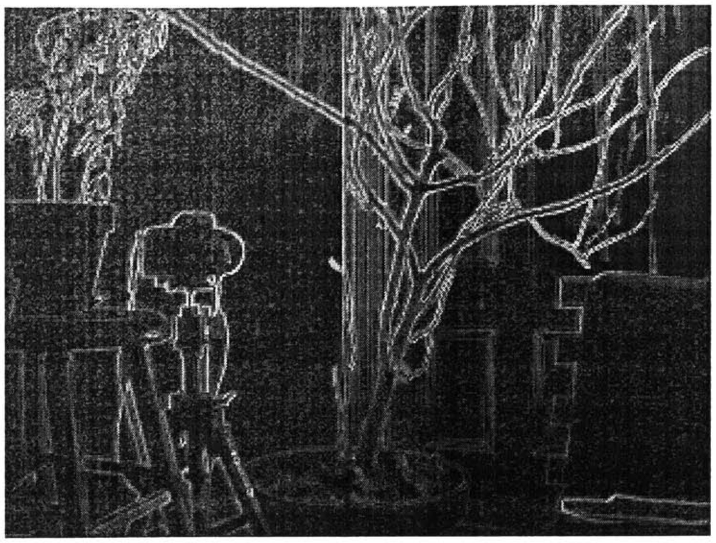
Figure 4-2: Unigned edge detection

4.5 Image Pyramid

Local optimization methods usually perform poorly when attempting to match areas of low contrast. Outside of the active window region, little or no data is used. While this is a win in terms of runtime performance, it means that low contrast regions will be dominated by noise and thus will not correspond to the correct depth because noise in the left image is correlating to other noise in the right image.

An image pyramid is constructed in an attempt to provide better correlation matching in areas of low contrast. For each level, the width and height of the image will be reduced by a factor of two, as shown in Equation 4.3. This is essentially similar to using a support window double the standard size, without the exponential performance hit.

$$E'_{i+1/2,j+1/2} = (E_{i,j} + E_{i+1,j} + E_{i,j+1} + E_{i+1,j+1})/4 \tag{4.3}$$

The image pyramid is generated by loading four vectors: $v_{i,j}$, $v_{i+1,j}$, $v_{i,j+1}$, and $v_{i+1,j+1}$. They are averaged horizontally and vertically and then merged. The new image point is located at the center of the old image points. This is repeated successively for the total number of desired levels.

```
   mergeFirst = (vUInt8) (0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E,
0x10, 0x12, 0x14, 0x16, 0x18, 0x1A, 0x1C, 0x1E),
   mergeSecond = (vUInt8) (0x01, 0x03, 0x05, 0x07, 0x09, 0x0B, 0x0D, 0x0F,
0x11, 0x13, 0x15, 0x17, 0x19, 0x1B, 0x1D, 0x1F);
   first = vec_ld(0,indata);
   second = vec_ld(16,indata);
   third = vec_ld(0,indata + j);
   fourth = vec_ld((16,indata + j);
   temp1 = vec_avg(first, third);
   temp2 = vec_avg(second, fourth);

   temp3 = vec_perm(temp1, temp2, mergeFirst);
   temp4 = vec_perm(temp1, temp2, mergeSecond);
   final = vec_avg(temp3,temp4);
```

When performing the correlation match described later, the top (smallest) level is calculated first. Then the next larger level is calculated, continuing until the original, full size level is reached. Future implementations could continue this progression downwards to provide sub-pixel disparity estimation at the expense of performance.

One difficulty in working with the image pyramid is that because the images decrease in size and the window remains the same size, a decision has to be made about how to fill in the depth map at each level. The initial attempt was to scale the depth map up at each level. This did not work well because motion at the top level became overly scaled relative to the lowest level. This means the higher level disparity maps would have a border around them. An attempt was made to fill that border in by extending the edge of each level, however this provided poor results. When noise caused certain values to be flipped, a large part of the depth map would change.

The image pyramid does provide a major advantage. The number of disparity offsets to check can be drastically reduced if there are already results one level higher. This is used to reduce the number of calculations significantly. A much smaller set of disparity offsets can now be checked.

4.6 Discriminator

The goal of the discriminator is to decide whether the depth value at a given pixel will be used or if it will be based on neighboring pixels. Areas of low contrast will have a lower confidence and thus more likely to use neighboring depth values over local values. Two types of discriminators are introduced. The initial attempt was to take the sum of squares of all of the edge values in the current 16x16 window. While this provides a good indication of contrast in the current region, it does not indicate accurately the current pixel's relative contrast. A smaller window would likely help in this situation. The code calculates edge squared sums for 16 elements in parallel. It performs two adjacent vectors 8 multiply-sums, followed by using the permute unit extensively to reorder the values for each appropriate vector. Refer to the Appendix for the detailed code.

4.7 Stereo Matching Algorithm

The major limitation of the PowerPC G4 chip is the bandwidth it has available to access main memory. Also, all computers are affected by the overall latency of accessing main memory compared to data in the cache. As computers continue to get faster, this latency will become more apparent. A cache miss can mean potentially hundreds of wasted cycles.

Even when data is prefetched from memory, the algorithm must be tuned so that the computational throughput does not exceed the available memory bandwidth. Because of the extensive parallelism provided with AltiVec, there is a minimum number of cycles of work that must be completed per cacheline loaded in order to maximize the processor's ability to consume data. For example, in [ref: apple perf memory], a 866 MHz G4 with 133 MHz bus speed needs to perform 50 cycles of work per cacheline loaded for optimum performance. Extrapolating these values for a computer running at 1.5 GHz G4 with 167 MHz bus, the algorithm must perform approximately 70 cycles of work per cacheline. Since the cacheline is 32 bytes wide for the G4 processor, that equates to 35 cycles per vector. Any fewer cycles per vector and the processor will be idle for parts of the algorithm waiting for more data to consume.

As a result, it is important to optimize the data sets to work within the limits of the L2 cache, which, in the case of this G4, is 512 KB. Some algorithms calculate all of the possible disparity values followed by subsequent calculation on those results. This operation would be fairly bandwidth intensive, as it requires, for 320 x 240 pixels and 96 offsets, over 7 megabytes.

The main loop of the correlation method works as follows. Load the vector for the desired window in the right image. For each disparity offset, load the approprate vector for the left image, shift it by the necessary amount, and calculate the sum of squares difference. In AltiVec, this looks like the following:

```
left1 = vec_ld(0,data);
left2 = vec_ld(16,data);
right1 = vec_ld(0,data);
right2 = vec_ld(16,data);
right = vec_perm(right1, right2, shiftforrightvector);

// get the correct window

left = vec_perm(left1, left2, leftvectorshift);

max = vec_max(left, right);
min = vec_min(left, right);
out = vec_sub(max, min);

total = vec_msum(out, out, total);

// aggregate the total for our full window height
```

Since the price to load the vectors from memory into the registers has already been paid, a few of these operations can be performed concurrently to ensure that the pipeline is full. The minimum total is kept track of for each disparity calculated. The best match gets chosen based on the lowest sum.

4.8 Aural Output

The code used in the discriminator can be reused to perform the calculation of paths and obstacles in the derived depth map. Rather than taken the sum of squares values of the calculated edge values, the sum of squares values are calculated from the depth map itself. At each pixel, a window around that pixel can be created and the sum of the values enclosing it can be calculated. Because of the parallel nature of the AltiVec processor, 16 pixel windows can be calculated in the time it would normally take to calculate one window with a standard integer unit. The permute unit is used extensively here to maximize use of the 32 available vector registers and minimize repetition of work while also dividing the workload between the permute unit, simple integer unit, and complex integer unit. This ensures increased throughput by reducing bubbles in the pipeline and enabling multiple instruction dispatches per clock cycle.

To understand the procedure, a 16-by-1 pixel window is used as an example. To perform a sum of squares calculation on one window requires a multiply-sum followed by a sum-across-vector. A permute is required to align the data for subsequent windows. Thus for the naive method, 15 permute instructions and 32 complex instructions are required. Two vector loads and 16 vector stores are also required because of the output of the sum-across-vector instruction format. While this appears to be fairly straightforward, the problem is that there is an issue with register starvation. Since 16 vector registers are required in the end result, there are only 16 available for the interim processing for this function. With an optimal compiler, this would not be a problem. However no compiler is perfect, and when the function is compiled in this form, there are excessive loads and stores resulting from register overload.

An alternative approach is to note that much of the work is being repeated when calculating the next adjacent window. Careful reuse by taking permutations of prior calculations can reduce the number of complex instructions required. This reduces the final output register requirements to four and eliminates the register spilling that had been occurring. This may provide an even greater increase in calculation performance.

Shown below is the implementation to produce an obstacle and a clear path value from the calculated contrast of the edge map around a window. The window size is 16x32 pixels, which is adequate to locate larger obstacles. With a smaller window (16x16), there is potential for false positives due to problems in the stereo matching algorithm. Finally, the balance values are calculated based on the size of the window compared to the current x location within that window.

```
   [self calcContrast:disparitybuffer outbuffer:contrastbuffer
gridSize:32 numChannels:1];
   for (j = ((contrastbuffer->height - 32) >> 1) *
(contrastbuffer->rowBytes >> 2);
j < (contrastbuffer->height - 64) * (contrastbuffer->rowBytes >> 2);
j += (contrastbuffer->rowBytes >> 2)) {
   for (i = 64; i < contrastbuffer->width - 64; i++)
{ int curVal = data[j + i] + data[j+ i + 16];
   if (curVal < lowestValue)
{ lowestValue = curVal;
lowestLoc = i;
lowestY = j / (contrastbuffer->rowBytes >> 2); }
   if (curVal >= highestValue)
{ highestValue = curVal;
highestLoc = i;
highestY = j / (contrastbuffer->rowBytes >> 2);
} } }
   balancePoints.x = ((lowestLoc << 6) / disparitybuffer->width) + 31;
balancePoints.y = ((highestLoc << 6) / disparitybuffer->width) + 31;
```

Chapter 5

Results and Analysis

5.1 Comparison of Image Pyramid against SSD

The image pyramid was used for a couple of different reasons. The primary reason was to improve results in areas of low contrast, due to the localized nature of the least sum-of-squares differences algorithm. This is especially important since any noise in the image more adversely affects disparity calculation in low contrast regions. Figure 5-1 shows a hallway which contains regions of low contrast like the walls and high contrast regions such as the rug on the floor. Intuition states that the high contrast regions should be resolved accurately by both algorithms. However, when looking at the resulting depth maps, it is apparent that the image pyramid algorithm is able to more provide more accurate results over the entire image. This is likely due to the fact that there are a much greater number of disparity offsets being checked. This allows for this algorithm to correctly handle obstacles that are very near the user.

For the current program, there were up to 96 disparity offsets checked. The image pyramid consisted of 4 levels (320x240, 160x120, 80x60, and 40x40). The discriminator threshold was set to 128 for the image pyramid results, which is a relative value that has been scaled. Any value above this threshold resulted in a disparity value for that level. The discriminator window size was 16 pixels by 16 pixels. For this window, the SSD was calculated for the edge map. For SSD results, the threshold had to be set much lower (approximately 32) to provide a dense depth map. Compare the results of Figure 5-2 with Figure 5-3.

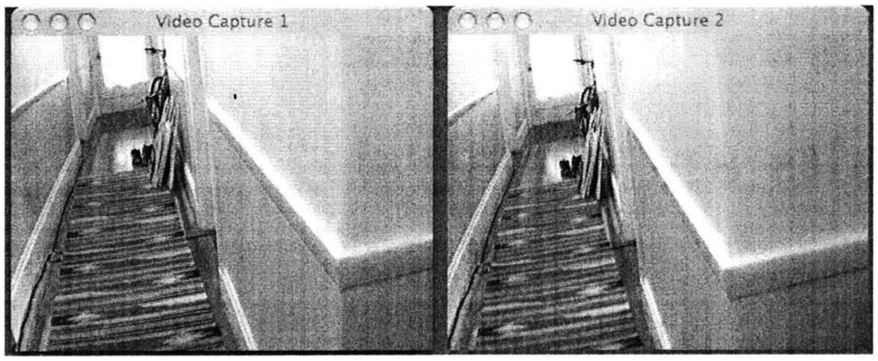

Figure 5-1: Sample input from cameras

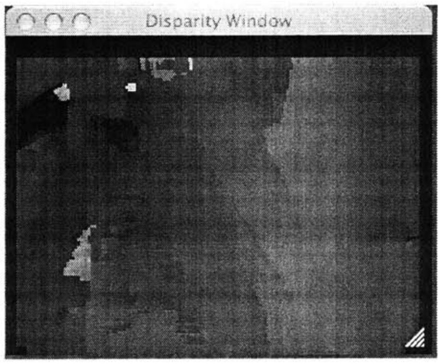

Figure 5-2: Depth map computed with image pyramid

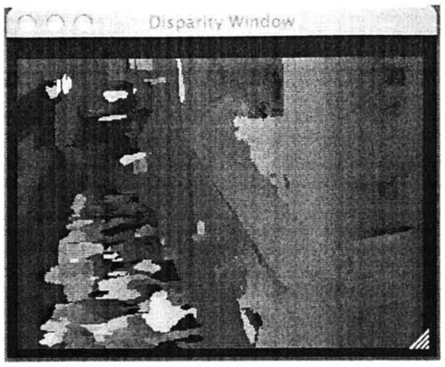

Figure 5-3: Depth map computed without image pyramid using only least sum-of-squares differences calculations

5.2 Noise Reduction

Using the implementation described above, the noise reduction provides an increase in the signal to noise ratio, which can improve the performance of the stereo vision algorithm. Although there is increased blurring due to Note the images have been enhanced to highlight the differences. Compare the results of Figure 5-4 with Figure 5-5.

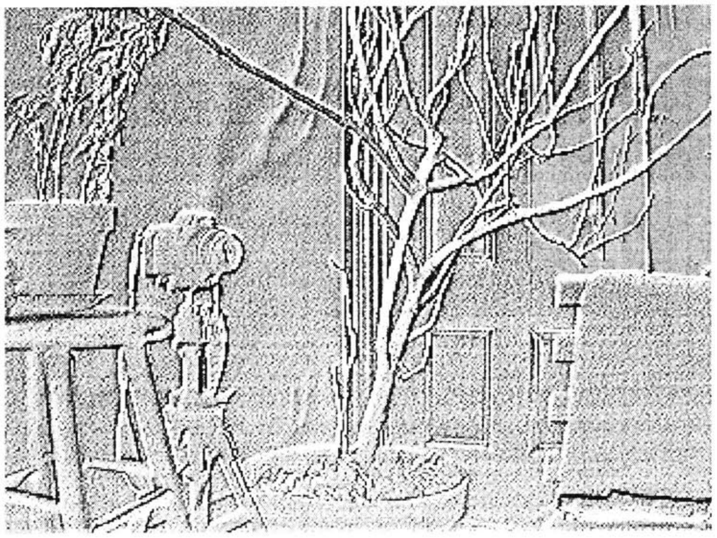

Figure 5-4: Signed edge detection without noise reduction

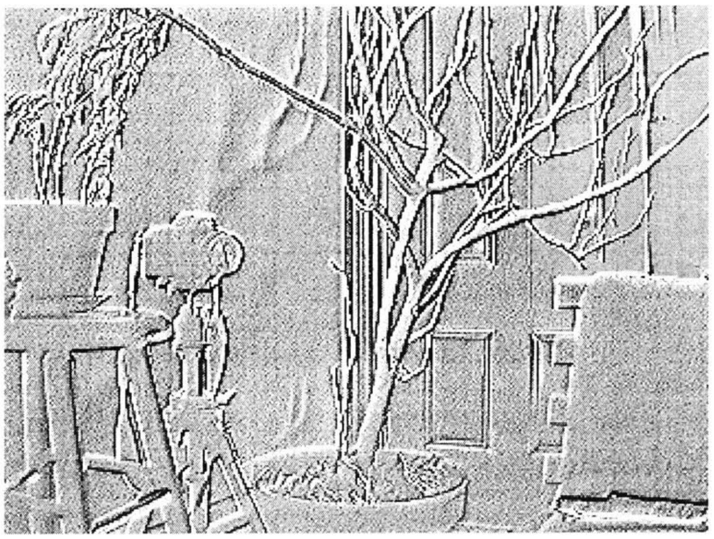

Figure 5-5: Signed edge detection with noise reduction

5.3 Discriminator Results

The two discriminators described in the implementation section are compared. The first version uses the sum of squared edge values for the entire window to determine whether to calculate the depth at the center location. For the second version, the edge value of the pixel itself is taken. This produced more detailed depth maps, but the also increased the amount of noise present. As opposed to the first method, this method is trivial to compute in order to find $(\partial E/\partial x)^2 + (\partial E/\partial y)^2$. Refer to Figures 5-6 and 5-7.

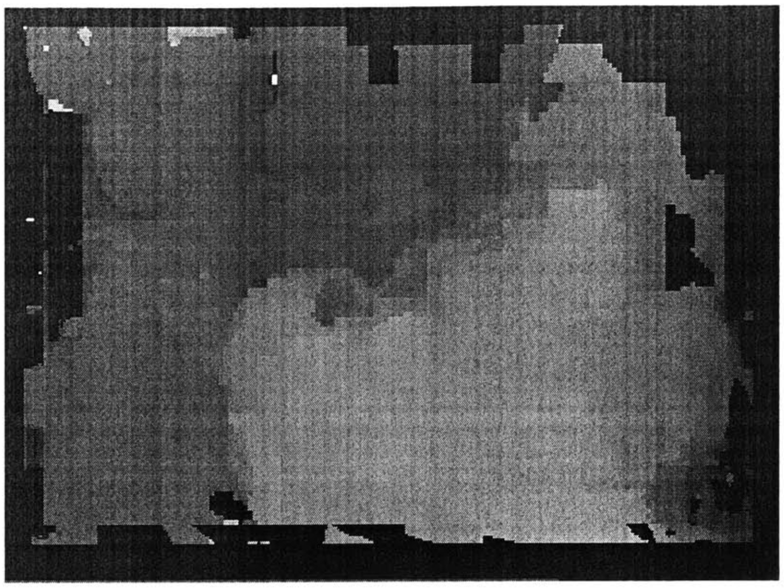

Figure 5-6: Results with discriminator based on sum of squared edges in window

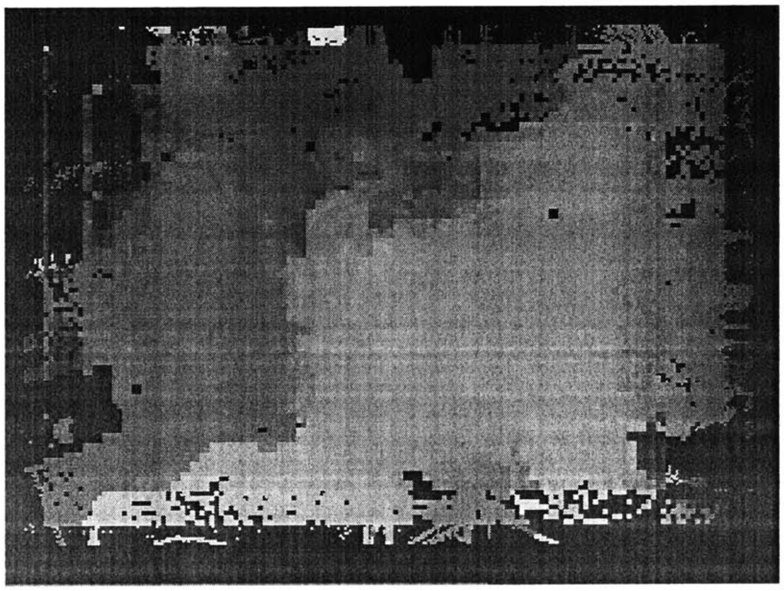

Figure 5-7: Results with discriminator based on edge of current pixel

5.4 Performance Results

The stereo vision system is capable of full 30 frames per second operation at 320 x 240 pixels with up to 128 disparities. Even at the full 640 x 480 pixels and up to 128 disparities, the system can achieve an estimated 10-12 frames per second. At 320 x 240, there is still about 20% of the CPU not being utilized on a PowerBook G4 1.5 GHz.

As a comparison, for the least sum-of-squares differences without using the image pyramid results in much worse performance, close to one frame per second. This is due to the fact that there are 128 offsets being calculated at each pixel location. Since each level of the image pyramid allows for a fourfold reduction in computation, the performance increases dramatically. The higher resolution disparity search is based on the results of the lower resolution depth map, thus allowing a decrease in the total number of disparity searches at each progressive level.

5.5 Aural Feedback

In order to demonstrate the functionality of the program, the same image of the hallway from a previous section is shown below. The triangle represents an obstacle in front of the user, while the circle represents an open path. To the user, the triangle is represented by a popping sound. Since the sound is slightly louder in the right earphone, the sound appears to be from ahead but slightly to the right. The circle represents the best found open path. It is the sound of a piano key and was chosen to be significantly different from the obstacle sound. The sound is slightly louder in the left ear, thus appearing to the user to be coming from ahead and to the right. The two sounds are played back at alternating times in order for the user to more easily identify the two sounds.

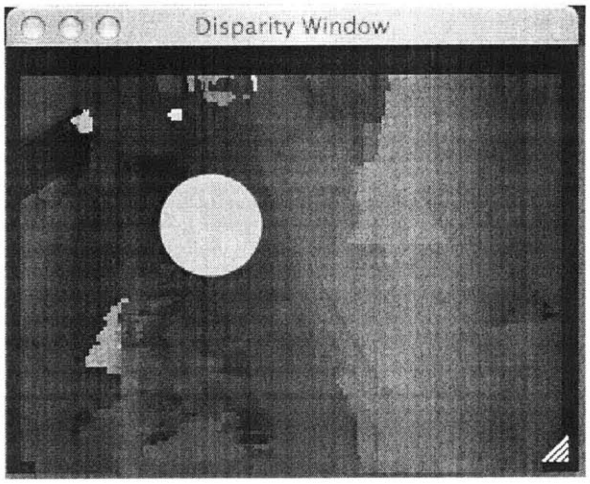

Figure 5-8: Sample aural feedback. Triangles represent obstacles. Circles represent open paths.

Chapter 6

Related Work

6.1 Real-time Stereo Vision

Because of the vast number of calculations required, there have been attempts to implement a stereo vision algorithm in hardware[11]. Because the chips are designed specifically for the task of stereo vision, they are able to achieve very high performance through the use of parallelization. The approach taken in this program also tries to take advantage of parallel processing to improve the calculation throughput of the algorithm and allow for more in-depth processing.

As 3D games have become progressively more advanced, the graphics hardware that these games rely on have also followed a similar path. Current graphics cards are now becoming vastly more programmable, allowing specialized tasks to be performed solely by the graphics processing unit (GPU), freeing the CPU to perform more complex tasks at the same time. While there are still major limitations to using the GPU in this way, a couple stereo vision algorithms have begun to appear that make use of the GPU to provide real-time dense stereo depth maps[12]. An algorithm like this would allow the cpu to allocate more processing power towards more robust post-processing algorithms and audio generation.

6.2 Rapid Object Detection

Recent progress has been made in developing a system to quickly and accurately detect objects[10]. The detector uses machine learning for training, and works by building a cascade of simple haar-like classifiers that satisfy the training data. While the initial work was to perform face detection in images, the detector has been shown to be useful when trained in detecting other objects, such as pedestrians, in real-time.

A system that could detect a number of common objects in real-time or near real-time could be integrated into the program described in this paper. Each object could be assigned a distinct musical instrument. The user would be able to gain an even better understanding of the surrounding area by learning about the location of people and other important objects depending upon their current environment. A set of detectors trained for outdoor objects could provide information about the location of cars, sidewalks, roadways, and buildings. Detectors trained for an office environment may provide feedback on the location of desks, chairs, and doors or doorways.

Chapter 7

Conclusions and Future Work

Using vision methods, a real-time stereo vision system is implemented taking advantage of the parallel processing of the AltiVec execution unit. As SSE is very similar in nature, these ideas could be extended to improve stereo vision performance on Intel chips as well. Since the current PowerPC chip is appropriate for embedded systems with strict power consumption requirements, the system implemented could achieve sufficient performance while still meeting those goals. Many other methods in vision are also suitable for implementation using AltiVec.

In the current program, vision techniques are used to gain a greater understanding of the scene in front of the user. From a stereo image pair, the visual data is analyzed using an image pyramid and least sum-of-squares differences to produce a depth map. Improvements over a naive SSD implementation are demonstrated. The depth map is then transformed into audio signals through further post-processing.

The system described in this paper has applications in assisting blind people to navigate through unfamiliar environments. It could also be used to provide audio feedback when visual feedback is not possible, for example in situations where seeing may be hindered. A possible military application may allow aural aide without requiring the user to wear night-vision goggles.

The program has potential to be expanded in other areas. Rapid object detection may provide a supplementary method of providing information to the user in audio form. Objects that are learned by the system could be reproduced in audio form. Another approach could be to use the depth map to segment individual objects. Rather than have an object to be trained before it can be recognized by the system, each object could give off an audio signature based on their shape or outline. For example, a circle object could be represented by a sine wave. Any deviation in the object's outline would alter the sine wave accordingly. Other visual information, such as color or text contained within each object could also affect the waveform, by adjusting pitch or amplitude. A nonprofit organization is actively working to develop a program that can extract text from images[14]. A feature like this would be very useful to this system.

Bibliography

[1] Y. Boykov, O. Veksler, and R. Zabih. A variable window approach to early vision. IEEE TPAMI, 20(12):12831294, 1998.

[2] Heiko Hirschmller (2001), Improvements in Real-Time Correlation-Based Stereo Vi- sion, in Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision, 9-10 December 2001, Kauai, Hawaii, pp. 141-148.

[3] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. IEEE TPAMI, 23(11):1222-1239, 2001.

[4] O. Veksler. Stereo matching by compact windows via minimum ratio cycle. In ICCV, volume I, pages 540547, 2001.

[5] D. Scharstein and R. Szeliski. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *IJCV*, 47(1/2/3):7-42, April-June 2002. http://www.middlebury.edu/stereo/

[6] *AltiVec Technology Programming Interface Manual*, Freescale Semiconductor, Chandler, AZ, 1999

[7] Berthold K. P. Horn, *Machine Vision*, MIT Press, Cambridge, MA, 1986

[8] http://developer.apple.com/hardware/ve/algorithms.html

[9] http://developer.apple.com/macosx/coreimage.html

[10] P. Viola and M. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. *CVPR*, 2001.

[11] J. Woodfill, G. Gordon, R. Buck, Tyzx DeepSea High Speed Stereo Vision System, In Proceedings of the IEEE Computer Society Workshop on Real Time 3-D Sensors and Their Use, CVPR, (Washington, D.C.), June 2004.

[12] M. Gong and Y.-H. Yang. Near real-time reliable stereo matching using programmable graphics hardware. CVPR 2005

[13] Peter B.L. Meijer , http://www.seeingwithsound.com

[14] Myers, G.K., Bolles, R.C., Luong, Q.T., Herson, J.A., "Recognition of 3-D Scene Text," Fourth Symposium on Document Image Understanding Technology (SDIUT01), Columbia, Maryland, April, 2001, pp. 85-99. http://www.sri.com/esd/automation/video_recog.html

Appendix A

CalcContrast function used by Discriminator

Take the multiply-sum of each set of 4 elements. Since multiply-sum combines the 16 byte values to 4 32-bit values, some permutations of the vector must be made. The first element in the first set is $0x00$ through $0x03$ which becomes represented as a. The second element is $0x01$ through $0x04$, which is represented by b. Thus the first vector from $0x00$ to $0x0F$ becomes $aeim$ after the multiply-sum. The fourth offset can reuse the eim portion that was already calculated by performing further permutation of the data.

A.1 Vector Transformations required for CalcContrast

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10
   02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11
   03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12
   00 - 03, 04 - 07, 08 - 0B, 0C - 0F ...
   01 - 04, 05 - 08, 09 - 0C, 0D -
   a, e, i, m, q, u, y, C
   b, f, j, n, r, v, z, D
   c, g, k, o, s, w, A, E
   d, h, l, p, t, x, B, F

   aeim -> 0
   bfjn -> 1
   cgko -> 2
   dhlp -> 3
   eimq -> 4
   fjnr ... etc...
```

A.2 Inner Loop performed per each row

```
first = vec_ld(0,data);
   second = vec_ld(16,data);

   aeim = vec_msum(first, first, zero);
   total0 = vec_add(aeim, total0);

   temp = vec_sld(first, second, 1);
   bfjn = vec_msum(temp, temp, zero);
   total1 = vec_add(bfjn, total1);

   temp = vec_sld(first, second, 2);
   cgko = vec_msum(temp, temp, zero);
   total2 = vec_add(cgko, total2);

   temp = vec_sld(first, second, 3);
   dhlp = vec_msum(temp, temp, zero); total3 = vec_add(dhlp, total3);

   quyC = vec_msum(second, second, zero);
   total4 = vec_add(vec_sld(aeim, quyC, 4), total4);
   total8 = vec_add(vec_sld(aeim, quyC, 8), total8);
   total12 = vec_add(vec_sld(aeim, quyC, 12), total12);

   temp = vec_sld(second, second, 1);
   rvzD = vec_msum(temp, temp, zero);
   total5 = vec_add(vec_sld(bfjn, rvzD, 4), total5);
   total9 = vec_add(vec_sld(bfjn, rvzD, 8), total9);
   total13 = vec_add(vec_sld(bfjn, rvzD, 12), total13);

   temp = vec_sld(second, second, 2);
   swAE = vec_msum(temp, temp, zero);
   total6 = vec_add(vec_sld(cgko, swAE, 4), total6);
   total10 = vec_add(vec_sld(cgko, swAE, 8), total10);
   total14 = vec_add(vec_sld(cgko, swAE, 12), total14);
```

```
temp = vec_sld(second, second, 3);
txBF = vec_msum(temp, temp, zero);
total7 = vec_add(vec_sld(dhlp, txBF, 4), total7);
total11 = vec_add(vec_sld(dhlp, txBF, 8), total11);
total15 = vec_add(vec_sld(dhlp, txBF, 12), total15);
```

A.3 Code used to get values out to memory

All of the vector elements need to be merged and then stored. *vec_sums* will put each of the values into the last element. Merge Low and Merge High simply merge these last elements of each vector into one vector.

```
vUInt32 templong = vec_mergel(
vec_mergel(
vec_sums(total0, zero),
vec_sums(total2, zero)
),
vec_mergel(
vec_sums(total1, zero),
vec_sums(total3, zero)
)
);

vec_stl(templong, 0, data2 + (j << 2));

templong = vec_mergel(
vec_mergel(
vec_sums(total4, zero),
vec_sums(total6, zero)
),
vec_mergel(
vec_sums(total5, zero),
vec_sums(total7, zero)
)
);

vec_stl(templong, 0, data2 + (j << 2) + 1);

templong = vec_mergel(
vec_mergel(
vec_sums(total8, zero),
vec_sums(total10, zero)
),
vec_mergel(
vec_sums(total9, zero),
vec_sums(total11, zero)
)
);

vec_stl(templong, 0, data2 + (j << 2) + 2);

templong = vec_mergel(
vec_mergel(
vec_sums(total12, zero),
vec_sums(total14, zero)
),
vec_mergel(
vec_sums(total13, zero),
vec_sums(total15, zero)
)
);

vec_stl(templong, 0, data2 + (j << 2) + 3);
```

3389 . 101