12-2006

Moving Object Tracking: Seeking Extensible Solutions

Jeremy Pack
Utah State University

MOVING OBJECT TRACKING: SEEKING EXTENSIBLE SOLUTIONS

by

Jeremy Pack

Thesis submitted in partial fulfillment
of the requirements for the degree

of

HONORS IN UNIVERSITY STUDIES
WITH DEPARTMENTAL HONORS

in

Computer Science
in the Department of Science

Approved:

Thesis/Project Advisor
Dr. Stephen J. Allan

Departmental Honors Advisor
Myra Cook

Director of Honors Program
Dr. Christie Fox

UTAH STATE UNIVERSITY
Logan, UT

Fall 2006

Moving Object Tracking: Seeking Extensible Solutions

Jeremy Pack

January 10, 2007

Abstract

Moving object tracking is a difficult field with no "best" solution. The documents contained here detail software that was developed in order to perform tracking of objects that are in the line of sight of multiple cameras. Some of the software developed is already in use by the US Army Dugway Proving Ground. Other software represents prototype or early development code. This software was written by Jeremy Pack, who was the lead programmer, and Luke Andrew at the Space Dynamics Laboratory in Logan, Utah.

Description of Contents

This document is separated into five sections:

1. Overview of Surveillance Image Processing Software
2. Cloud Edge Analysis user manual
3. Cloud Volume Reconstruction user manual
4. SIP RT development
5. Extension plugin library proposal

Author's Biography

Jeremy Pack was born in Logan, Utah in 1983, but his family lived in six different states as he was growing up. He was accepted to Utah State University as a University Club Scholar where he received a BA in Computer Science and a BS in Computational Mathematics, with a minor in Russian. He spent two years in Russia as a missionary for the LDS Church.

As of 2007, he and his wife Julianne live near Seattle, Washington with their son Thomas where he works for Google, Inc. He enjoys swimming, playing the violin, reading and eating the free food provided by his employer.

Overview of the Surveillance Image Processing Software

Jeremy Pack

January 10, 2007

1 Original Purpose

At Dugway Proving Ground, chemical releases are generated for the purpose of simulating clouds of dangerous chemicals. DPG uses software to track these releases based on various instruments such as cameras or lasers. They required a new suite of software to analyze the generated data that was to run on the Windows platform and give better, more efficient results than their current software without requiring as much work from human operators.

2 Initial Implementation

Seth Call designed the original SIP program and delivered it to Dugway. At the time, it tracked objects in two dimensions. The initial implementations he designed culminated in a graphical program that was capable of processing Dugway IR images as well as color images of various types and IR images from SDL IR cameras.

This software used various image processing techniques to clean up the imagery and isolate the objects that had moved in relation to a software selected compare image. The user was presented with a large number of options that could be used to modify the processing algorithms for use with different types of imagery. As such, the software could handle images with many types of noise, gain changes or other flaws.

The end result was an output of the edge information for each moving object in each image in 2D. No 3D information was generated.

3 Cloud Edge Analysis

In 2005 Seth Call left the project and it was taken over by Jeremy Pack. Meanwhile, the Dugway Proving Ground acquired a new infrared camera capable of running at over 100 frames per second. The software did not, at the time, run in real time, but the sheer volume of data generated by the new camera overwhelmed the SIP program within as little as 20 seconds of data. As Dugway required much more data than this to be processed, this was unacceptable. As such, a new version of SIP called Cloud Edge Analysis (CEA) was written from the ground up focusing on optimization for speed and memory use. It achieved the following:

1. A speed increase of over 200% (i.e. images which originally processed at 45 frames per second now processed at 150 frames per second on the same machine).
2. Memory savings making it possible to handle multiple orders of magnitude more frames than the previous software

The user interface was also redesigned to more closely meet the requirements at DPG.

It was written using C++, with the MFC, ATL, OpenGL and Boost libraries.

4 Cloud Volume Reconstruction

In mid 2005, work was begun on software to reconstruct 3D objects based on the output of the CEA software as the other half of the SIP software suite. Dugway provided the initial algorithms for this work, which were based on algorithms from prior software that they had used. An initial prototype was developed using these algorithms for 3D reconstruction, and inaccuracies of the algorithms were noticed during testing. This is described in the next section.

CVR presents a 3D user interface in which the operator can move through space to view the moving objects from different angles. When given the output of CEA is input, it generates these objects and they are displayed as they move. It is also possible to go to any point in time and view the moving objects generated at that time.

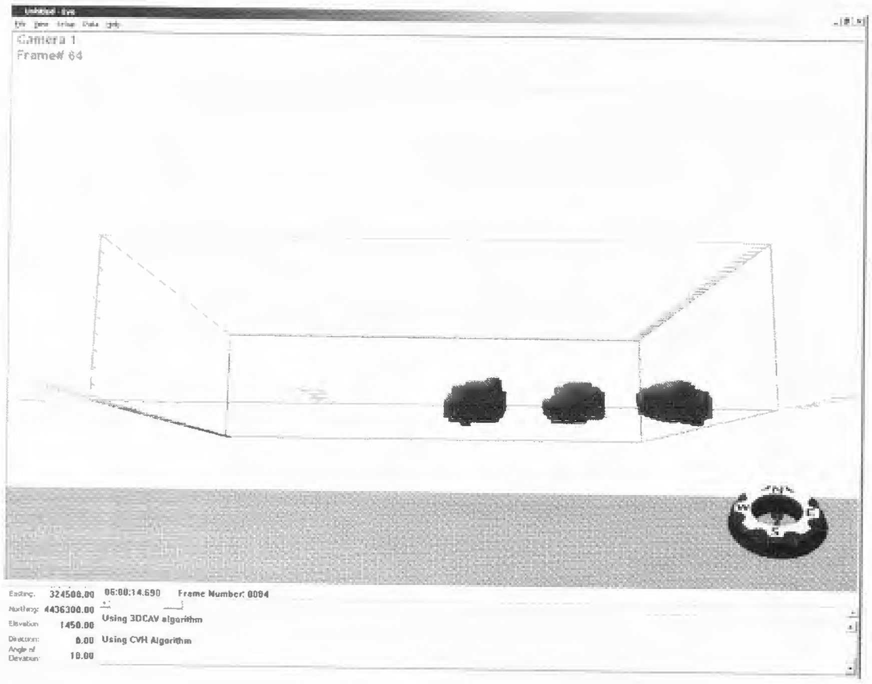
The speed of CVR naturally depends on the accuracy desired by the user. Doubling the number of voxels (3D pixels) in each dimension naturally increases the number of total voxels by a factor of eight, and thereby also increasing the processing time by a similar factor. With array dimensions of 50X50X50, speeds of 30 frames per second could usually be achieved (depending on the computer used). This would translate to 3.75 million voxels per second. This is much faster than Dugway's older software, and, as explained in the next section, more accurate.

5 CVR Algorithm Redesign

When work was begun on CVR, DPG provided the source code of 3DCAV, as well as some papers describing the major algorithms involved. Some of these algorithms were initially tested in the CVR software, but have since been replaced because of serious flaws in their design. The Dugway versions of the algorithms are still accessible in CVR by pressing the 'D' key, and though they give the correct results in certain special cases, in other cases the results are completely inaccurate. Some of the reasons for this inaccuracy follow:

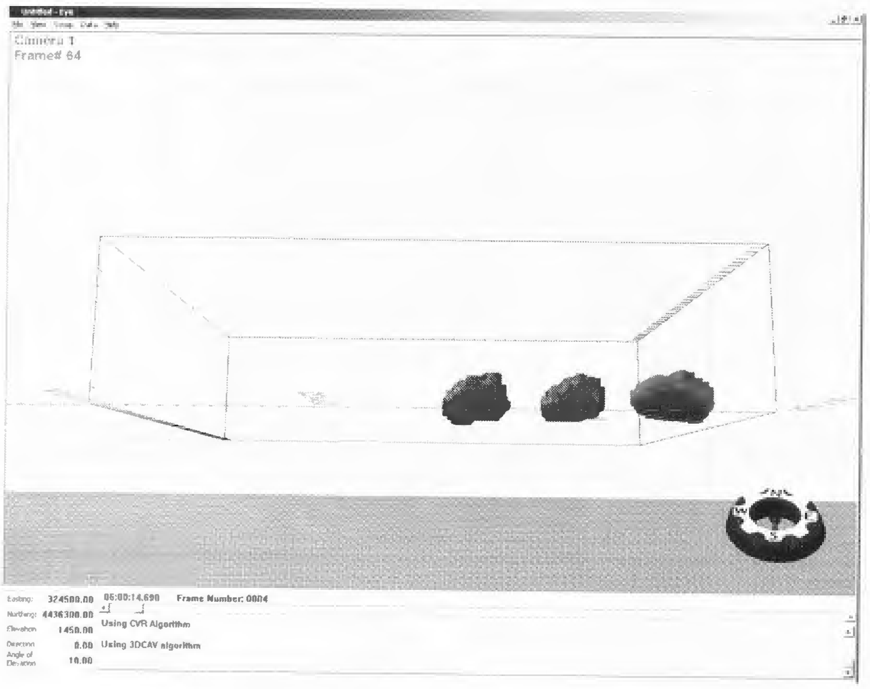
5.1 Small Angle Approximations

In a number of places in the software, Dugway's original programmers took advantage of the small angle approximation, which states that as $\theta \to 0, sin(\theta) \to \theta$. The primary function used to determine whether a certain point is seen by a given camera makes use of this. The angle from the horizon is basically ignored in the equation. This is perfectly correct when $\theta = 0$. But as θmoves away from 0, this estimate loses accuracy. Commonly, Dugway cameras are positioned at about 4 degrees up from the horizontal. At this angle, the small angle approximation does not result in noticeable error. However, in the event that cameras would need to be pointed at more extreme angles, the difference would become more apparent. In the figures below, the red represents what the camera actually sees. The grey represents the cloud generated by the Dugway and SDL algorithms respectively. As the angle increases, some error begins to appear when using the small angle approximation. SDL's algorithm does a full three-dimensional rotation, allowing for any combination of direction, angle or tilt of the camera.



```
Easting:     324500.00   06:00:14.690   Frame Number: 0084
Northing:   4436300.00
Elevation:     1450.00   Using 3DCAV algorithm
Direction:        0.00   Using CVH Algorithm
Angle of
Elevation:       10.00
```

Figure 1: CVR Algorithm at 10 degree angle



Easting: 324500.00 05:00:14.690 Frame Number: 0004
Northing: 4436300.00
Elevation 1450.00 Using CVR Algorithm
Direction 0.00 Using 3DCAV algorithm
Angle of Elevation 10.00

Figure 2: 3DCAV Algorithm at 10 degree angle

5.2 Integer Rounding

Using the new image projection capability of CVR, it became apparent that there were small errors in the line drawing algorithm. This algorithm was the same that had originally been used in 3DCAV. The error was usually quite slight, but became more apparent when the "accuracy" was too large. An algorithm was found that yielded more accurate results.

There were two reasons for needing a change:

5.2.1 Correct Float Conversion

In many programming languages (including C++), when floats are converted to integers, they are rounded towards zero. Thus, .8 is rounded to 0, 54.1 is rounded to 54, and 54.999999 is rounded to 54. Also, -.8 rounds to 0, as does -.1.

To round floating point numbers in the standard way in C++, (i.e. rounding .5 yields 1, rounding .4 yields 0, rounding -.4 yields 0 and rounding -.8 yields -1) .5 must be added to positive numbers, and subtracted from negative numbers. This was not done. In some cases, it can be assumed that this would not cause a major problem. However, it leads to incorrect cloud locations. The error is proportional to the accuracy. Perhaps the primary danger of this error is that it is most noticeable under the same conditions as the error of the small angle approximation: when the angle of the camera away from the horizon is not sufficiently close to zero.

This error is lessened as the accuracy becomes smaller.

5.2.2 Modern Processors

On older processors, floating point operations are usually much slower than integer operations. On newer processors, only division is significantly slower. This algorithm can be written with three floating-point-to-integer conversions and three floating point additions per voxel, as opposed to the three integer additions, three integer multiplications, and an integer division that are required by the Dugway algorithm.

When this algorithm was designed, the method used was probably much faster than the alternative. Now, however, there is probably no longer a need, especially given the fact that some error is introduced.

6 CVR Future Design

A new method of object reconstruction has been designed which will make it possible to spread the processing load among multiple machines, increasing the speed with which data can be processed. CVR becomes the real bottleneck in the processing pipeline, since CEA is quite fast. The optimal accuracy for CVR cannot be reached with any reasonable processing speed currently (i.e. making it any more accurate than the optimal accuracy would in no way give better results, similar to adding 5.2 and 3.5 and displaying the result as 8.800000 rather than 8.8).

The future design involves a single computer used for display, one computer used to direct the other computers in processing and divide up tasks, and any number of computers, each in charge of a certain region in space.

The 3D processing algorithms would need to be changed in a few fundamental ways in order to make this possible, and those ways have been decided but not yet implemented.

Space Dynamics
LABORATORY
Utah State University Research Foundation

1695 North Research Park Way • North Logan, Utah 84341
Phone 435.797.4600 • Fax 435.797.4495
www.spacedynamics.org

Cloud Volume Reconstruction

Training Manual Prepared For:
Dugway Proving Ground

Revised: August 31, 2006

I. Table of Contents

II. Purpose

This manual serves as an introductory tutorial for the beginning operator to become familiar with the Cloud Volume Reconstruction program and image processing features. Cloud Edge Analysis is a program designed to detect the motion of vapors, from IR or color cameras, analyze specific spectral and geometric information and output the resulting data into Cloud Volume Reconstruction, a three-dimensional view viewing program.

III. Background of Surveillance Image Processing Software

The Cloud Volume Recognition Software (CVR) was developed at the USU Space Dynamics Laboratory to process data from cloud releases at Dugway Proving Ground (DPG). The purpose of this software is to make possible near-real-time on site analysis of the IR images generated during these tests and also to perform post test analysis of the results. Cloud edges are generated by processing sequences of images from IR, bmp, or jpeg files by the Cloud Edge Analysis (CEA) program. The resulting cloud information from each camera can then be used by this program, CVR, to form a 3D representation of the cloud.

During the next phase of this project, work will be done to stream data directly from the cameras to the software (running on computers attached to each camera). These data will be processed immediately and then sent over a network (probably wirelessly) to a central computer which will construct a 3D representation of the cloud. The software is capable of performing these calculations as fast as the cameras can generate the information (over 100 frames per second) on a relatively fast computer, thus providing those present at the test to immediately be able to determine the location and movement of the cloud, and be able to analyze these data immediately after the test is completed.

The software is appropriate for use with the data generated by Dugway Proving Ground as a set of post-test analysis tools. This documentation has been provided to help meet the needs of Dugway Proving Ground to help users to become familiar with the software.

This version of the software was written by Jeremy Pack and Luke Andrew. Much of the functionality of the CEA Program is based on SIP II by Seth Call. The Cloud Volume Reconstruction program is similar in purpose to the 3DCAV program written at Dugway Proving Ground. CEA includes optional use of an algorithm from the TRACE program, also developed by Dugway Proving Ground.

IV. System Requirements

For high speed processing, the following minimum requirements are strongly recommended:

Windows XP Professional

512 MB RAM (The program does not usually use much RAM itself, but for various reasons more RAM is beneficial)

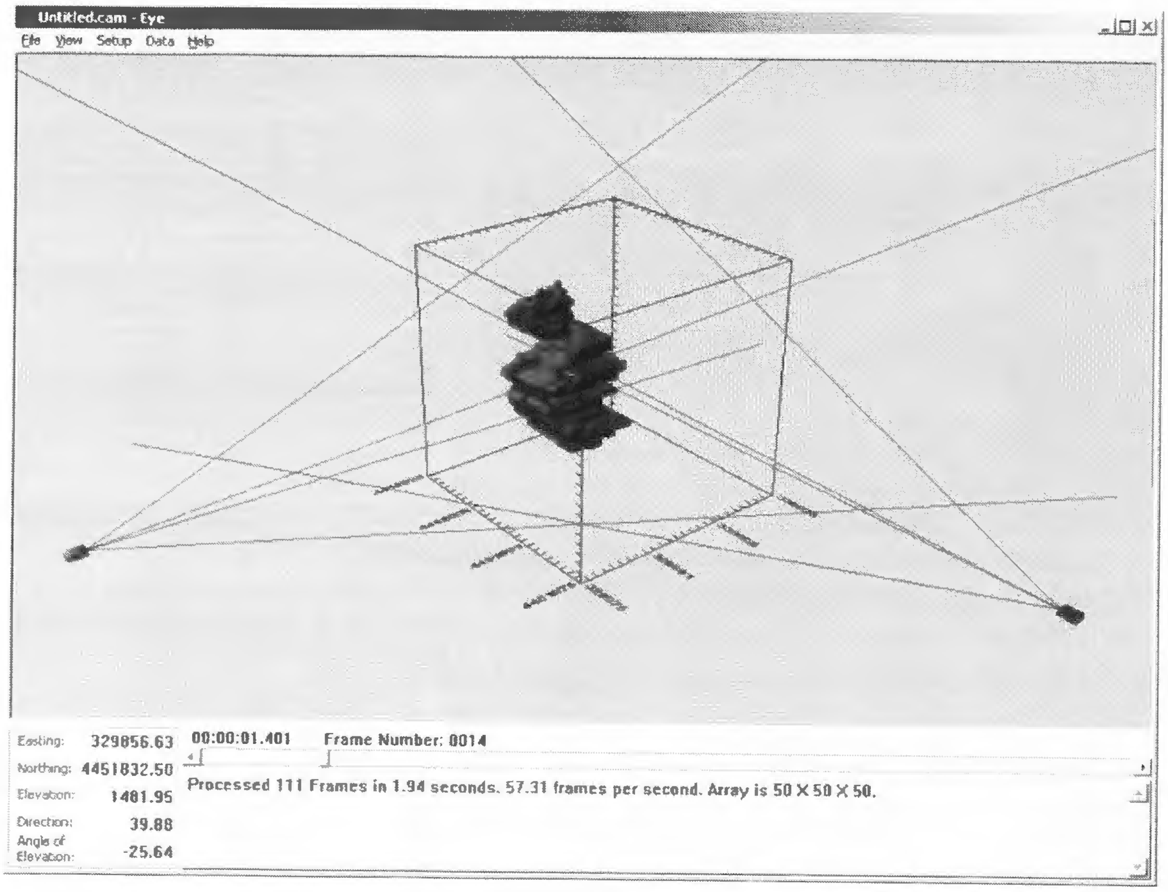
Pentium IV, Athlon, or equivalent processor with speed of at least 2 GHZ.
In addition, it can benefit from multiple processors or hyperthreading.

CVR relies heavily on the graphics card. For fast processing, a graphics card capable of accelerating OpenGL 1.3 or higher is recommended. For advanced functions, the card must be capable of accelerating OpenGL 2.0.

Note:
Many computers do not have the latest OpenGL drivers from the graphics card manufacturer. It is recommended to visit the graphics card manufacturer's website and download the latest driver for the card before using this software. It will run with unaccelerated graphics, but performance will be severely hindered. In addition, requirements are similar to those for CEA. Currently, this program will not benefit much from multiple processors or hyperthreading. This is planned for the future however.

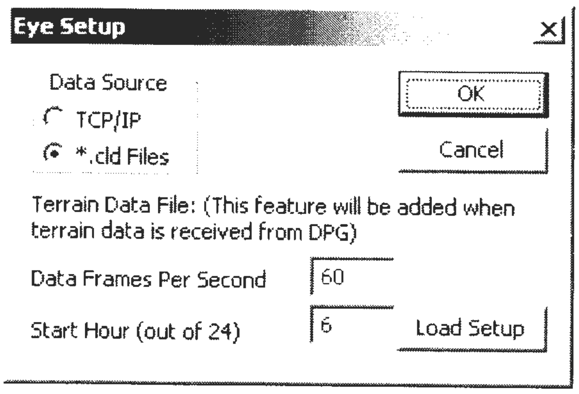
V. Cloud Volume Reconstruction Quick Start

After creating and exporting an image set from the Cloud Edge Analysis program for each camera view, the user is now ready to view the resulting cloud data. Cloud Volume Reconstruction enables the user to combine each camera view into a thee-dimensional (3D) representation of the clouds. An example of the output generated through this process is shown below. Two image sets were processed in CEA, and the data was then sent to CVR to reconstruct the 3D image.



```
Easting:     329856.63   00:00:01.401     Frame Number: 0014
Northing:   4451832.50
Elevation:     1481.95   Processed 111 Frames in 1.94 seconds. 57.31 frames per second. Array is 50 X 50 X 50.
Direction:       39.88
Angle of
Elevation:      -25.64
```

To get started, use the CEA Quick Start Guide, in the CEA manual to generate one or more *.cld files. Then start the CVR program. The first window that will open is the following Setup Dialog.



Data Source
( ) TCP/IP
(•) *.cld Files

Terrain Data File: (This feature will be added when terrain data is received from DPG)

Data Frames Per Second: 60

Start Hour (out of 24): 6

A. Setup Dialog

The options associated with this window are described below.

1. Data Source

Select *.cld. TCP/IP is currently disabled.

2. Data Frames Per Second

The default value is 60, which means that if the camera data used as input was generated at 120 frames per second, then every other frame will be skipped. If the camera data used as input was generated at 30 frames per second, every frame will be repeated.

For Dugway *.ptw files, 60 to 110 frames per second is appropriate (Since they are recorded at about 108 frames per second). "Start hour" should be selected to be some hour that was not passed during the test. This is to compensate for tests running at midnight.

3. Load Setup

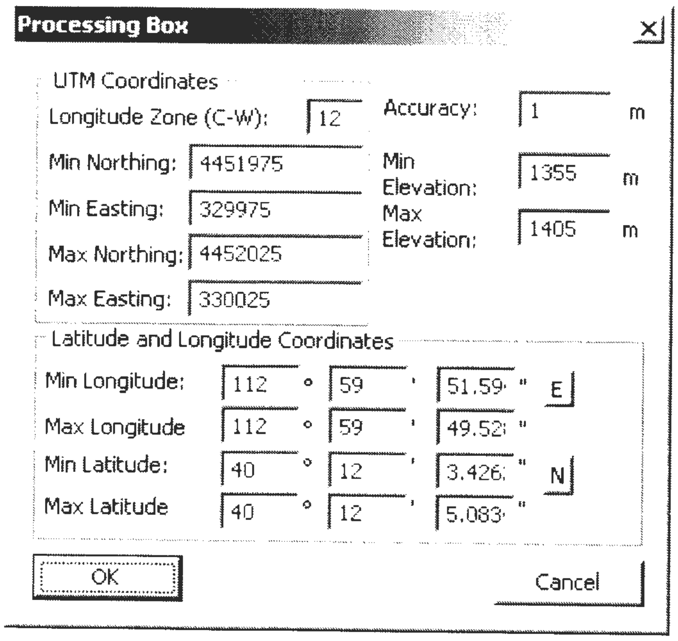
This button allows a previous setup to be loaded. This is useful if there is a basic setup that is the same for many tests, or to review previously processed data.

4. Apply Values

Press "OK".

B. Processing Region

The next dialog box to appear is the processing box dialog. This dialog box is used to determine what region in space is processed by the program. This menu is shown in the following image.



Processing Box

UTM Coordinates

Longitude Zone (C-W):	12	Accuracy:	1	m
Min Northing:	4451975	Min Elevation:	1355	m
Min Easting:	329975	Max Elevation:	1405	m
Max Northing:	4452025			
Max Easting:	330025			

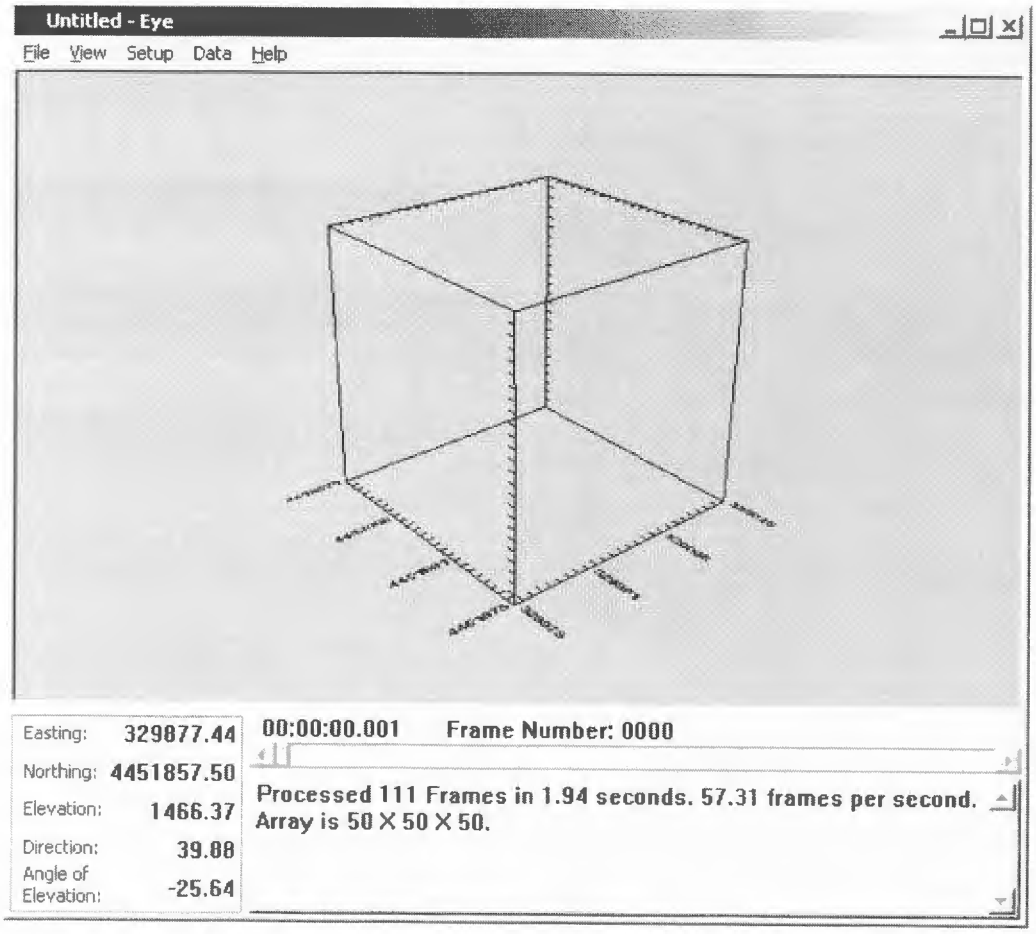
Latitude and Longitude Coordinates

	°	'	"	
Min Longitude:	112	59	51.59	E
Max Longitude	112	59	49.52	
Min Latitude:	40	12	3.426	N
Max Latitude	40	12	5.083	

The Processing Box, or Processing Region, describes the extent of CVR for processing the data. Coordinates can be input either as latitude and longitude or UTM Northing and Easting (WGS84). All measurements are in meters or degrees. The default location is near Dugway Proving Ground. The processing box window can also be opened by clicking on "Setup", "Processing Box".

For using DPG survey data, please see section VI.

Accuracy determines how many voxels (3D pixels) will be considered in the processing region. An accuracy of 4 would mean that each 4X4X4 meter region would be processed as a block. The lower this number is, the slower processing will be. For normal processing speeds, it is recommended to keep the accuracy around 1/100 of the length of the largest side. Press "OK". Now the window appears with the 3D display as shown below. The processing region is the volume enclosed by green lines.



Easting: 329877.44
Northing: 4451857.50
Elevation: 1466.37
Direction: 39.88
Angle of Elevation: -25.64

00:00:00.001 Frame Number: 0000

```
Processed 111 Frames in 1.94 seconds. 57.31 frames per second.
Array is 50 X 50 X 50.
```

C. Shortcut Keys

The following keys can be used for navigation through the scene.

- North, South, East, West - Use the first letter of the word (N, S, E, W).
- Change the viewing direction - Use the arrow keys.
- Move forward, backward, left, right - Use 2,4,6,8 on the number pad. Also use the mouse scroll wheel to zoom in and out.
- Up - Plus key.
- Down - Minus key.
- Toggle Camera Views – C (view from camera) or alt-C (view from right behind camera)
    - Note that when viewing from the camera, the 2D edge outline that was generated by CEA is projected. Some camera information is also displayed in the upper left corner.
- Toggle Object Views – V
- Rotate the Processing Box - O and P
- Stop Moving – spacebar
- View from the top – T
- Default view – H
- Iso 3D view – I
- Start Processing – The tilda key (~)
- Stop Processing – The escape key (Esc)

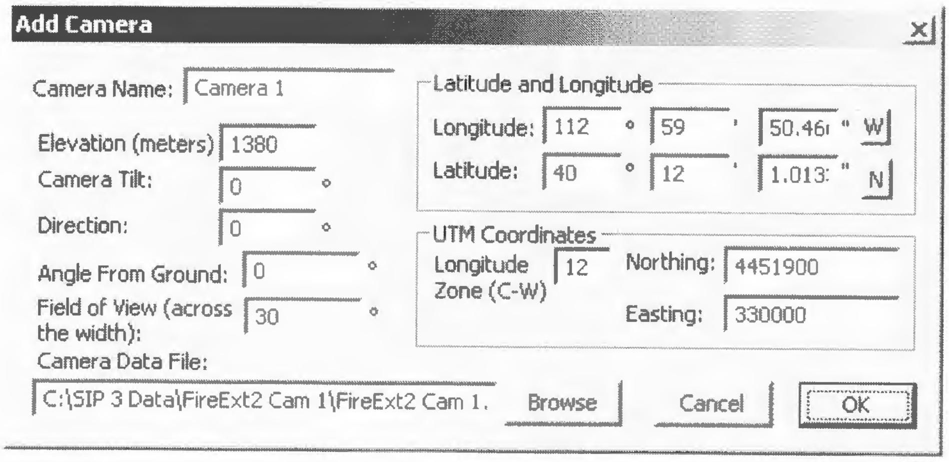
- View from Camera 1-10 – Use the appropriate number key ('0' = 10)
- View from Obect 1-10 – Use Alt + the appropriate number key ('0' = 10)

D. Camera Setup

Add two cameras to the setup using the previously created camera .cld data files by completing the following steps.

- Click on "Setup", "Camera Setup".
    - Click "Add Camera". Change only the following:
    - Click "Browse" to find a *.cld file to use for input.
    - Change the Northing from 4452000 to 4451900 (to move it to the south of the processing region, which is centered at 4452000 north).

This camera window should be similar to that shown below.



Add Camera

Camera Name:	Camera 1
Elevation (meters)	1380
Camera Tilt:	0 °
Direction:	0 °
Angle From Ground:	0 °
Field of View (across the width):	30 °
Camera Data File:	C:\SIP 3 Data\FireExt2 Cam 1\FireExt2 Cam 1.

Latitude and Longitude

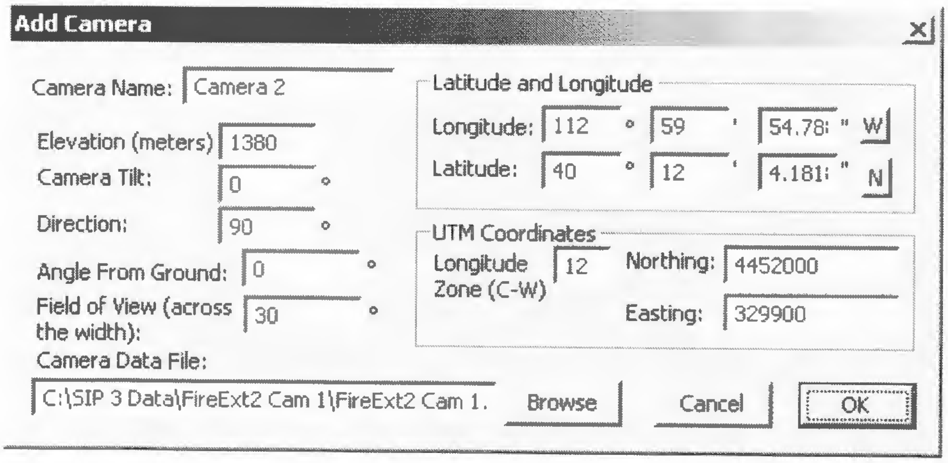
Longitude:	112 °	59 '	50.461 "	W
Latitude:	40 °	12 '	1.013 "	N

UTM Coordinates

Longitude Zone (C-W)	12
Northing:	4451900
Easting:	330000

- Press "OK", then add another camera by again using the "Add Camera" button. This time, change the following:
    - Click Browse to find a *.cld file. This file must either have the same timestamps, or be the same file as for the previous file (if it is the same file, this won't of course be a valid cloud reconstruction, but this Quick Start Guide is intended only to show the features and functionality of CVR).
    - Change the Easting to 329900.
    - Change the direction to 90 degrees and click "OK".

This camera window should now look like the following figure.


Add Camera

Camera Name:	Camera 2
Elevation (meters)	1380
Camera Tilt:	0 °
Direction:	90 °
Angle From Ground:	0 °
Field of View (across the width):	30 °
Camera Data File:	C:\SIP 3 Data\FireExt2 Cam 1\FireExt2 Cam 1.

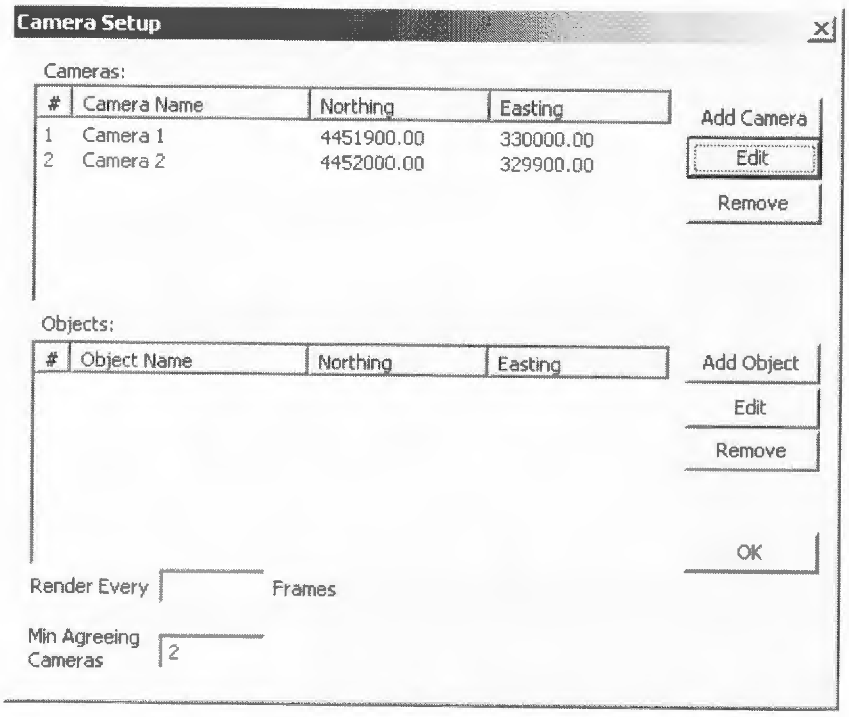
Latitude and Longitude

Longitude:	112 °	59 '	54.78 "	W
Latitude:	40 °	12 '	4.181 "	N

UTM Coordinates

Longitude Zone (C-W)	12
Northing:	4452000
Easting:	329900

An image of the Camera Setup window is shown below with appropriate values for this example.


Camera Setup

Cameras:

#	Camera Name	Northing	Easting
1	Camera 1	4451900.00	330000.00
2	Camera 2	4452000.00	329900.00

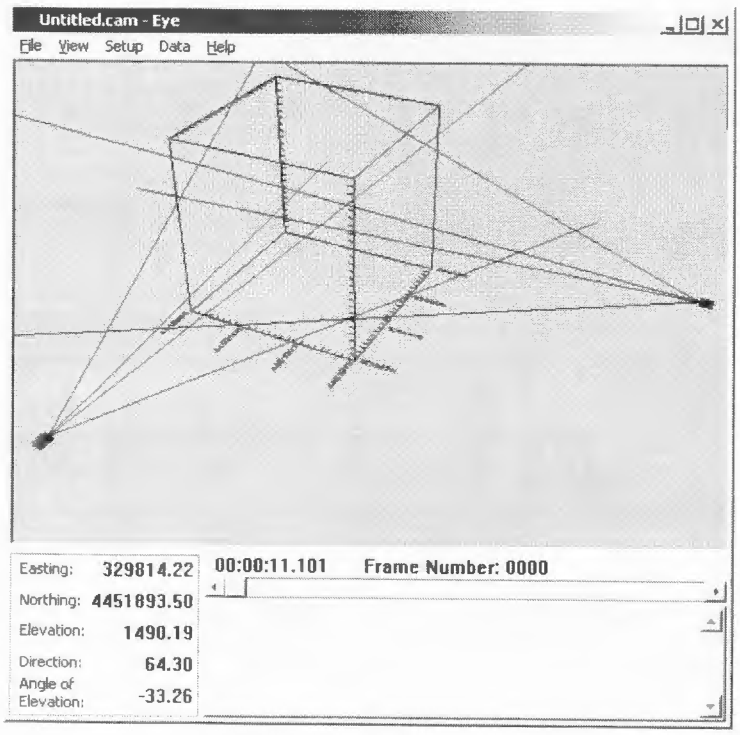
Objects:

#	Object Name	Northing	Easting

Render Every Frames

Min Agreeing Cameras: 2

Now, exit the camera setup menu by clicking “OK”. There should be two cameras pointing towards the processing region, one from the south and one from the west. These cameras are shown in the following image.
(Note that the above image uses the older Camera Setup Dialog. The steps and basic appearance are the same with the updated dialog.)



Easting: 329814.22
Northing: 4451893.50
Elevation: 1490.19
Direction: 64.30
Angle of Elevation: -33.26

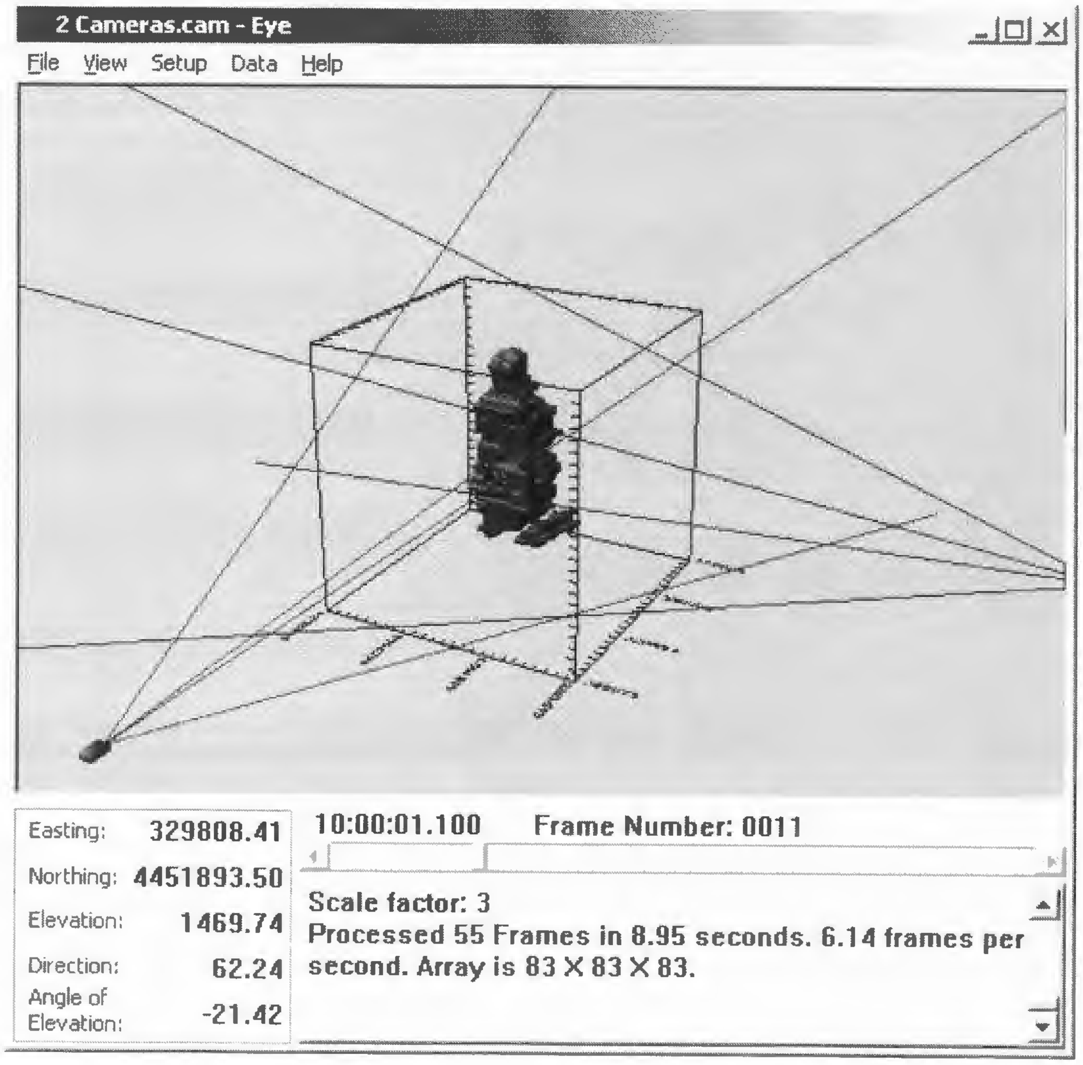
00:00:11.101 Frame Number: 0000

E. Processing Data

Now, select "Data", "Process Data" from the menu. If no clouds appear, check the following settings:

- Set both cameras to use the same *.cld file as input. Changing the camera settings in this way will work if the problem is that the two *.cld files refer to two image sets that do not overlap in time.
- Make sure that both cameras point into the processing region. If one does not, make sure it is not too far from the processing region. Also, check the "Direction" setting by clicking on "Setup", "Camera Setup" and selecting the camera of interest and clicking "Edit".

If the settings are correctly set, an image will be created from the combination of the two cameras. If the same .cld file is set for both cameras, the image will be symmetric, as shown below.

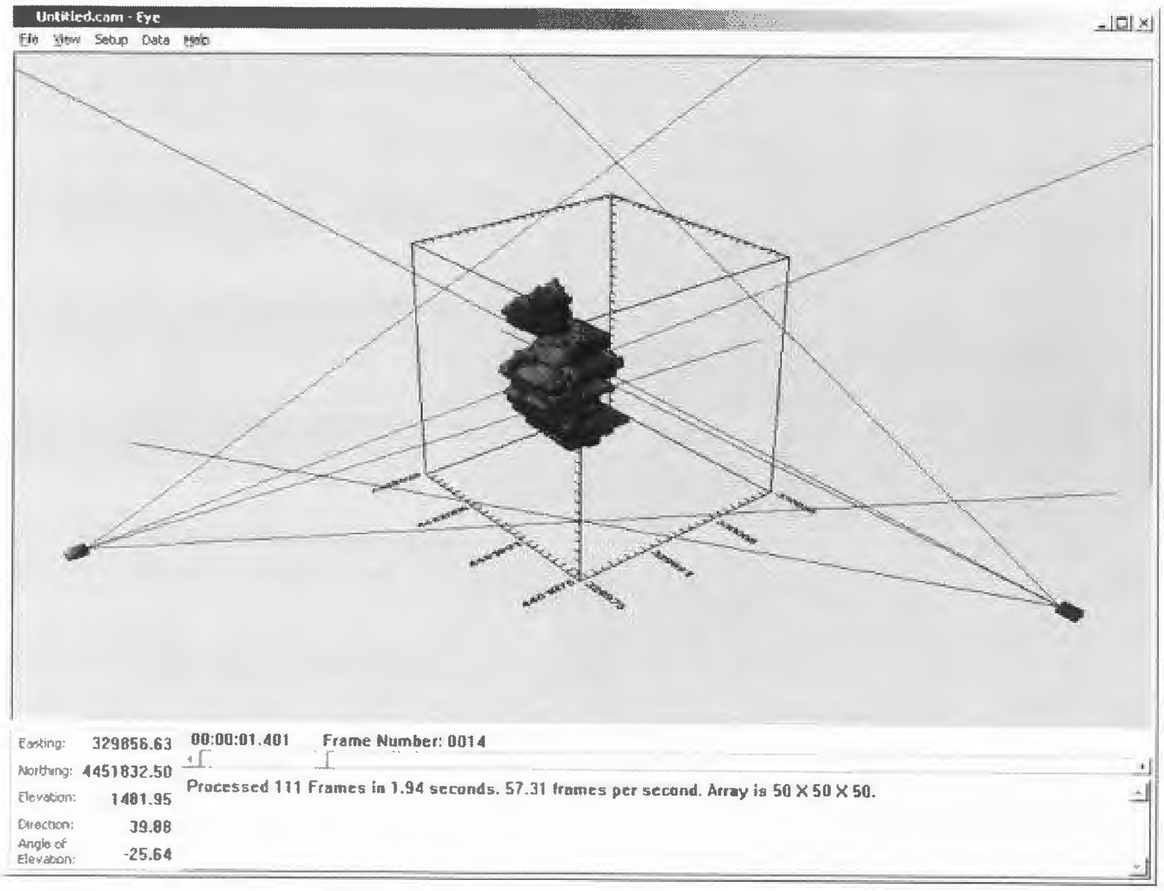


Easting: 329808.41
Northing: 4451893.50
Elevation: 1469.74
Direction: 62.24
Angle of Elevation: -21.42

10:00:01.100 Frame Number: 0011

Scale factor: 3
Processed 55 Frames in 8.95 seconds. 6.14 frames per second. Array is 83 X 83 X 83.

Another example of cloud imagery is shown in the following figure.



Easting: 329856.63
Northing: 4451832.50
Elevation: 1481.95
Direction: 39.88
Angle of Elevation: -25.64

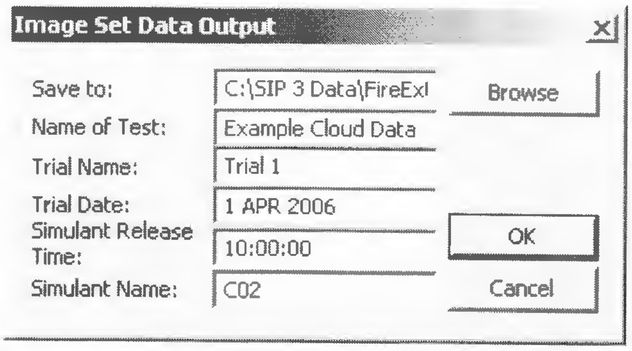
00:00:01.401 Frame Number: 0014

Processed 111 Frames in 1.94 seconds. 57.31 frames per second. Array is 50 X 50 X 50.

Congratulations. You have successfully reconstructed the cloud image in 3D.

F. Exporting Data

To output numerical data for the corresponding image, click "Data", "Export ASCII Data", "Comma Separated ASCII". This will open up the following window, which is used to describe the cloud data, which can be populated similar to that shown below.

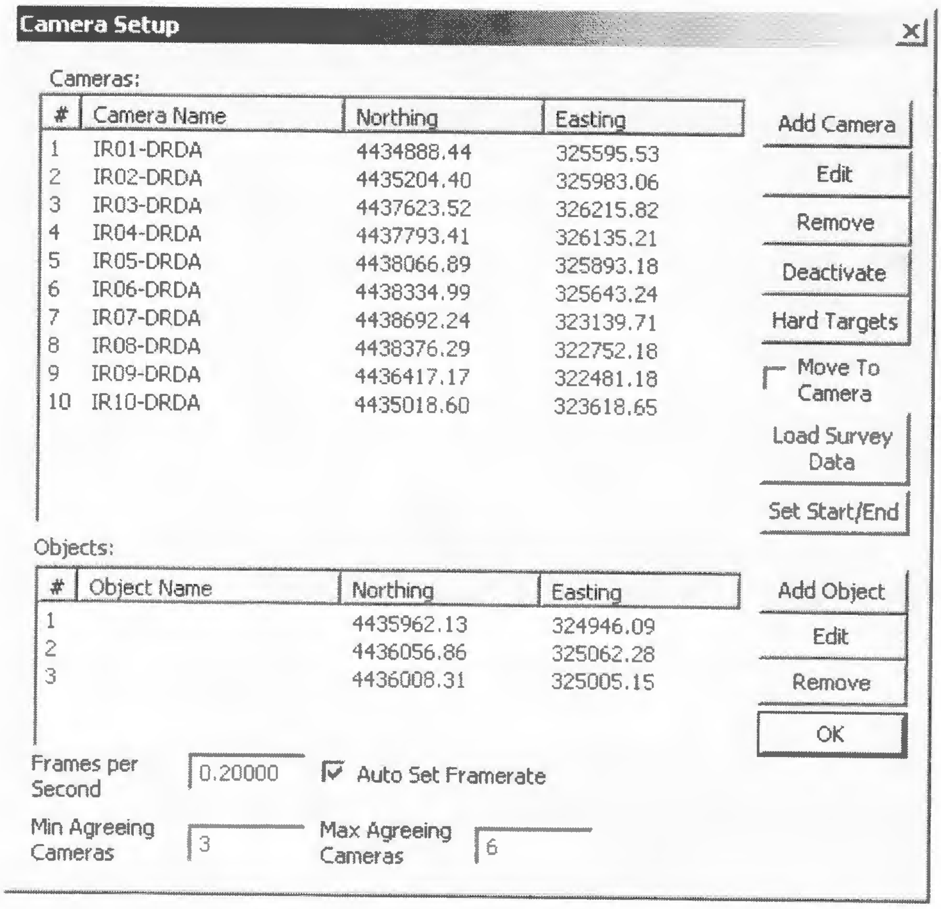


Save to:	C:\SIP 3 Data\FireExl
Name of Test:	Example Cloud Data
Trial Name:	Trial 1
Trial Date:	1 APR 2006
Simulant Release Time:	10:00:00
Simulant Name:	C02

A comma delimited text file is now available in the directory set in the previous window.

VI. DPG Survey Data Quick Start

Processing is simpler when Dugway Proving Ground survey data spreadsheet files are available. In this case, it is possible to automatically input camera position information. If CEA was used to generate the coordinates of the hard targets, then the camera orientation is automatically calculated.



Cameras:

#	Camera Name	Northing	Easting
1	IR01-DRDA	4434888.44	325595.53
2	IR02-DRDA	4435204.40	325983.06
3	IR03-DRDA	4437623.52	326215.82
4	IR04-DRDA	4437793.41	326135.21
5	IR05-DRDA	4438066.89	325893.18
6	IR06-DRDA	4438334.99	325643.24
7	IR07-DRDA	4438692.24	323139.71
8	IR08-DRDA	4438376.29	322752.18
9	IR09-DRDA	4436417.17	322481.18
10	IR10-DRDA	4435018.60	323618.65

Objects:

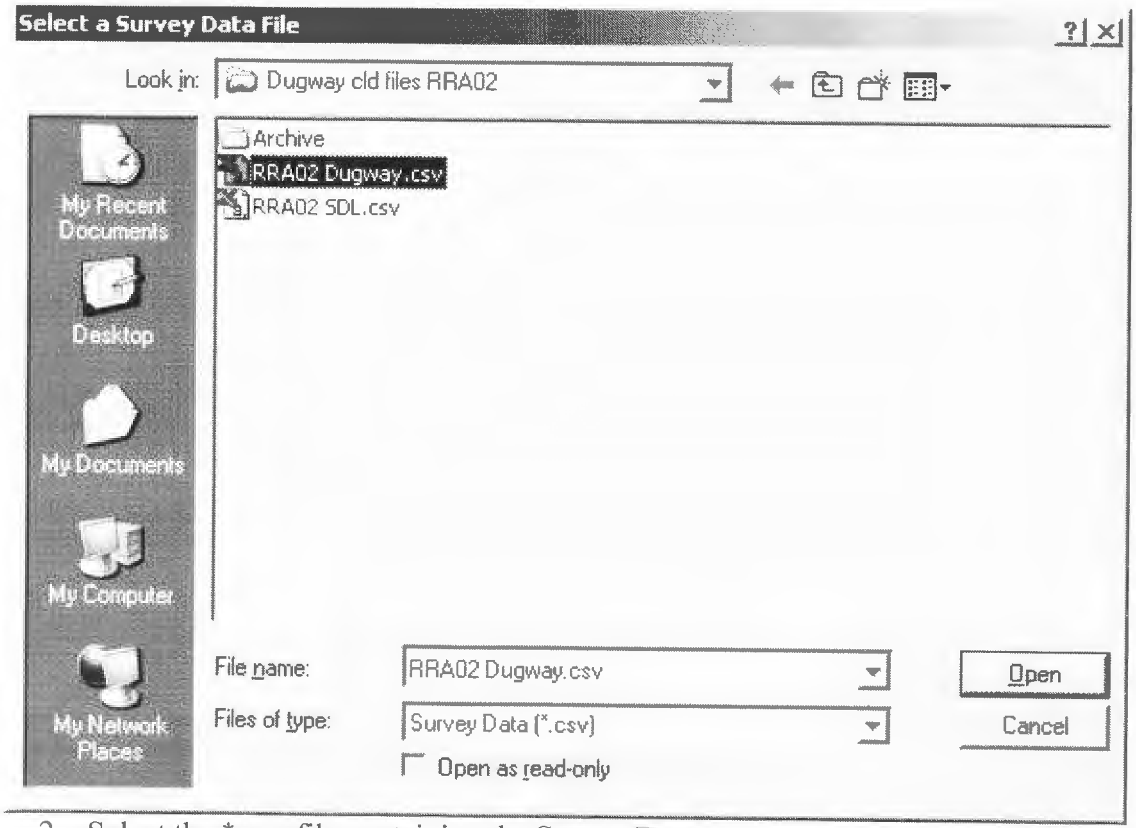
#	Object Name	Northing	Easting
1		4435962.13	324946.09
2		4436056.86	325062.28
3		4436008.31	325005.15

Frames per Second: 0.20000 [✓] Auto Set Framerate

Min Agreeing Cameras: 3 Max Agreeing Cameras: 6

To use this functionality, review Section V of this documentation, and then do the following:

1. Open the Camera Setup Dialog Box and press the Load Survey Data button.



Archive
RRA02 Dugway.csv
RRA02 SDL.csv

File name: RRA02 Dugway.csv

Files of type: Survey Data (*.csv)

Open as read-only

2. Select the *.csv file containing the Survey Data spreadsheet (you may need to convert the spreadsheet to *.csv format using Microsoft Excel or a similar program)
3. Successive dialog boxes will open asking for the location of the *.cld files for each camera found in the survey data.
4. Important: If EXACTLY three hard targets were designated in each cloud in CEA using the Calibration Dialog before outputting the *.cld file, the cameras will be automatically given the correct orientation. Otherwise, only the position will be set.
5. See Section VII for advanced options such as selecting appropriate levels for the "min and max agreeing cameras" parameters.

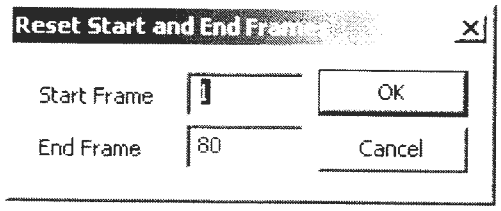
VII. Advanced Options

A. Camera Setup Options

1. Start and End Frames

New functionality has recently been added to the Camera Setup Dialog to allow the user to better control the starting and ending of individual cameras.

The frames that are processed from each camera can be modified by right clicking on the camera, or by left clicking on the camera and then clicking on the "**Set Start/End**" button.



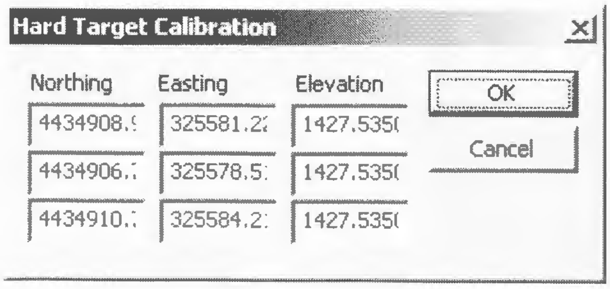
Note that these are the frames relative to the image set that the *.cld file that was processed, not relative to CVR!

2. Min/Max Agreeing Cameras

It is now possible to set two new parameters that make CVR more useful with large numbers of cameras:
Min Agreeing Cameras: This represents the minimum number of cameras that can see a point in space for that point to be tested for containing a cloud.
Max Agreeing Cameras: This represents the number of cameras that must agree that a certain point in space is part of an object to override one or more cameras that does not agree.
Recommended setting for 10 camera Dugway data:
Min Agreeing Cameras: 4
Max Agreeing Cameras: 6

3. Hard Target Calibration

The hard target information can be manually input or modified in this dialog. Once the northing, easting, and elevation are entered for each hard target, the camera's orientation, field of view, and angle from the horizontal are each calculated automatically. Note that this requires that the user specified exactly three hard targets in CEA.



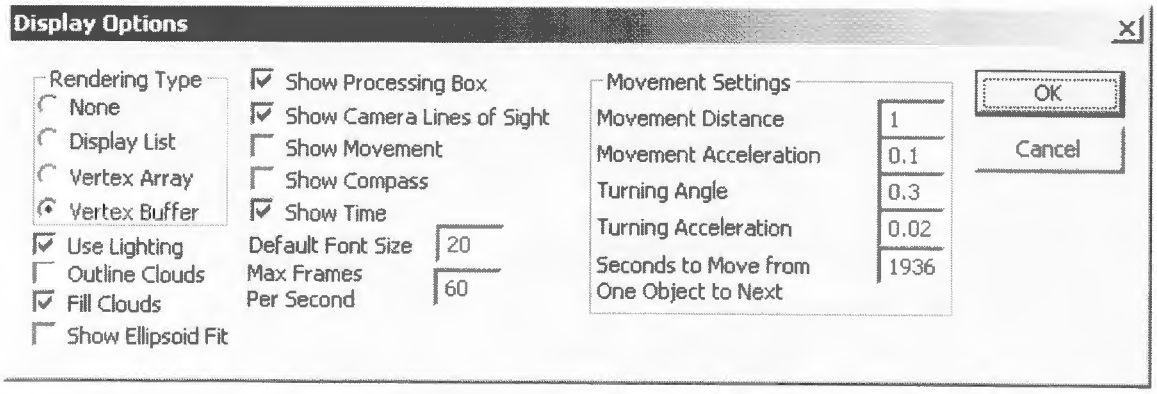
Hard Target Calibration

Northing	Easting	Elevation
4434908.	325581.2	1427.535
4434906.	325578.5	1427.535
4434910.	325584.2	1427.535

OK
Cancel

B. Display Options

There are several options available for displaying data. These are found by clicking on "Setup", "Display Options". This window is shown below.



Display Options

Rendering Type
- None
- Display List
- Vertex Array
- [✓ Vertex Buffer]

- [✓] Use Lighting
- [ ] Outline Clouds
- [✓] Fill Clouds
- [ ] Show Ellipsoid Fit

- [✓] Show Processing Box
- [✓] Show Camera Lines of Sight
- [ ] Show Movement
- [ ] Show Compass
- [✓] Show Time

Default Font Size: 20
Max Frames Per Second: 60

Movement Settings

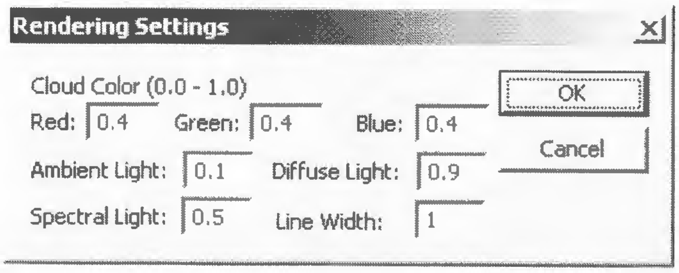
Movement Distance	1
Movement Acceleration	0.1
Turning Angle	0.3
Turning Acceleration	0.02
Seconds to Move from One Object to Next	1936

OK
Cancel

These settings are used to determine the display of the cloud data. These options are fairly straightforward, and the user is encouraged to try various combinations and see the results.

C. Render Settings

Settings describing the rendering of the image set can be changed by clicking on "Setup", "Render Settings". This window is shown below.



Rendering Settings

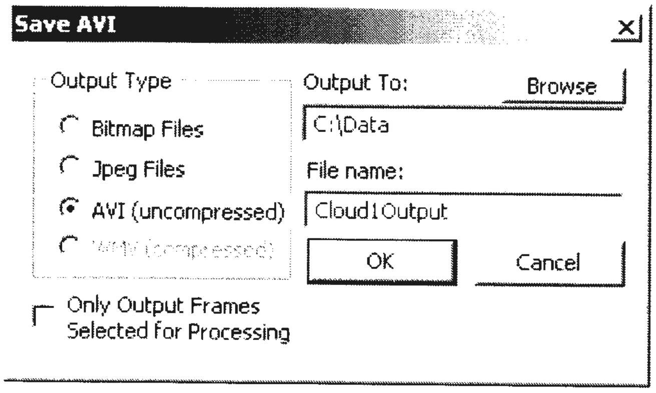
Cloud Color (0.0 - 1.0)
Red: 0.4 Green: 0.4 Blue: 0.4
Ambient Light: 0.1 Diffuse Light: 0.9
Spectral Light: 0.5 Line Width: 1

OK
Cancel

This window allows the user to customize the color and lighting environment of the image rendering. The user can set the color of the cloud through the Red, Green, and Blue options. The ambient, spectral, and diffuse light can be changed in this menu, by filling out the appropriate boxes. The line width of items such as the bounding box can also be changed here.

D. Graphical Output

By clicking on "Data", "Graphical Output", a user can export a video of the image rendered image results. This window dialog is shown below.



Save AVI

Output Type
- Bitmap Files
- Jpeg Files
- [✓ AVI (uncompressed)]
- WMV (compressed)

Only Output Frames Selected for Processing

Output To: Browse
C:\Data

File name:
Cloud1Output

OK Cancel

By clicking on the appropriate options, a user can set the desired file type and location of the output.

VIII. Cloud Reconstruction Algorithms

A. Camera to Cloud Angles

The algorithm contained in the function 'GenerateCameraVectors' in the 'CData' class is the primary algorithm used to determine which point in the cloud correspond to which point in the camera's field of view. To do this, it generates three 3D vectors:

- The vector that is parallel to the view angle of the camera
- A vector coming straight through the top of the camera
- The vector coming out of the right side.

These vectors are all orthonormal. They are used to determine how much of an effect the x,y, and z coordinates of the cloud have on the location of the corresponding pixel in the camera image. For instance, if the camera is pointing straight north and not tilted, the vector through the right side of the camera is (1, 0, 0) (where x is east/west, y is up/down and z is north/south). This means that the angle in the to the x coordinate in the y,z plane has an effect of 100% on which x pixel in the camera image that point corresponds to, has a weight of 1 in determining the x coordinate, while the y and z values have no weight. The vector through the top of the camera is (0,1,0) showing that the angle in the x,z plane to the y coordinate has a weight of 1 on the final y coordinate. In this simple setup, the direction (i.e. N,S,W,E) can be used to determine the x coordinate, and the elevation (from the horizon, i.e., 0 degrees is level) can be used directly to determine the pixel location. (Note: The third vector, pointing forward is used only for angle calculation)

The benefit of the type of calculation used here (i.e., 3 normalized vectors) is that **it can be used with any orientation of the camera**. There is no assumption that the camera is level, as with other programs. This algorithm will still function correctly if the camera is pointing straight down, turned on its side, or flipped in a random direction. It has performed as fast as the 3DCAV algorithm in tests.

Once this algorithm has completed running, the results are saved and used during each set of processing. There is no reason, currently, to recalculate these values every time. In the event that Dugway requires more movement of the cameras during processing, this function can be modified to store all of the probable angles of the camera to minimize reprocessing.

B. Location Algorithms

The location of a camera or other object is stored in two ways – UTM coordinates or longitude/latitude and elevation and the relative location of the object to the processing region (in meters). The relative location is used in calculations, and is found by using the latitude, longitude, and height. Though this software would probably have difficulties using this method at the poles, it should therefore be rather accurate at any other points on the Earth. The relative location is given in (x,y,z) coordinates. Because of the OpenGL implementation used, x goes from west to east as it increases, y goes from down to up, and z goes from north to south. For the sake of the algorithms, this is reversed for z to make it more intuitive and the z-values are all reversed before they are rendered.

IX. CLD File Format

*.cld files are one of two ways that CEA sends its data to CVR. The second method involves TCP/IP communication, and is currently disabled. These files contain the information needed by CVR to reconstruct 3D clouds from the camera data processed by CEA, including the following:

- File Version - Since the data stored in cld files changes, the fileVersion lets the program know if the file is valid, and in some cases it allows the program to load the data differently depending on the file version. This number changes every time the type of data stored in the *.cld files is changed (e.g. the software is modified).
- Number of Frames - The number of frames stored in the file.
- Frames Per Second - The frames recorded per second by the camera that produced the image set.
- Width - Width of each frame in pixels.
- Height - Height of each frame in pixels.
- Frame Info - Includes data about objects in each frame.
- Object Info - Includes data about the objects found by the software.

To generate *.cld files, follow the instructions in the Quick Start Guide for CEA. Then follow the directions in the Quick Start Guide for CVR to use the *.cld files for cloud reconstruction.

SIP: Operating in Real Time

Jeremy Pack

January 10, 2007

1 Goal

During the course of the Surveillance Image Processing related contracts with Dugway Proving Ground, it was made clear that DPG needed a real-time object tracking solution. A system was needed that would show what was occurring in the test range within less than a second of it occurring.

2 Rapid Prototype

The existing CEA and CVR was used to build up a quick prototype, capable of running in real time using camera input. CEA was started on each machine connected to a camera (through USB). Then CVR was started. Each computer running CEA was connected to the CVR computer, and all location data for the cameras was also provided. Once this was accomplished, each process was set to wait for a signal to begin processing.

Using this technique, it was possible (though unwieldy) to run real-time processing. SDL computers are synchronized very well, so the time data for each frame was stored as it was taken from the camera. This provided CVR with the accuracy it needed.

Based on these tests, it was clear that real-time processing was quite feasible. The rapid prototype was not put together in a robust fashion, however, and was only used as a proof of concept vehicle.

Tests were done using people walking through a scene seen by 3 or 4 cameras, and with other moving objects. The results were good.

Though the solution was only temporary, it showed a lot of potential for an expanded solution.

3 CEA Rewrite

As an initial stage, the CEA program was rewritten to focus on, and be optimized for, color camera input. Originally, CEA was intended primarily for use with IR and grayscale images. For color images, however, the algorithms did not make the best use of the information provided by each individual pixel.

Extension Library Proposal

Jeremy Pack

January 10, 2007

Abstract

This document describes a C++ library designed and implemented by the author for use in developing so-called plugins for C++ programs. This library was developed under the Boost software license (boost.org). It is intended for submission to the Boost C++ Libraries.

1 Purpose

For the purpose of this document, the library will be applied to the problem of an image processing application such as the following:

Consider an application that must load and process imagery that could be of the following types:

1. Grayscale bitmap or jpeg images
2. Infrared image data files from company X

Now, consider that the software development team knows that, in the future, more types of images may need to be loaded and processed, but they do not know which types. These types could include, for instance:

1. Color images
2. Png, tiff etc. formatted images
3. Infrared image data files from other companies (which may be in a format used only by that particular company)

Many of the algorithms and code developed for the first two types of images would apply to the latter three. Some functionality could perhaps be done in a more efficient or correct way for the later images though, and some new code could very well be necessary.

Differences between the images could include, among other things, the following:

1. The number of bits required to store the intensity information for each pixel

2. The number of channels for each pixel (i.e. grayscale requires one channel, color requires 3: red green and blue). Multi-spectral IR imagery could contain many channels.

3. The way that an image can be displayed to the screen

The standard object oriented response to such a problem is inheritance. Using C++, a base class similar to the following could be used:

```
class GenericImage
{
public:
  virtual load() = 0;
  virtual process() = 0;
}
```

Using this base class, subclasses such as CompanyXIRImage, BWBmpImage or BWJpegImage could be developed originally, and classes such as CompanyYIR-Image, ColorPngImage, ColorJpegImage etc. could be added later.

Consider, in addition, that perhaps certain algorithms would be appropriate for multiple types of images (i.e. perhaps all color images should be processed the same way once loaded). A multi-level inheritance structure would usually be appropriate, but sometimes this does not fit the problem well, and it may be more appropriate to split the image class into ImageLoader and Image-Processor classes, where a ColorImageProcessor can only be instantiated if a BmpImageLoader or JpegImageLoader has been instantiated and loaded color imagery.

Whenever new functionality needed to be added, a new derived class or classes would be developed and the program would be recompiled.

This is a good solution, but often it would be preferable not to have to recompile the original program. In addition, there could be various reasons not to want to include all of the functionality all of the time. In many different pieces of commercial and non-commercial software, different plugins are available which can enhance the functionality of the software. Sometimes these plugins are developed by the company that released the original software, and sometimes they are developed by other parties, even by the customers themselves.

This library, Boost.Extension, has a number of goals:

1. Make it possible to create these types of plugins, such as for multiple cameras and enable a program to detect available plugins automatically and correctly

2. Allow for the managed loading of interdependent plugins

3. Allow for these interdependent plugins to share information between themselves, even if that information was not known when the original base classes were designed.

4. Allow for complicated inheritance classes, including virtual base classes and multiple inheritance

5. Allow for loading of derived classes that cannot be modified in order to be made "loadable"

6. Allow for classes to provide information about themselves for loading purposes.

2 Possible Applications

2.1 Multiple Clients with Different Needs

In the situation above, each possible client could be using different camera equipment. Each client only needs to be able to load data from their own camera types. They don't require the additional functionality, so only the required linked libraries would be provided, but the base executable would be the same.

2.2 User-created Add-ons

In a web browser, often additional plugins are added to play certain types of files or provide additional information to the user. This technique could be used to simplify creation and use of these add-ons for such an application.

2.3 Classified or Secret Information

In many contracts, some of the algorithms or information incorporated into software should not be disclosed to other clients of the software. Secret or classified information could be placed in linked libraries that are only provided to those clients that have a right to see that information.

2.4 Complicated Inheritance Structures

This could also be used just as a standardized way to keep a large program modular and deal automatically with the difficulties inherent in dealing with the linked libraries in a directory as well as tracking interdependencies between loaded derived classes. Also, since the library is cross-platform, it can be used to work with linked libraries in general without having to worry about the different Posix or Windows APIs.

3 Proposed Solution

The library that has been designed meets the above goals and provides a relatively simple API to the user. There are a number of other libraries available that attempt to address this same problem, and some advantages of this solution include:

1. No macros are required - templates are used instead. There are several reasons why it is preferable not to use macros. In this library, templates are used without any loss of functionality, and actually make for rather concise code.

2. It is cross platform - Windows, Linux and Macintosh, and, theoretically, any Unix based system.

3. It requires only one externally visible function to be declared in each linked library. Other solutions usually make many functions and classes visible externally. In order to circumvent this, the one visible function provides functors that can call the required non-external functions.

4. It can automatically load all linked libraries from a given directory, or libraries selected by name.

5. It provides for making classes loadable that are in the same executable. In theory, it should be able to make it possible to load classes from other executables as well, if they are declared loadable.

4 Examples

4.1 Loading unmodified classes

This first example uses the library to load simple classes with no interdependencies. These classes thus do not need to be modified in any way to be made loadable.

The following represents the source code of a test file using Boost.Test to verify correct library functionality.

```
//  (C) Copyright Jeremy Pack 2006.
//  Use, modification and distribution are subject to the Boost Software License,
//  Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
//  http://www.boost.org/LICENSE_1_0.txt).
//
//  See http://www.boost.org/libs/type_traits for most recent version including doc
#include <boost/test/unit_test.hpp>
#include <boost/test/included/unit_test_framework.hpp>
#define BOOST_AUTO_TEST_MAIN
#include <boost/test/auto_unit_test.hpp>
#include <boost/extension/config.hpp>
#include <boost/thread.hpp>
#include <boost/extension/loader.hpp>
#include <boost/extension/extension.hpp>
#include <boost/extension/intrusive_capable.hpp>
#include <boost/extension/loadable.hpp>
#include "number_base.hpp"
```

```
using namespace boost::extensions;
class car//this is an abstract base class - it doesn't have to be abstract though
{
public:
  virtual ~car(){}
  virtual const char * get_type()=0;
};
class chevy : public car
{
public:
  virtual const char * get_type(){return "Chevrolet";}
};
class honda : public car
{
public:
  virtual const char * get_type(){return "Honda";}
};
BOOST_AUTO_UNIT_TEST(basic)
{
  /*create loader*/
  loader load;
  /*make local classes loadable*/
  library * lib = new library();
  lib->declare_unaltered<chevy, car>();
  lib->declare_unaltered<honda, car>();
  load.add_library("Local Classes", lib);
  /*create an object that can hold multiple cars*/
  multi_loadable<car> car_ptr(load);
  /*Get an iterator to the first available loadable class*/
  available_class_iterator it = car_ptr.get_class_begin();
  /*Make sure the list isn't empty*/
  if(it==car_ptr.get_class_end())
  {
    BOOST_CHECK(0);//This shouldn't happen
    return;
  }
  /*Load the class pointed to by the iterator*/
  car_ptr.load(it);
  /*Check that the virtual function output is correct*/
  BOOST_CHECK_EQUAL(std::string("Chevrolet"), std::string(car_ptr[0].get_type()));
  /*Go to the next available loadable class*/
  ++it;
  if(it==car_ptr.get_class_end())
  {
    BOOST_CHECK(0);//This shouldn't happen
    return;
  }
  /*Load Honda*/
  car_ptr.load(it);
  BOOST_CHECK_EQUAL(std::string("Honda"), std::string(car_ptr[1].get_type()));
  /*There are now two cars loaded into car_ptr*/
}
```

4.2 Loading interdependent classes

This example uses classes that are modified in order to provide tracking for interdependencies and to enable the classes to provide information about themselves.

```
//  (C) Copyright Jeremy Pack 2006.
//  Use, modification and distribution are subject to the Boost Software License,
//  Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
//  http://www.boost.org/LICENSE_1_0.txt).
//
//  See http://www.boost.org/libs/type_traits for most recent version including doci
#include <boost/thread.hpp>
#include <boost/extension/extension.hpp>
#include <boost/extension/repository.hpp>
#include <boost/extension/loadable.hpp>
#include <boost/test/unit_test.hpp>
#include <boost/test/included/unit_test_framework.hpp>
#define BOOST_AUTO_TEST_MAIN
#include <boost/test/auto_unit_test.hpp>
using namespace boost::extensions;
/*The classes below shows the most powerful way of declaring
loadable classes. For this method, each class must have
extension in it's inheritance hierarchy.*/
/*
structure is not itself loadable, but classes that inherit
from it are loadable as structures
*/
class structure : public extension
{
public:
  /*
    repository contains any shared variables and interfaces
    that are relevant for multiple loadable classes.
  */
  structure(repository & rep):extension(rep){}
  //This function is just here for demonstration.
  virtual std::string get_my_name(){return "generic structure";}
  /*
```

```
    Virtual destructors are not required - the extension
    virtual destructor takes care of those issues - but
    without virtual destructors, some compilers will give
    warnings (when a class has other virtual functions)
  */
  virtual ~structure(){}
};
/*
Notice that this class below is also not exported - though
it could be.
*/
class garage : public structure
{
protected:
  static void get_interface_info(boost::extensions::library_class & lc)
  {
    /*
    This function declares that
  garage is loadable as a structure, or as a garage.
  Dependencies could also be declared here:
  for example: lc.require<car>();
*/
    lc.provide<garage>();
    lc.provide<structure>();
  }
public:
  /*
    This style of constructor (with a single repository argument)
    is the simplest to use -
    but different constructors can be created with some work.
    They are not necessary, since all parameters can be placed
    in the repository, but in some cases a more specialized
    constructor is needed, and the library does not restrict this.
  */
  garage(repository & rep):structure(rep){}
  virtual std::string get_my_name(){return "some sort of garage";}
  virtual ~garage(){}
};
/*
house is exported. Notice the functions:
generate
get_extension_info - provide basic description
get_interface_info - list requirements and provisions
  The last two are separate so that a class can recursively
  include the dependencies of its super classes, if desired.
*/
```

```
class house : public structure
{
private:
  static void get_interface_info(boost::extensions::library_class & lc)
  {
    /*To construct a house, a garage is required to have been
    constructed.
    */
    lc.provide<house>();
    lc.provide<structure>();
    lc.require<garage>();
  }
  garage * garage_ptr;//pointer to the required garage
public:
  std::string get_garage_name(){return garage_ptr->get_my_name();}
  virtual std::string get_my_name(){return "some sort of house";}
  virtual ~house(){}
  static boost::extensions::extension * generate(boost::extensions::repository & rep
  {return new house(rep);}//standard generate function - required
  static void get_extension_info(boost::extensions::library_class & lc)
  {//called when a class is declared loadable
    lc.describe("A basic house");
    get_interface_info(lc);
  }
  house(boost::extensions::repository & rep)
    :structure(rep)
  {
    rep.set_to_first(garage_ptr);//Load the first available garage
  }
};
//similar to house
class four_car_garage : public garage
{
protected:
  static void get_interface_info(boost::extensions::library_class & lc)
  {
    lc.provide<four_car_garage>();
  }
public:
  static extension * generate(repository & rep){return new four_car_garage(rep);}
  static void get_extension_info(boost::extensions::library_class & lc)
  {
    lc.describe("A four car garage");
    get_interface_info(lc);
    garage::get_interface_info(lc);
  }
  four_car_garage(repository & rep):garage(rep){}
  virtual std::string get_my_name(){return "a four car garage";}
};
//similar to house
class two_car_garage : public garage
{
protected:
  static void get_interface_info(boost::extensions::library_class & lc)
  {
    lc.provide<two_car_garage>();
  }
public:
  static extension * generate(repository & rep){return new two_car_garage(rep);}
   static void get_extension_info(boost::extensions::library_class & lc)
  {
    lc.describe("A 2 car garage");
    get_interface_info(lc);
    garage::get_interface_info(lc);
  }
  two_car_garage(repository & rep):garage(rep){}
  virtual std::string get_my_name(){return "a two car garage";}
};
BOOST_AUTO_UNIT_TEST(creation)
{
  /*Upon creation, the loader searches the current
  directory. It is possible to also manually specify other
  files or directories*/
  loader load;
  /*
  This next part is only necessary because the
  classes are being loaded from the current executable.
                                  */
  library * lib = new library();
  lib->declare<two_car_garage>();
  lib->declare<four_car_garage>();
  lib->declare<house>();
  load.add_library("Local Classes", lib);//don't worry, library won't leak now
  //it is stored in a smart pointer.

  /*This single_loadable can carry exactly one instance of a garage
  it is initialized with the loader, and it points to a list
  of all available garages.
  */
  single_loadable<garage> garage_loader(load);
  //can load one house
  single_loadable<house> house_loader(load);
```

```
    /*No houses are available, becase the only house
    requires a garage, which has not been constructed.*/
    BOOST_CHECK_EQUAL(house_loader.get_num_available(), 0);
    /*Neither garage depends on anything, so they are both loadable*/
    BOOST_CHECK_EQUAL(garage_loader.get_num_available(), 2);
    if(garage_loader.get_num_available()!=2)
    {
      BOOST_CHECK(0);
      return;
    }
    /*Load the first garage*/
    garage_loader.load(garage_loader.get_class_begin());
    BOOST_CHECK_EQUAL(house_loader.get_num_available(), 1);
    /*Load the house - it will take a pointer to the
    loaded garage from the repository
    */
    house_loader.load(house_loader.get_class_begin());
    /*make sure that the first garage has loaded successfully.*/
    BOOST_CHECK_EQUAL(std::string(house_loader->get_garage_name()), std::string("a tw
  }
```

5 Design Recommendations

5.1 XML Descriptions

The library provides for creating a string to describe each class. A useful way to do this would be to create a parseable description, perhaps in XML. Thus, a BmpImageLoader can contain in its description a reference to the fact that it loads bitmap images, and describe it in a parseable way, such as:

```
<loader>
  <file-extension>bmp</file-extension>
</loader>
```

Whereas a jpeg could be:

```
<loader>
  <file-extension>jpeg</file-extension>
  <file-extension>jpg</file-extension>
<loader>
```

5.2 Small base classes and minimizing dependencies

In software development in general, minimizing dependencies is an important goal. Here as well, it is important to design classes in order to minimize coupling. Though it can handle very complicated interdependencies fine, it will complicate future development of plugins. The purpose of this library is to simplify the requirements for the programmers, but this is defeated if the interdependency feature is abused.

5.3 Avoid shared data - prefer shared interfaces

Although the library makes it possible to share actual data directly between multiple classes, it is better to share interfaces between classes, since this makes it easier to modify the system in the future and minimizes dependencies.