A Web Interface for a Large Calibrated Image Dataset

Zachary Bodnar
MIT Computer Graphics Group
Advanced Undergraduate Project
Prof. Seth Teller, Advisor
May 17, 2001

Abstract

During the Spring of 2000 the City Scanning Project team of the Computer Graphics Group at the Laboratory for Computer Science collected approximately ten thousand geo-referenced, color, digital images at over five hundred locations, or "nodes", in an out-door region of MIT's campus. Each of these nodes is a collection of 20 images which share a common optical center. The Computer Graphics Group would like to make this dataset available to other researchers in the field of machine vision, as well as members of the general public. This paper describes a world-wide-web based interface that allows members of the public to browse the data via the internet. The interface allows the user to navigate through the collection of nodes by ID number, date and time of acquisition, or by location by selecting points from clickable map of the campus. The interface provides access to the raw images, spherical textures generated from the nodes, files detailing the camera position and orientation for each of the images, as well as edge and point features extracted from each of the images. The inter face also provides a number of visualization tools such as a mosaic viewer that displays a planar projection a spherical texture created from the node, a tool that allows the user to view the epipolar geometry of two nearby nodes and a mini-map that shows a close-up of the node's location with lines connecting the node to each of its 3 or 4 nearest neighbors.

Introduction

Over a two month period in Spring 2000 the City Scanning Project team of the Computer Graphics group at the Laboratory for Computer Science collected roughly ten thousand geo-referenced color images at about five hundred locations in an-out door region of MIT's campus. At each of these "nodes" a camera fixed to an automated pan tilt head collected 20 images with a common optical center in orientations tiling a portion of the sphere. Each of these images in annotated with time and date of acquisition as well as approximate position and orientation with respect to an Earth-centered, Earth-fixed coordinate system. After acquisition a suite of algorithms was applied to the images during four stages of post-processing, mosaic, rotation, translation, and geo-referencing. These algorithms were used to recover intrinsic and extrinsic parameters, extract edge and point features and generate spherical textures from the images. The City Scanning team would like to make this large collection of calibrated imagery, which we believe to be the larges of its kind, available to the public and other researchers in the field.

To make this data easily accessible I have created a web interface to the image repository. The web interface allows users to browse the date set by node number or by selecting a node's location from a clickable map. When a user selects a node, a web page is generated using a Perl CGI script that displays thumbnails of the raw images that comprise the node as well as the spherical mosaic of these images viewed as a cylindrical projection. The spherical mosaic is also displayed, via a java applet, as a planar projection from a vantage point inside the sphere. The viewpoint of the projection can be rotated horizontally or vertically using the keyboard or mouse. In addition, the user can zoom-in or zoom-out of the projection. The node display page also contains links to directories containing the raw images, as well as the spherical and cubic mosaics -all at full, half, or one-quarter resolution. It also has links to directories containing the camera position files as well as the edge and point features for each image of each resolution from each stage of post-processing. Figure 1 shows a screen shot of the node display page:

Figure 1: A Screenshot of the Node Display Page Viewed With Netscape Navigator

On the node display page, as seen in Figure 1, the node's location is shown on mini-map of the node's immediate surroundings. Each node is color-coded to indicate the most refined stage of post processing that the node has completed and is labeled with its ID number. The mini-map also displays the other nearby nodes in the region of interest with lines connecting each of the visible nodes to its three or four nearest neighbors. These lines are called adjacencies. Clicking an adjacency on the mini-map brings up another java applet that displays planar projections of the spheres of both of the adjacent nodes. This applet can be used to view the epipolar geometry of the nodes. When the user selects a point in one of the images its epipole is displayed in the other.

Navigating the Data on the Web is Independent from how the Data is Organized on Disk

The nodes that comprise this dataset are all stored on a large disk array accessible from any computer on the Computer Graphics Group's intranet. The images, camera position files, features, mosaics, etc. that correspond to a single node are all stored in a unique directory named "nodeXXXX" where XXXX is the four digit ID number of the node. All of the node directories are stored in a single, top level directory called "all_nodes". Symbolic links to certain subsets of the nodes, such as the nodes around the Green Building or Ames Court are contained in other top-level directories. This provides access to some subsets of the data, but not necessarily the set of nodes a user accessing the repository over the internet might be interested in. For example, the remote user might be interested in nodes within a certain geographical region, nodes acquired on a given day, or any number of other attributes which are not implicit in the flat, one-level organization of the physical data. For this reason, it was important to provide a layer of abstraction between the way the data appears to be organized on the web, and its actual structure in physical storage.

To provide this abstraction I wrote a number of Perl scripts which are used to generate hierarchical symbolic link trees which map the physical data files to new directory structure which is suitable for navigation on the web. These trees are placed on the web server and given world readable permissions so that they can be browsed by a remote user. The primary advantage of abstracting the directory structure in this way is that eliminates the need to copy or move any physical data. Another advantage is that the mapping between the physical data and the way it is presented on the web is as simple as reassigning a few variable bindings in the Perl scripts. This method is disadvantageous, however, in that small changes in the organization of the physical data require complete regeneration of the symbolic link trees. A better solution might be the use of a transactional database which stores a pointer to each node directory and can be keyed on a number of different navigable attributes, such as data of acquisition, location in Cartesian coordinates, etc. This is one direction for further work on this project.

A symbolic link tree that is navigable by node number is generated by the Perl script **gensymtree2.pl**. This script basically creates a web accessible version of "all_nodes" in which links to the nodeXXXX directories are placed. In addition this script also re-maps the internal structure of the nodeXXXX directories themselves to a structure that is more intuitive to a web user who may be unfamiliar with the data. The re-mapping is shown in Figure 2:

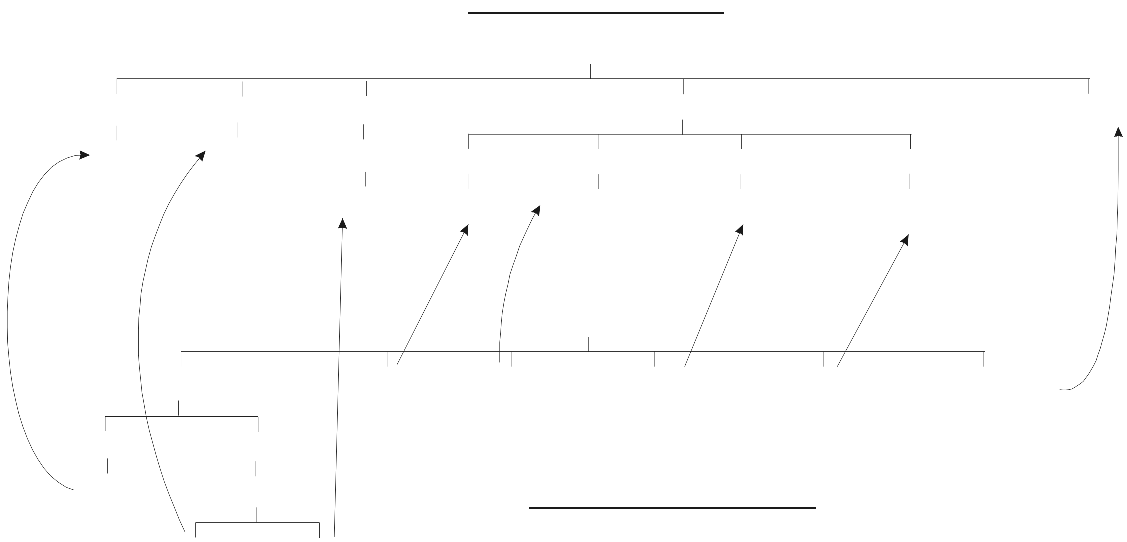

Figure 2: Re-mapping of the Physical Directories to the Web Accessible Navigation Tree

Of course, browsing by node number in the fashion is not the most natural way of getting at the data. An additional script, **gensymtree_by_date.pl**, constructs a symbolic link tree like the one generated by **gensymtree2.pl**, except that the nodes are organized into subdirectories named after the date and time of the session during which the nodes were acquired. This works by taking advantage of the fact that the collection of node directories in the physical "all_nodes" directory is itself an abstraction. When the nodes are acquired all of the nodes collected during a given session are stored in a single directory with a name such as Apr_25_00_1541, which denotes date and time that the camera was activated at the beginning of the session. The nodeXXXX directories are actually symbolic links to the node directories contained within the data-and-time directories. **gensymtree_by_date.pl** exposes this hidden organization by creating a symbolic link tree that duplicates the structure of the date-and-time directories. Like **gensymtree2.pl**, it also re-maps the nodeXXXX directories to the less complicated structure shown in Figure 2.

A Clickable Map Makes the Nodes Navigable by Location

Perhaps the most convenient way to visualize the dataset is by displaying the locations of the nodes superimposed on top of a map of MIT's campus. An interactive map that allows the user to select a node directory to view by clicking on the node's location is a nice way to navigate the dataset. I implemented a map viewer java applet to provide just such an interface. The map viewer loads the positions of the nodes from a file called **nodelocs.txt** which is generated by a Perl script, **getlocs.pl**, that is run prior to deployment of the website. This script extracts the most refined position data available for each node from the camera position files in each of the nodeXXXX dirs and places a line of the following format for each node in **nodelocs.txt**:

```
node0009 110.8753617353333900 -354.1783813950600600 0
```

The first element of each line is the name of the node's nodeXXXX dir. This is followed by the nodes x and y coordinates in meters, expressed in the City Group's local tangent plane coordinates (the z component of the node's location is omitted because it is of no use to the map viewer). The origin of these coordinates is at the location of the GPS base station on the roof of the Laboratory for Computer Science and is oriented with the y axis pointing north, and the z axis pointing in the vertical direction. The coordinates are extracted from the camera position files that were generated during the most advanced stage of post processing that has been completed for each node. The final element of each line in **nodelocs.txt** is an integer ranging from 0 -2 that denotes this level of refinement. Table 1 explains the meaning of these integers:

2	mosaiced	This is the first stage of post processing. During this stage the overlapping edges of the raw images are aligned and the images are blended together to form a spherical texture. This stage refines the orientation estimates of the raw images with respect to the optical center.
1	rotated	This is the second stage of post processing. During this stage the rotational estimate of the node's orientation is refined by performing the alignment of visible features.
0	translated	This is the final stage of post processing. During this stage the position estimates of the nodes are refined.

Table 1: Post Processing Refinement Codes

When the map viewer applet is instantiated each node's name, location and refinement is extracted from **nodelocs.txt** and stored in a java class representing a node. As each node is loaded, the City Group's coordinates are transformed into the local x, y coordinates of the map image. The origin of the map's local coordinate system is located at the point on the map where the GPS base station would be found (on the roof of the Laboratory for Computer Science). The transformation from the City Group's coordinates to the map viewer's local coordinates simply consists of a rotation, a translation, and multiplication by a scale factor. There is also a sign reversal of the y coordinate because north is up on the map, but the y axis in image coordinates points down. The coordinate transformation for the map viewer was calibrated by calculating the scale factor and rotation angle from the known position of the GPS base station and one of the nodes near the Green Building in map coordinates, used in conjunction with coordinates of the corresponding points in the City Groups coordinate system. I wrote a general 2D vector class in java, Vect2D, that is capable of computing the distance between two points, the angle between two vectors, rotating a vector by a fixed angle as well as other vector operations to aid with these transformations.

The nodes are rendered on the map as circles of different colors representing their level of post processing refinement. A click that lies on one of these circles is detected by determining whether the distance between the mouse click and a given node location is less than the radius of the circles rendered on the map. When a node is clicked on, a URL to the corresponding web page is constructed by appending the node's name to the URL of the base of the "all_nodes" symbolic link tree which is given to the map viewer via a <PARAM> tag in the HTML document in which the applet is embedded.

Adjacency Mode Displays a Mini-map Showing Each Node's Neighbors

The map viewer can also be invoked in adjacency mode by including the tag <PARAM Name="mode" VALUE="adj"> in the HTML document in which the map viewer is embedded. In adjacency mode the map viewer zooms in 4 times on the location of a particular node, which is also specified by a parameter in the HTML codebase of the applet. This node is highlighted with a green square. Each of the node's 3 or 4 nearest neighbor's (the default is 3 but this value can also be specified with a <PARAM> tag) is shown by a line that connects the node to each of its neighbors. These adjacencies are loaded from one of two files, **all_nodes_3.txt** or **all_nodes_4.txt** which correspond to 3 and 4 nearest neighbor adjacencies respectively. These files are generated during post processing, and are parsed when the map viewer is initialized. The entries in the adjacency files for each node are one line and have the following format, where the node IDs are indexed beginning at 0 and are listed in an abbreviated form (1 is equivalent to 0001, as in the directory name node0001):

```
Node 0 has 3 neighbors: 1 350 145
```

The adjacencies are stored in the map viewer as a vector of edges, where an edge is a class that represents a connection between two nodes. As the adjacency files are parsed the edges are inserted into the vector after first checking that a duplicate edge between the same two nodes does not already exist. That way if node **A** and node **B** both happen to be nearest neighbors of each other, a duplicate edge between them will not be inserted into the vector of adjacencies. This prevents the map viewer from wasting time by drawing adjacencies twice, or by searching through a redundant vector of edges when detecting whether a mouse click is near an edge. In adjacency mode, the map viewer also removes all of the nodes outside of the clipping region that are not nearest neighbors of a visible node from the vector of nodes. This is done so that the applet will not waste time rendering nodes or edges that are not visible, or searching through vectors which contain a large number of nodes or edges that are impossible to click on.

Click detection for edges is performed using the following algorithm: To test whether a mouse click is near an edge a vector **P** is formed from the first node of the edge to the coordinates of the mouse click and a second vector, **B**, is formed between the two nodes defining the edge. The point on the line through the two nodes that is nearest the coordinates of the mouse click is then found by projecting **P** onto **B**. If this point is within a reasonable radius (3 pixels) of the mouse click, the algorithm reports that the edge has been clicked on. Of course, the line passing through the two nodes is infinite in length, so the algorithm must also check that the x and y coordinates of the mouse click are contained within the bounding rectangle with vertices at the locations of the two nodes before it can be sure. When a pair of nodes lie on a vertical or horizontal line the bounding rectangle is infinitesimally thin in which case only the x values (in the horizontal case) or the y values (in the vertical case) are compared.

When an edge has been clicked on, the user is redirected to a URL for a Perl CGI script called **epiview.pl** that takes the two nodes comprising the edge as arguments and displays an applet that can be used for viewing the epipolar geometry of the two nodes. This applet is invoked in this way because epipolar geometry is only meaningful if two images share common features, and nodes that are nearest neighbors have the highest probability of sharing visible features.

HTML, CGI, and Java are Used for Data Presentation

The URL of the online interface is http://city.lcs.mit.edu/data/. This takes the user to the **index.html** document of the interface. This page displays some information about the dataset, its authors, as well as a paper that describes the data in more detail. This page also contains links to two HTML documents, **feedback.html** and **browse.html**, which are located in the same directory as **index.html**. **feedback.html** provides a form in which users can enter (optionally) their names, organizations, and email addresses as well as a few comments about the web site (required for submission). When the user submits this data a CGI Perl script called **feedback.pl** is invoked, which sends an email message to the City Group containing the user's comments and contact information.

The **browse.html** document is the real heart of the web interface. It contains links to the symbolic link trees and an instance of the map viewer applet that displays all of the nodes in the dataset. The links to the symbolic link trees do not actually point to the trees themselves but actually to a Perl CGI script called **dirlist.pl** which formats the directories nicely by displaying them as HTML pages with a white background. The subdirectories are shown as hyperlinks which are arranged into rows and columns. These hyperlinks actually call **dirlist.pl** recursively so that the entire symbolic link tree can be given a uniform appearance. I created **dirlist.pl** for two reasons. The first was to keep the appearance of the website uniform without having to create HTML documents for each branch of the symbolic link trees. The second was to format the data in each of the directories in such a way as to be both natural and informative to the user by suppressing files that are necessary for the operation of the web interface, but need not be made visible to the user, such as **nodelocs.txt**. **dirlist.pl** also makes it possible to navigate around the site using hyperlinks with informative names, rather than literal copies of the directory names. **dirlist.pl** takes two arguments, a root directory and a target directory. The root directory is the path to the parent of the target directory. The target directory is the actual directory to be displayed. These two arguments make it possible to call **dirlist.pl** recursively by forming a new root directory by appending the current target dir to the current root dir and then setting the new target dir to the selected subdirectory.

The nodeXXXX directories are handled with a special Perl CGI script, called **fmtnode.pl**. This script formats the data contained within the node directory into an HTML document that is meant to convey as much information possible about the data to the novice user. Text descriptions of each of the subdirectories are listed, as well as information about the images, mosaics, features, and camera pose files, such as when and how these files are generated and where they can be found. Hyperlinks to the image, feature, and post processing subdirectories are embedded naturally in the text descriptions. The web page generated by **fmtnode.pl** also contains an instance of the map viewer in adjacency mode with selected node as the center node, and an instance of the mosaic viewer.

Additionally, the HTML document generated by **fmtnode.pl** contains an image of the spherical mosaic viewed as a cylindrical projection as well as thumbnails of the raw images. These images are stored in .jp g format in the symbolic link trees' nodeXXXX directories under the "img/thumbnails" and "img/mosaic/sphere" subdirectories. The images of the actual repository are stored in the SGI .rgb format which most web browsers are not capable of viewing. This is the reason the .jpg thumbnails are used instead. The thumbnails are generated, prior to deployment of the website, by a Perl script called **genthumbs.pl**, which traverses the symbolic link trees and populates the nodeXXXX directories with copies of the raw images and the mosaics in .jpg format. **genthumbs.pl** performs the conversion from RGB to .jpg by invoking the UNIX command *convert*.

The Mosaic Viewer Displays a Planar Projection of the Spherical Texture

The mosaic viewer is a java applet which loads the spherical moscaic .jpg and warps it onto a planar projection. The image displayed is a view from inside the sphere looking out. The viewpoint can be rotated by clicking on the mosaic viewer and dragging the image around, or by using the arrow keys of th e keyboard. The z and w keys can be used to zoom in and out respectively.

The mosaic viewer also employs a gamma correcting class that improves the color saturation of the mosaic. Images encoded with RGB values from 0 -255 cannot capture the full dynamic range of the luminance values that are present in a typical outdoor scene which might contain both a very bright object like sun, and a very dark object, like a black sculpture. To overcome this problem the RGB files in the City Scanning Dataset are act ually encoded on a logarithmic scale , meaning the values from 0 -255 are actually logarithms of the true radiance values. When displayed using a linear scale from 0 -255, as is done on the web site, this causes some of the images with a high dynamic range to appear gray or solarized. This artifact could be undone in the mosaic viewer simply by translating the linear values back to the logarithmic scale, however, this would require an exponentiation on each pixel value which would slow the mosaic viewer to th e point of completely crippling its interactivity. For this reason I employed another approach to boosting the color saturation of the image in the mosaic viewer. I created a java class, called *GammaCorrector*, which is given a pointer to an image raster. This class then uses an algorithm developed by Manish Jethwa, a PhD candidate in the City Group, that I adapted to inflate the color space of the image.

This algorithm works by computing the 3x3 covariance matrix of the RGB values of the visible pixels. The covariance matrix is computed only for the visible pixels of the current view because the mosaic is formed from the tessellation of many smaller images, each of which may have dramatically different dynamic ranges. This means that some regions of the mosaic image may need more of a saturation boost than others, therefore it is better to inflate only the color space in the region of interest. Once the covariance matrix is formed it is factored into its eigenvector decomposition. This is done by using a freeware java matrix library called Jama, developed by *The MathWorks* and *NIST*, which is available online at http://math.nist.gov/javanumerics/jama/. The eigenvalues that are obtained from the eigenvector decomposition are the lengths of the major and minor axes of an ellipsoid that defines the color space of the visible image section. The eigenvalues are stretched so that ellipsoid fills as much of the RGB cube as possible and then the decomposed matrices are recombined to form a new 3x3 matrix called the reamp matrix. Each of the pixels, represented in the form of a 3D vector containing each of the RGB components, is multiplied by the reamp matrix in order to increase its saturation. This matrix multiply is hand coded in java floating point arithmetic rather than implemented using matrix functions from the Jama package because these generic matrix functions are meant to work on matrices of arbitrary dimensions and are very slow.

The Dual Node Viewer is Used to Visualize Epipolar Geometry

An epipolar visualization of two neighboring nodes can be brought up by clicking on one of the adjacencies displayed on the mini-map on the node view page. This sends the user to a web page generated by the Perl CGI script **epiview.pl** which contains an instance of the Dual Node Viewer java applet. The Dual Node Viewer is basically a two panel, multithreaded version of the Mosaic Viewer and displays a view from the optical center of each of the two neighboring nodes. The mouse and the keyboard can be used to rotate the viewpoint or zoom in and out of each of the two views in the same manner as in the single view Mosaic Viewer. **Figure 3** and **Figure 4** (next two pages) are typical screenshots of the Dual Node Viewer.

The animation loop of each of the two panels runs in a separate thread. This is necessary for the components to be drawn correctly using Java's component model architecture. When one of the two panel has the mouse focus the animation thread that renders the planar projection of the image runs at 24 frames per second, but when the panel has lost the focus the animation thread sleeps for one tenth of a second before checking to see if has regained the focus. If this is the case, the animation thread continues, otherwise the sleep cycle repeats. This ensures that when the user rotates the viewpoint of one of the projection panels the animation will not be slowed because the animation thread of the second stationary image is being scheduled needlessly often.

To display an epipole using the Dual Node Viewer a user first clicks on the "show epipole" button. When the "show epipole" button is clicked the mouse cursor in each of the views changes to a crosshair, signifying that the function of a mouse click has changed from rotating the sphere to projecting an epipole. When the user clicks on a point in one of the views the point is marked with a red cross. In the other view a line representing a ray originating at the optical center of the first node and passing through the point selected in the first view, is displayed in the second. This line begins at the location of the first node

Figure 3: Epipolar Visualization Showing Node 347's Optical Center as Viewed from Node 348. *Points selected in node 347 (red crosses) are seen as rays originating from the optical center of node 347 (blue cross) viewed from the vantage point of node 348.*

(which is marked by a blue cross circumscribed by a blue circle) and terminates at a vanishing point on the opposite side of the sphere. The screenshots in **Figure 3** and **Figure 4** illustrate this more clearly.

In **Figure 3** the user has selected two points on the spires of the large black sculpture in the view from node 347. These points are marked with red crosses. In the view from node 348 the rays projected through the selected points are shown as green lines originating at the optical center of node 347. The optical center of node 347 is marked by the blue cross circumscribed by the blue circle in the view from node 348. The epipolar lines intersect with the same two spires of the black sculpture demonstrating that the positions and orientations of the two cameras are well registered. The viewpoint from node 348 is in the general direction of node 347 so the vanishing point of the epipolar lines is not visible.

Figure 4 shows a different epipolar visualization. In **Figure 4** the point of view from node 349 is looking toward node 350, which is visible from node 349 as the blue cross inside the blue circle. In the view from node 349 the user has selected points on the corners of two buildings and a third point near the corner of one of the central building's windows. The view from node 350 is looking away from node 349. For this reason the epipolar lines appear to pass overhead in the view from node 350 and converge at a vanishing point in the lower right of the image. The location of this vanishing point is the point on the sphere that is directly opposite the position of node 349. In other words, it is the mirror image of node 349's location projected onto the sphere.

The epipolar visualization is uses three coordinate transformations to change two dimensional image coordinates to 3D vectors in the node's local coordinate system, *rayToSphere*, *pointToRay*, and *rayToPoint*. The node's local coordinate system is a right handed coordinate system with the y axis

Figure 4: Epipolar Lines Originating from Node 349 Converge at a Vanishing Point in a View from Node 350. *The view on the left is from node 350 looking away from node 349. Since node 349 is behind the viewer in this projection, the epipolar lines appear to pass overhead and converge at a distant vanishing point.*

pointing down. *pointToRay* and *rayToPoint* are implemented as methods of the Java class *PlanarMap* which encapsulates the transformations needed to project a vector in 3D space to a point on a planar image oriented in the direction of the current viewpoint. *rayToPoint* takes a 3D vector and returns the corresponding x, y point in the planar projection of the current view. *pointToRay* is the inverse transformation. It takes the (x, y) coordinates of a point in the planar projection of the current view and returns a 3D vector in the direction of the ray that originates at the camera's optical center and intersects the planar projection at (x, y). The third transformation, *rayToSphere*, takes a ray in 3D coordinates and returns the (x, y) coordinates of the corresponding point in the cylindrical projection of the spherical mosaic (this is the flat, unwrapped image of the spherical texture that is stored in the "img/mosaic/half/sphere" directory).

rayToPoint, pointToRay, and *rayToSphere* transform coordinates between the 2D coordinates of planar or cylindrical projections and the local 3D coordinate system of the node. In order for 3D rays in one node to be projected in the other, the rays must first be transformed to world coordinates (coordinates in the City Group's local tangent plane coordinate system) and then transformed into the local coordinates of the second node. This transformation is done using information from the nodes' camera files. There is a camera file for each of the 20 raw images that make up a node, a portion of which describes the image's estimated position and rotation with respect to world coordinates:

```
TRANSLATION   136.9876810140612500  -392.9228685741972000  -40.6730981659422710
ROTATION      -0.7053809944651622   -0.6998717346941172    -0.0816009954282809
0.0771912246871364
```

The translation line gives the (x, y, z) Cartesian coordinates of the camera's optical center in the City Group's local tangent plane coordinates. The rotation line is a quaternion of the form (t, x, y , z) which describes the rotation used to rotate a vector in world coordinates to the orientation of the image's local coordinate axes. A 3x3 matrix that will perform this rotation on any 3D vector that it multiplies is obtained from the quaternion using the formula:

$$R = \begin{matrix} t^2 + x^2 - y^2 - z^2 & 2xy + 2tz & 2xz - 2ty \\ 2xy - 2tz & t^2 + x^2 - y^2 - z^2 & 2yz + 2tz \\ 2xz + 2ty & 2yz - 2tx & t^2 + x^2 - y^2 - z^2 \end{matrix}$$

This rotation matrix is symmetric and orthogonal so the inverse rotation to world coordinates can be obtained by taking its transpose.

The quaternion listed in each of the camera files describes the rotation from world coordinates to the coordinates of the raw image (a right handed coordinate system with z representing depth) not the local coordinate system of the node, so the rotation will be different for each of the 20 images. The orientation of the raw images with respect to the node's local coordinates, however, is the same for all nodes; i.e. the azimuth and elevation of any raw image's viewpoint, taken with respect to it s node's local coordinates, will be the same for the corresponding image in any other node. This means that the relative rotation between the local coordinate systems of two nodes is simply the product of the rotation from a single image in the first node with the rotation to the coordinates of the corresponding image in the second node. In esse nce, we can use a single image in each node to compute the transformation so long as we use the same base image in each node. Image 0 is guaranteed to exist for every node so it is the best choice of base image.

These transformations are applied in the f ollowing way to project an epipolar line: When the user designates a point of projection in node **A,** by clicking somewhere in its planar projection, *pointToRay* is called on the pixel coordinates of the selected point. This returns a ray, represented by a 3D vector, in the node's local coordinates. To model this ray as a line we actually need two vectors in the direction of the ray (i.e. two points that lie on the epipolar line). The first point can simply be the location of node **A**. The second point is obtained by scaling the ray by a reasonably large number (10,000 meters) to get a point that is very distant from node **A**. When these two vectors are transformed to **B**'s coordinates the epipole can be rendered by drawing the line that passes through the two points.

Each of the two planar views contains a pointer to its neighbor (the reference counting mechanism used by Java's garbage collector can handle such circular references safely, without causing memory leaks or segmentation faults). The Java class wh ich implements the planar projection, *NodeVR*, contains a method called *exportEpipoles.* Every time a NodeVR is redrawn it calls its neighbor's *exportEpipoles* method, which returns a vector of the rays projected from the neighboring node in world coordinate s (Though the discussion here involves only two nodes, *NodeVR* can actually handle references to multiple neighbors so that in the future it will be possible to expand the epipolar visualization to support any number of different views). Although each epip olar line is modeled as two points, this vector contains only the coordinates of the distant points since the starting point of each epipolar line, the location of the neighboring node from which it was projected, is implicit. It can be obtained by calling a separate method of *NodeVR* called *getLocation*, which returns the node's position in world coordinates. *exportEpipoles* transforms each of the distant points to world coordinates by first applying rotation between the coordinate system of the base image and the world coordinate system. The rotated vector is then translated to world coordinates by adding it to the 3D vector formed from the coordinates of the node's optical center. After node **B** has obtained the rays projected from node **A** by calling *exportEpipoles*, it then transforms each of the vectors into its own local coordinate system by first subtracting the node's location from each vector. After the vectors have been translated this way they are rotated into node **B**'s local coordinate system by applying the rotation computed from the quaternion in the camera file of its own image 0.

In order to ensure that the planar projection that is displayed on the screen is in alignment with all of these transformations I implemented the warp function that takes the pixel values from the cylindrical texture file and maps them to image coordinates using the same functions used to transform the epipoles. The warp function works by iterating over each pixel in the planar projection and calling *pointToRay* on its (x, y) coordinates. This ray is then used to find the corresponding pixel in the cylindrical texture by calling *rayToSphere.* Any rays that don't intersect with the texture because they are outside of the node's field of view are rendered in blue. The use of *pointToRay* in both the transformation of node coordinates and pixel coordinates ensures that the epipoles rendered in each projection will line up properly with the projected texture map.

The Planar Projections are Not Unique

The projection from a ray in world coordinates to a point in the planar projection that is performed by *rayToPoint* is not one-to-one. A single ray projects to two different points on the opposite side of the sphere. Only one of these points is visible at any time, since they are in opposite hemispheres. To check that only the correct projection is drawn, and not its mirror image, it is necessary to check whether the ray to be projected corresponds to a point in front of the viewer. If it is a point behind the viewer, the mirror image point will be projected. To check whether a vector is in front or behind the viewer the angle between it and the current viewpoint is computed. If this angle is greater than 90° the ray to be projected is behind the viewer and the projection should not be rendered.

The situation is more complicated for epipolar lines, however, because the lines are represented by two points. One of these points could be in front of the viewer while the other is behind. In order for the line to be rendered correctly, though, two points in front of the viewer are needed. In this situation I use a trick to find a second point on the epipolar line that is in front of the viewer. Suppose we define a plane that is perpendicular to the current viewing direction and divides the sphere in half. The epipolar line must pass through this plane. One could find the point of intersection and use it as the second point of projection in rendering the epipolar line. In practice, however points that are on the boundary between the two hemispheres could still result in mirror image problems. If the plane is moved forward a little bit in the direction of the viewpoint, however (the actual value I used is one meter), the intersection between the epipolar line and the plane will certainly be in front of the viewer and will project to the correct point in the planar map. This is illustrated more clearly in Figure 5:

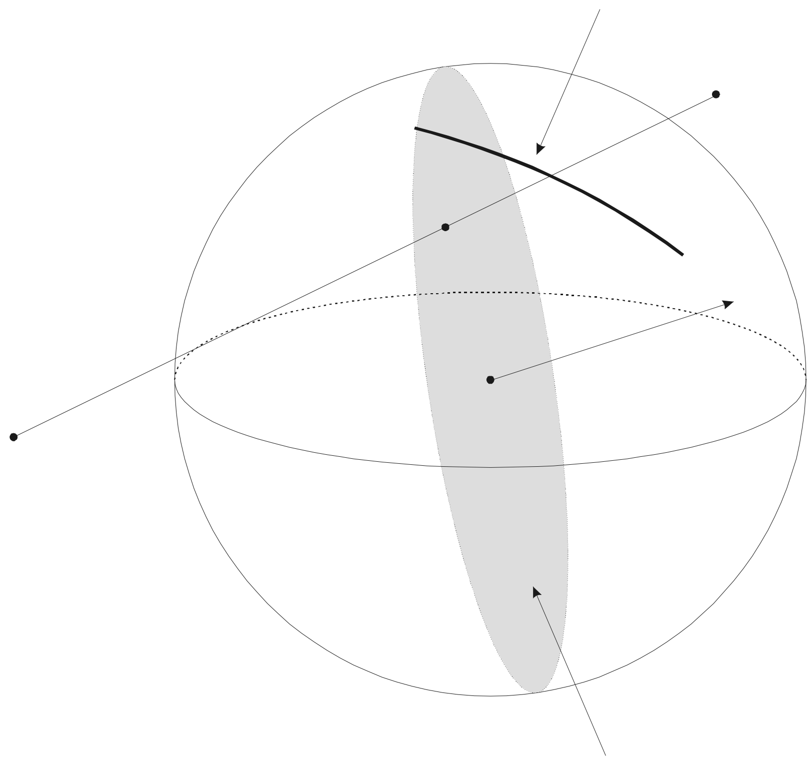

Figure 5: Finding a Point on the Epipolar Line in Front of the Viewer By Intersection with a Plane

In Figure 5 points **A** and **C** define an epipolar line. Point **C** is in front of the viewer but point **A** is behind. The gray oval represents a portion of a plane that is perpendicular to the direction of the current viewpoint and is one meter beyond the camera's optical center in the direction of the view point. Point **B** is where the epipolar line intersects this plane and is in front of the viewer. Points **B** and **C** are used to project the epipole, which is shown as it would appear if it were projected on the surface of the sphere.

CVS and Make are Used to Maintain Parallel Production and Development Environments

As with any software product that is to be made available to the public, it was important to maintain parallel production and development environments. The users expect a fully functional, bug-free website, but it was also necessary, for me as a developer, to have a suitable test bed for new features and extensions to the existing web site. By creating a completely parallel version of the site that was not available on a publicly disclosed URL I could perform all of my development and testing without disturbing the published site that was made available to users. When new components were ready to be placed on the production website they could simply be copied from the development version to the production site. Revisions to the website typically involve multiple files, however, so copying the necessary changes by hand can be cumbersome and unreliable. For this reason, I created a makefile that can generate two versions of the site from a common set of source files.

The source code for the web site is maintained in a CVS repository with the hierarchy shown in Figure 6. The source code is checked out to the web server using the command `cvs –co –d city citydata` which checks out the citydata module shown in Figure 6 to the directory "city" on the web server. make is then run in the "city" directory to build a working version of the web site. Figure 7 shows the structure of the site that is generated on the City Group's web server.

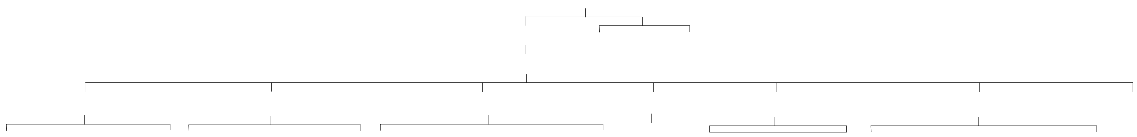

Figure 6: CVS Source Tree for the Citydata Web Interface

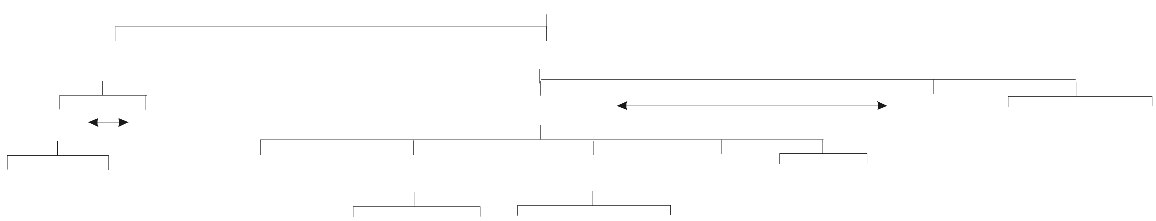

Figure 7 Structure of the Website Deployed on the City Group's Web Server

In Figure 7 the deployment directories, rel and dev, are shown in bold. The arrows in Figure 7 indicate production and development directories that are equivalent. The subdirectories of the production directories are omitted in Figure 7 for clarity because their structure is identical to that of the development directories, with the sole exception of the "src" directory. The "src" directory contained in the development directory "datadev" is the common source of both environments, and is not duplicated to the production directory "data".

The makefiles **Makefile** and **main.mk** contain the instructions to build both the development and the production versions of the site. If make is invoked without any arguments, the site is built and deployed to the development directories. If make is run with the parameter "release" (as in `make release`) the site is built and deployed to the production directories. In this way the production site is updated with any changes that have been made on the development site automatically. The production and development sites are accessed via two different URLs. http://city.lcs.mit.edu/data points to the production site, and http://city.lcs.mit.edu/datadev points to the development site.

When the site is built using make, the build scripts **gensymtree2.pl**, **genthumbs.pl**, and **getlocs.pl** are used to create and populate the symbolic link trees as described above. The CGI scripts are then copied from their location in the "src" directory to the appropriate deployment directory on the web server (If you look carefully at Figure 6 you will notice that the CGI Perl scripts' filenames are prefixed with src_ in the source tree. This is because make cannot handle dependencies between source files and target files of the same name). The HTML documents must then also be copied to the appropriate locations, however, there is a small difficulty that arises from the fact that the CGI scripts and the HTML documents are deployed to different locations on the web server. In accordance with good software development practice, I used relative paths in the HTML wherever possible. However, because the "cgi-bin" directory of the web server is not a subdirectory of the "datadev" or "data" directories where the HTML documents are located, relative path names are not sufficient to distinguish between the CGI scripts of the development and production environments. Conversely, CGI scripts must know the complete path to the HTML documents or java binaries in order to reference the files which correspond to the appropriate environment. This problem is addressed by two additional build scripts, **configure.pl**, and **genhtml.pl**.

configure.pl is invoked by **main.mk** with arguments that specify the target directory of the HTML code base, which is either "datadev" or "data" depending on whether a development or production version is being built, as well as the target directory for the CGI scripts. This script generates a file called **cggd.conf** which is placed in the appropriate CGI directory (either "dev" or "rel"). This configuration file tells the CGI scripts where the HTML and java classes are located.

The **genhtml.pl** script makes the corresponding path bindings in the HTML documents. This done by performing a search and replace on the .html_src documents located in the "html_src" directory. These .html_src files are special HTML documents with tags that I created that are meant to function as unresolved variables to be bound by **genhtml.pl** at build time. These variables mainly consist of references to CGI scripts which are replaced with the appropriate URLs for a given environment by the **genhtml.pl** script. After the bindings are made, the resulting HTML documents are copied to their appropriate locations on the web server.

The rest of the build process is straightforward. **main.mk** invokes the java compiler on the java source files using the –d argument to specify the appropriate deployment directory. **main.mk** also copies all of the images referenced in the HTML from the source tree to their appropriate locations. Sometimes, if the site is being built on a clean web server, most of the deployment directories will not exist. In this case the deployment directories are first created by the **main.mk** and then assigned the necessary world readable permissions. The build process then proceeds as described above.

Opportunities for Further Work

The system that I have developed thus far provides a great deal of new functionality but it is only the first step in what could be a very powerful and versatile interface. Some possibilities for future work on this project include:

- **Improving the abstraction between the physical data and the navigable hierarchy presented to the user** The symbolic link trees provide an excellent re-mapping of the physical directory to a more intuitive organization, but any small changes require re-construction of the trees so they will not be very utilitarian when the dataset begins to grow or change more rapidly
- **Developing more advanced visualization tools** One could envision applets that allow the user to view a re-projection of the image data onto crude proxy geometry. Another useful visualization would be one that shows the boundaries of the current planar projection in the mosaic viewer in the cylindrical projection of the sphere to illustrate the action of the coordinate transformations. The *sphereToRay* transformation could also be used to create a better way to change the viewpoint of the Mosaic Viewer and the Dual Node Viewer. Instead of dragging the viewpoint around using the mouse, the user could simply select a point of view from the cylindrical projection.
- **Adding features to the epipolar visualization** The epipolar visualization could be made to place a metric scale of tick marks on each epipolar line to give an indication of distance. In addition all of transformations are in place to display the edge and point features extracted from the raw images in the planar projection of the sphere; it's just a matter of writing code to put this feature in place. Once the features are displayed, they could then be used to select points of projection that would better illustrate the quality of the rotation and position estimates because the edge and point feature data has better than sub pixel accuracy. Finally, since camera files are available for each stage of post processing, one ought to be able to view the epipolar geometry at each of the different stages to see how the rotation and position estimates improve.
- **Registering the clickable map (and any future map-based visualizations) with a more universal coordinate system.** The map should be registered with some standard system such as NSR522 State Plane Coordinates. This would not only make the maps easier to work with for members of the City Group, but would also provide useful information about node and feature positions to people outside the City Group who are browsing the data over the web.
- **Improving the quality of the displayed images.** Currently only the node viewer applets provide gamma correction. The raw images and the sphere images are still displayed without re-mapping the radiance values. Also, since most browsers cannot display .rgb files, it should be made possible to view the .rgb image files over the web using a java applet or some other means.

- **Integration of the navigation interface with a transactional database.** This could allow users to find nodes based on other criteria besides the curr ent navigable features of node number, time of acquisition, and node location. For example, the user could query the database for all nodes in a certain region, or all nodes acquired using a lens with a particular coefficient of radial distortion.

Contributions

The City Scanning Dataset WWW interface is still very much a work in progress, but in the short time span of one semester, I have made important headway in the following areas:

I have

- Created a informative layout with useful applications that user's who may be unfamiliar with the data can use to browse the dataset, and learn about the data
- Set up separate production and development environments with a clean mechanism for maintaining the two parallel, concurrent versions of the site
- Developed applications that can be used to view the locations of the nodes, information about their adjacencies, the spherical textures generated by the mosaic stage, and the epipolar geometry of neighboring nodes.
- Formed an abstraction between the physical data and the way that it appears to be laid out to the remote user.
- Improved the quality of the viewable images using the *GammaCorrector*
- Written CGI and java applications, such as the clickable map, to make navigation of the web site as intuitive as possible
- Created a CGI form for users to provide anonymous feedback back to the City Group about the site.
- Created an epipolar visualization tool that can be used to demonstrate the quality of the image registration.

Appendix A: Summary of Script Names and Functions

Build Scripts

Table 2 lists the makefiles used to build either a production or development version of the website:

main.mk	primary makefile that is used by **Makefile** to build either a production or a development snapshot of the web site.
Makefile	makefile that initiates the build process by providing the arguments to **main.mk** that will build either a development or a production site

Table 2: Summary of Makefile Names and Functions

Table 3 lists the Perl scripts which are used by *make* in building the website:

configure.pl	generates **cggd.conf**, the configuration file describes the path to the html and CGI deployment directories
genhtml.pl	parses .html_src files and completes variable bindings
gensymtree2.pl	generates a symbolic link tree to the nodeXXXX directories organized by node number
gensymtree_by_date.pl	generates a symbolic link tree to the nodeXXXX directories organized by date and time of acquisition
genthumbs.pl	traverses symbolic link tree and creates .jpg thumbnails of the raw images and mosaics
getlocs.pl	generates **nodelocs.txt**, a file which describes the position of each node used by the map viewer
publish_src.pl	copies any source code to be made publicly available to an appropriate location on the website

Table 3: Summary of Perl Build Script Names and Functions

CGI Scripts

Table 4 summarizes the Perl CGI scripts used by the web interface:

cggd_init.pl	library routines used by the CGI scripts for parsing the configuration file **cggd.conf**
cgi-lib.pl	open source library of Perl CGI routines
dirlist.pl	CGI script that generates an html page with links to the directory contents and subdirectories formatted in a nice tabular structure
epiview.pl	generates an HTML document to display the epipolar visualization applet
feedback.pl	handles submissions from the HTML feedback form in **feedback.html** by sending an email message to the City Group
fmtnode.pl	displays the contents of a nodeXXXX directory using HTML and Java applets

Table 4: Summary of CGI Perl Script Names and Functions

Appendix B: Summary of Java Classes and Their Uses

Map Viewer Classes

Table 5 summarizes the Java classes used by the Map Viewer applet and their functions:

Edge.java	represents a clickable adjacency between two nodes
mapvr.java	main applet class for the Map Viewer
Node.java	storage class for node's location and other information; provides a hit test for detecting mouse clicks
Vect2D	represents a two dimensional vector and provides vector manipulation functions

Table 5: Summary of Map Viewer Java Classes and their Uses

Mosaic Viewer and Dual Node Viewer Classes

Table 6 summarizes the Java classes used by the Mosaic Viewer and Dual Node Viewer applets and their purposes:

ClickMode.java	shared variable used by the two different view panels that describes the cu rrent function of a mouse click
DualNodeVR.java	main applet class for the Dual Node Viewer
GammaCorrector.java	boosts the color saturation of an image map
Matrix3x3.java	3x3 matrix representation with matrix manipulation functions
PlanarMap.java	performs the projection from rays in space to a plane
Raster.java	image raster class that holds an array of pixels in memory
Vector3D	3D vector representation with vector manipulation routines
NodeVR.java	animation panel used by the Dual Node Viewer that contains one planar projection of a node which can be rotated, scaled and used to display epipolar lines
nodeVR.java	main applet class for the Mosaic Viewer
Pose.java	storage class for a node's location and orientation with methods for transforming to and from world coordinates.
Quaternion.java	simple quaternion class with methods for producing rotation matrices

Table 6: Summary of the Mosaic Veiwer Java Classes and their Functions