Aggregating Building Fragments Generated from Geo-Referenced Imagery into Urban Models

Barbara M. Cutler

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

19 May 1999

Author ____

Department of Electrical Engineering and Computer Science
19 May 1999

Certified by ____

Seth Teller
Thesis Supervisor

Accepted by ____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Aggregating Building Fragments Generated from Geo-Referenced Imagery into Urban Models

Barbara M. Cutler

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

19 May 1999

The City Scanning Project of the MIT Computer Graphics Group is developing a system to create automatically three-dimensional models of urban environments from imagery annotated with the position and orientation of the camera, date, and time of day. These models can then be used in visualizations and walk-throughs of the acquired environments.

A camera platform equipped with navigation information acquires the large datasets of images necessary to adequately represent the complex nature of the urban environment. Drawing appropriate conclusions from multiple images is difficult without a priori knowledge of the environment or human assistance to identify and correlate important model features. Existing tools to create detailed models from images require large amounts of human effort in both the data acquisition and data analysis stages.

The aggregation algorithm presented merges the output from the work of several other graduate students on the project. Each of their programs analyzes the photographic data and generates three-dimensional building fragments. Some of these algorithms may work better and produce more accurate and detailed fragments for particular building structures or portions of environments than others. The aggregation algorithm determines appropriate weights for each these data fragments as it creates surfaces and volumes to represent structure in the urban environment. A graphical interface allows the user to compare building fragment data and inspect the final model.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would like to thank the other students in the Computer Graphics Group: J.P. Mellor, Satyan Coorg, Neel Master, and George Chou for providing me with samples of output from their programs as they became available; Rebecca Xiong, for the use of her Random City Model Generator; Kavita Bala, for explaining how to organize Edgels within a 4D-tree; Eric Amram, for the XForms OpenGL interface code; and Eric Brittain, for helpful comments on the pictures and diagrams in the thesis. My advisor, Professor Seth Teller, suggested ways to visually analyze and aggregate building fragments and pointed me in the direction of several helpful papers and sources of test data.

Erin Janssen has been a great friend and provided many years of editorial criticism, even though she didn't wade through my entire thesis. Derek Bruening helped me find several insidious errors in my program, and after being bribed with strawberry-rhubarb pie, he has patiently helped me proofread.

Carol Hansen, my figure skating coach, has been very helpful in diverting my attention. With her help, I landed my first axel ($1\frac{1}{2}$ revolutions!) and I recently passed my Pre-Juvenile Freestyle test (which is more of an accomplishment than its name implies). My parents have always been very supportive of whatever I want to do, but this thesis proves to them that I *have* been doing more than just skating in graduate school.

Contents

Chapter 1

Introduction

Capturing information about the structure and appearance of large three-dimensional objects is an important and complex computer graphics and vision problem. The goal of the MIT City Scanning Project is to acquire digital images tagged with an estimate of the camera position and orientation (pose) and automatically create a three-dimensional computer graphics model of an urban environment.

Several different approaches to environment reconstruction using these images are under development in the group. The algorithms produce *building fragments* with varying degrees of success depending on the quality of the images and annotated pose or geometrical properties of the environment. My program aggregates the available building fragments into a model which can be used in visualizations. The graphical interface to my program allows the user to inspect the data objects and intermediate results and adjust aggregation parameters.

The next few sections describe the images, the algorithms that analyze them to produce building fragments, and possible uses of the final model.

1.1 Photographic Images

The first image dataset used by the group, the *Synthetic Image Dataset*, was a set of 100 images oriented uniformly in a circle around an Inventor model of the four buildings in Technology Square (see Figure 1.1). The model was texture-mapped with photographs of the buildings cropped and scaled appropriately. The dataset was not very realistic for several reasons. The camera placement and orientation was very regular and did not thoroughly

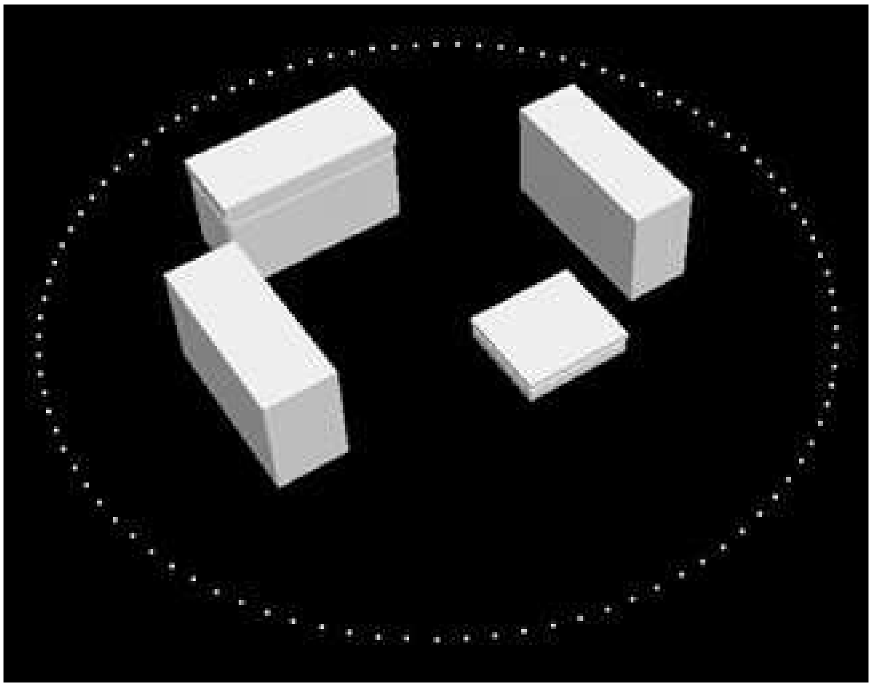


Figure 1.1: Visualizations of the camera positions for the *Synthetic Image Dataset*.

sample the environment. The uniformly-colored ground plane and sky background of the images could be easily cropped out to focus on the buildings. The most important failing of the *Synthetic Image Dataset* was the simplistic lighting model. Comparing stereo image pairs is much more difficult if the objects' colors change due to specular reflection or time of day and weather conditions.

The next dataset used by the group was the *100 Nodes Dataset*. Approximately 100 points located around the buildings of Technology Square were surveyed, and at each of these nodes approximately fifty high-resolution digital images were taken in a pattern tiling the environment visible from the node (see Figures 1.2 and 1.3). The photos in each tiling were merged and small adjustments in their relative orientations were made so that a single hemispherical image was created for each node. Next, the hemispherical images were oriented in a global coordinate system. I helped in this process as the user who correlated at least six points in each image to features in a model [9]. I used the detailed model I had created from my work with Façade described in Section 2.3. The orientations and the surveyed locations were optimized and refined.

Currently the City Scanning Group is working to automate the steps used to acquire the *100 Nodes Dataset*. The Argus [11] pose-camera platform shown in Figure 1.4 determines

Figure 1.2: Selected photos of Technology Square from the *100 Nodes Dataset*.

Figure 1.3: Sample hemispherical tiling of images and a blended projected cylindrical image in which straight horizontal lines in the environment appears as arcs.

Figure 1.4: The Argus pose-camera platform estimates its location and orientation while acquiring images.

its position using Global Positioning (GPS) and navigation information. The imagery will also be tagged with information on the weather and lighting conditions and the day of year to indicate the level of obstruction due to seasonal foliage. The most difficult aspect of the image acquisition has proven to be correlating features during pose refinement without user assistance or a model of the structures in the environment [27].

1.2 Algorithms which Generate Building Fragments

The images and their refined pose data described in the previous section are used by programs under development in the group. Currently several different approaches to analyzing this photographic data are in use and are producing building fragments.

One method developed by J.P. Mellor [19] does a pixel-based analysis of the images using epipolar lines. An agreement of several images about the color or texture of a point in space could indicate the presence of the surface of an object at that point, or it could

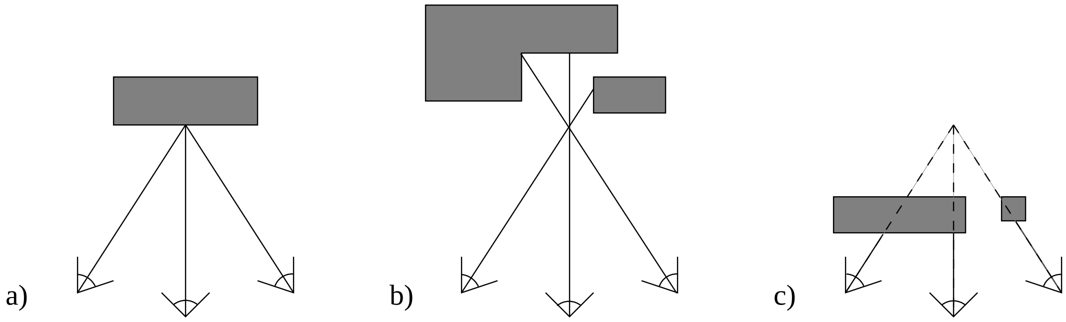


Figure 1.5: Illustration of how cameras that agree about the color or texture of a region of space could either indicate a) presence of a surface or a false correlation due to b) open space or c) occluders.

be a coincidental agreement as illustrated in Figure 1.5. An estimate of the normal is also known since the cameras that observed the surface must be on the front side rather than inside the object. Two drawbacks of this approach are that it is computationally expensive and does not yet fully account for lighting variations between photos.

A second approach, used by Satyan Coorg [8], interprets arcs which appear in the cylindrical images as straight horizontal segments in the environment (see Figure 1.3). The angle of each horizontal line relative to the global coordinate system can be determined uniquely. A sweep-plane algorithm is used on sets of similarly angled lines, and a plane position where many lines overlap suggests the presence of a vertical facade which is created by projecting the horizontal segments to the ground plane. The major limitation of this approach is that it reconstructs only vertical planes. While it works quite well for Technology Square and other urban buildings composed of facades perpendicular to the ground, it does not detect roofs, non-vertical walls, walls that do not meet the ground, or curved surfaces. The algorithm may not work well on buildings with less sharply contrasted horizontal edges or environments with false lines that, because they are assumed to be horizontal, may lead to spurious surfaces.

A third approach in progress by George Chou [6] extracts lines marking the edges of buildings, and through stereo image techniques determines their placement in the three-dimensional coordinate system. Points where several of the detected edges meet are identified as building corners. Since both Satyan Coorg's and George Chou's algorithms detect only planar building fragments, an algorithm like J.P. Mellor's, even though it requires more CPU time, is necessary to reconstruct more complex environments.

My program takes as input the fragments generated by these algorithms and is flexible enough to take other forms of data. Examples of other information that might be available for incorporation during aggregation include: existing survey data to give building corner locations; which direction is up; the approximate height of the ground plane; and a "depth map" acquired by a long range laser scanner (such as [24]) for each visual image. Each data input type requires a conversion matrix to situate the data in a common coordinate system and a file parser to load the information into one of the internal datatypes described in Section 3.1. It is important that my program combine data from many sources, as some algorithms may work better than others in certain environments. Also, some algorithms developed in our group may use early models of the environment to develop details for the final model, so the aggregation algorithm should be able to iteratively improve its current model of the environment.

1.3 Possible Uses

My program has an interactive graphical interface through which developers can view their data and merge it with other algorithm's results and models as these become available. It can also be used in a non-graphical batch mode. The final output of my program is a three-dimensional Inventor [25] model that can be viewed and manipulated inside standard CAD packages and rendered in real-time walk-throughs (such as the Berkeley WALKTHRU system [28]). The results could be presented in other formats; for example, it might be useful to have a simple two-dimensional representation of the model, similar to a city planner's map of the city with building cross sections and heights labeled, or a sparsely populated area could be drawn with contour lines to represent differences in terrain height.

The UMass Ascender project [1] is part of the RADIUS program [23] (Research and Development on Image Understanding Systems) which focuses progress and coordinates efforts of the many researchers working to solve problems in this area. RADIUS has divided the work into a few subproblems. One main objective is to assist image analysts who study aerial photographs in order to detect change and predict future activity at monitored sites [20]. Image analysts could be assisted by specialized algorithms that detect gradual change in the environment by comparing current photographs with an initial model. An application of the City Scanning Project would be to automatically create the models used in these systems. The aerial photographs used by the image analysts are tagged with camera pose, time of day, and weather information which matches the requirements of our system.

Future extensions to our system could analyze the surface textures of the final model. Simple estimates of the number of floors based on building height and window patterns or the square footage of a building could be computed. Texture maps obtained for the facades of buildings can be scaled, cropped, and tiled across other surfaces of the same or similar buildings if information is lacking for those facades. Analysis of the urban design is possible: Are there many similar rectilinear buildings arranged on a grid? Are buildings placed randomly? Does there seem to be a main road that the buildings are lined up along? Are there courtyards with trees, etc? The system could be extended to recognize objects in a scene or be tailored for use as a surveyor's tool.

Chapter 2

Related Work

Reconstructing three-dimensional objects from photographic images is a popular topic and many projects have been done and are in progress in the area. I will discuss some of the more relevant papers that outline methods for analyzing point cloud datasets and describe my experience with Raindrop Geomagic's Wrap software [29] which generates surfaces from point cloud datasets. I will also describe my experience with several semi-automated modeling packages for recovering three-dimensional architectural data from two-dimensional images: Paul Debevec's work at Berkeley on Façade [10], Shawn Becker's work at the M.I.T. Media Laboratory [3], and a commercial photogrammetry package, FotoG, by Vexcel [12]. All of these systems required large amounts of human interaction to generate the models. Also, the assumptions and specifications of these systems do not match our problem; they have special requirements and/or cannot take advantage of information present in our system such as camera placement and general building characteristics.

2.1 Point Cloud Reconstructions

Many algorithms exist for detecting surfaces in unorganized three-dimensional points (so-called point cloud datasets). Implementations of these algorithms provide impressive reconstructions of single non-planar surfaces or collections of surfaces, and some handle datasets with outliers (false correlations) and noise (imprecision).

Hoppe et al. [15] reconstruct a surface from unorganized points by first computing normals for each point using a least squares method with the nearest n neighbors. The corresponding tangent planes are adjusted (inside/outside) so that the surface is globally consistent. The algorithm relies on information about the density of surface samples (the maximum distance between any point on the surface and the nearest sampled point) and the amount of noise (maximum distance between a sampled point and the actual surface). Marching cubes [16] is used to create the final triangulated surface model.

An algorithm developed by Fua [13] first creates a point cloud dataset from multiple stereo images. Three-dimensional space is divided into equal cubic regions, and for each region with a minimum number of points a best fit plane is computed using an iterative re-weighted least squares technique. A small circular patch is created for each plane. Two neighboring patches are clustered into the same surface if the distance from each patch's center to the tangent plane of the other patch is small. Noise is dealt with by establishing requirements on the number of observations needed to create a surface patch. Erroneous correlations are handled by returning to the original images and projecting them onto the proposed surface. If several images disagree about the coloring of the surface then the chance of error is high. Reconstruction with this algorithm is not limited to single objects; sample test results for complex scenes are also provided.

Cao, Shikhande, and Hu [5] compare different algorithms to fit quadric surfaces to point cloud datasets, and for less uniform samples of data they found their iterative Newton's method algorithm superior to the various least squares approaches. McIvor and Valkenburg [17] [18] describe several methods for estimating quadric surfaces to fit point cloud datasets and compare their efficiency and results. They use different forms of quadric surface equations, finite differences in surface derivatives, and a facet approach to fit datasets of planes, cylinders, and spheres with noise but no outliers.

2.2 Raindrop Geomagic Wrap

Wrap [29] is a program developed by Raindrop Geomagic for generating complex three-dimensional models from point cloud data. The program comes with several sample datasets and provides an interface to many popular model scanning tools to collect the point cloud data. I used Wrap on both the sample datasets and on the synthetic point cloud data from an early version of J.P. Mellor's program described in Section 1.2. Wrap produced impressive three-dimensional models from the large 30,000 point sample datasets of a human head and foot. The surfaces were computed in about five minutes depending on the number of input data points.

To run the Wrap software on J.P. Mellor's point cloud dataset I arranged the data in an appropriate file format. Because the program incorporates all data points into the final shape, running Wrap on the original data produced an incomprehensible surface. I had to clean up the outliers and some of the noisy data points. I did not have any trouble doing this, but for more complex building arrangements it might be difficult to determine and select these points.

After these manual steps, I could run the wrap algorithm. I used a dataset of approximately 8000 points representing a portion of two buildings, two faces each. The initial model produced by Wrap is a single surface, so I used the *shoot through* option to break the model into the two separate buildings. The surface produced was very sharply and finely faceted because of the noise in J.P. Mellor's data. The sample files had produced very smooth surfaces, because those datasets had no outliers and zero noise.

Finally, I tried to simplify my surface; ideally I was hoping to reconstruct four bounded planar building facades. The *fill* option minimizes cracks in the surface, but the operation was very slow and the resulting surface was even more complex since it seemed to keep all of the initial surface geometry. A *relax* operation minimized the high frequencies in the surface, but over a hundred iterations (they recommend five to fifteen for most surfaces) were needed to create smooth surfaces. By that time, the corners of the building were well rounded. The *flatten* operation forces a selected portion of the surface to follow a plane which is uniquely specified as the user selects three points on the surface. This is not a practical solution since the user must select a portion of the surface (a non-trivial operation) and specify the points to calculate the plane.

While Wrap appears to be a great tool for general object modeling, the level of user interaction required is unacceptable for our particular problem, and the program does not support automatic culling of outliers or adapt well to noisy input data. Our problem allows us to generalize over data points that appear to lie in a single plane even if the data points vary from it.

2.3 Façade

Paul Debevec at Berkeley has developed a program called Façade [10] for creating models of buildings from a small number of images. The user first creates a rough sketch of the building from simple blocks. Some basic shapes are provided by the system, rectilinear solids, pyramids, etc., and the user can create and input additional arbitrary geometries. Each of these shapes is scaled and positioned in the scene. Next, the user loads the images into the system, marks edges on each image, and establishes correspondences between the lines on the image and edges of the blocks in the model. Then the user must approximately place and orient an icon representing the location from which the image was acquired. Several reconstruction algorithms work to minimize the difference between the blocks' projection into the images and the marked edges in the images. This is accomplished by scaling and translating the blocks and rotating and translating the cameras.

At first I had difficulty getting any useful models from Façade because the optimization algorithm would solve for the scene by shrinking all parameters to zero. This was because I had not specified and *locked* any of the model variables such as building size, position, and rotation or camera position and orientation, and the algorithm's optimization approach makes each dimension or variable a little smaller in turn to see if that helps error minimization. Even after I realized this and locked some variables, I still had many problems.

I thought perhaps Façade did not have enough information to solve for the model, so I loaded in more images. Unfortunately, this did not help very much; Façade was still confused about where the buildings were located on the site plan. Using only the photographic images, I was unable to guess building placement well enough to get good results. I then resorted to survey data the group had from an earlier project to build a model of the Technology Square buildings. Locking in these values, I was able to get more satisfactory results from Façade. In the end, I realized Façade is not as powerful as we had hoped. The sample reconstructions described in Debevec's paper [10] are for a single, simple building and a small portion of another building. Façade is not powerful enough to generate a model for several buildings in an arbitrary arrangement. Even though Technology Square has a very simple axis-aligned site plan with rectilinear buildings, Façade required much more additional data than had been anticipated. Also, the user must paint a *mask* for each image, to specify which pixels are unoccluded.

While working with Façade I realized how important it is to be able to *undo* actions (an option not available in the version I used) to go back and make changes in the specifications. I also saw just how difficult reconstruction can be due to under-specified and under-constrained model information or initial values that are not very near a globally optimal equilibrium.

2.4 Shawn Becker's work at the Media Laboratory

Another tool I used was a program written by Shawn Becker for his doctoral work [3]. This program has some ideas in common with Façade, but overall it has a different approach for the user interface. This system has the advantage of handling camera calibration within the algorithm automatically; the images were expected to be distorted and the calibration parameters would be solved for in the minimization loop.

The algorithm for computing a model from a single image is as follows: first, the user identifies lines in the image that are parallel to each of the three major axes and indicates the positive direction for each set of lines according to a right-hand coordinate system. Next, the user specifies groups of co-planar lines that correspond to surfaces in the three dimensional model. Finally, the user marks two features, the origin and one other known point in the three-dimensional coordinate system. From this information the system can generate a three-dimensional model of lines and features. I was impressed at how easily and accurately this model was generated.

Next, I tried to incorporate a second image into the model. Again, as with the first image, the user traces parallel lines in the image and specifies the co-planar lines as surfaces. When a building facade appears in both images, the lines from both images should be grouped as a single surface. It is necessary to specify two or more three-dimensional points in the second image and at least two of these points must also be specified in the first image. This is so the program can scale both images to the same coordinate system. Using the many correlations of planar surfaces and building corners, I was able to get a reasonably good reconstruction of the surfaces. However, the accuracy was rather poor. Lines on the building facade that had been specified in both images did not line up in the final model. I believe this inconsistency was due to low accuracy in the point feature placement. It was difficult to exactly specify the building corner locations because the images were slightly fuzzy and the feature marker is quite large making it difficult to place on the appropriate point. Misplacing the point by even one pixel can make a large difference when scaling images to the same size.

The program manual describes a method to minimize this inaccuracy by correlating edges between two images. I drew a line in each image connecting the tops of the same windows on a particular facade and tried to correlate these lines together; however, once I correlated a few edges the program seemed to ignore information I had specified earlier. The edges I linked were now drawn as a single line, but other building features were no longer modeled in the same plane. The program had failures in its optimization loop and did not handle feature placement noise.

2.5 FotoG

I also worked with FotoG [12], a commercial photogrammetry package developed by Vexcel for industrial applications. Pictures of complex machinery, such as beams, pipes, and tanks are processed as the user identifies correspondences between images and specifies a scale and coordinate system for the model. If enough points are specified the system can anchor each image's camera location. The model is sketched into a few images by the user, and the system helps locate more correspondences. The final result is an accurate CAD model of the objects texture-mapped with image projections.

I had a significant amount of trouble with FotoG's interface. The algorithm proceeds through a number of complicated steps, and unfortunately if you have a bad answer for any intermediate step you cannot proceed. It is the user's responsibility to verify independently the accuracy of these intermediate results. (In fact, the instructions for FotoG suggested printing out paper copies of every image and hand-rotating them in a complex manner to see if they were reasonable. I think the system could benefit immensely from some simple computer visualizations of these results.) I believe most of the errors arose when trying to calculate the camera position and orientation (pose) for each image. This was ironic because we had accurate estimates of the pose for the images in question. Unfortunately, the format used by FotoG for pose is different, and FotoG first operates in arbitrary local coordinate systems before moving into a common global system. These two factors made it very difficult to judge whether the intermediate values were reasonable.

FotoG incorporates many powerful algorithms into its system. It first establishes pair-wise relationships between images and then performs a global optimization of these relationships. In theory it can handle large numbers of images by computing and saving the results of small groups and then joining them to make a complete model. Unfortunately, it is not appropriate for our use. FotoG specializes in handling cylindrical pipes and other complex geometrical shapes. It also requires that the input be divided into groups of no more than thirty-two images, all of which show the same set of objects. I attempted to organize a set of our pictures which showed one facade of a building; however, in doing so I lost much of the needed variation in depth and it was impossible for FotoG to reconstruct the camera pose.

2.6 Common Difficulties

Many difficulties were common to each of these reconstruction efforts. Looking at the photographic images, it is sometimes nearly impossible to determine which facade of Technology Square is in the image since the facades of each building are so similar. It is difficult for a user who is unfamiliar with the physical structures to do correspondences, and likewise it would be very difficult to write a general purpose correspondence-finding program. Our system simplifies the problem by acquiring estimates of the absolute camera pose with the photographs, so the system will not need to guess how the photographs relate spatially to each other.

All the systems I tried seemed to struggle when noisy and erroneous information was present, or when little was known about the dimensions of the environment. By restricting the geometrical possibilities of our scene to buildings with mostly planar surfaces, we can reduce some of the complexity and be better able to deal with noise and outliers which could be prevalent in the building features produced by the algorithms of the City Scanning Project.

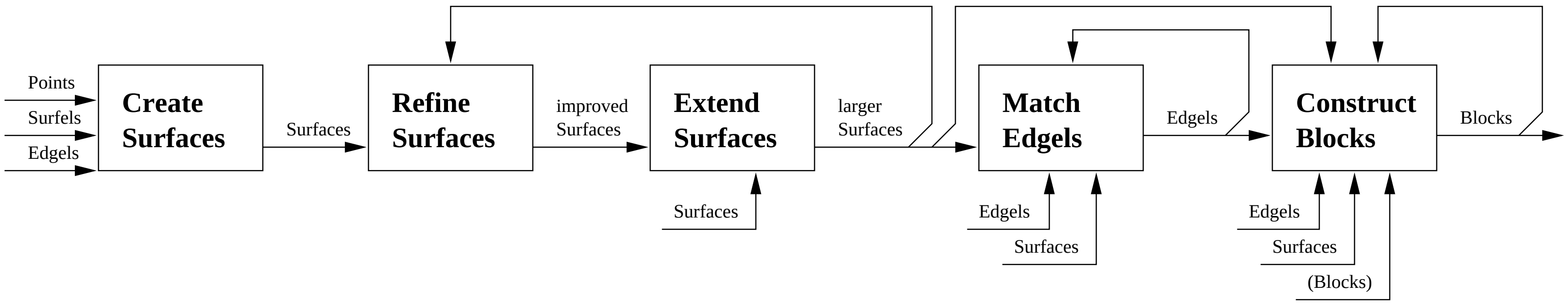
Chapter 3

Aggregation Algorithm

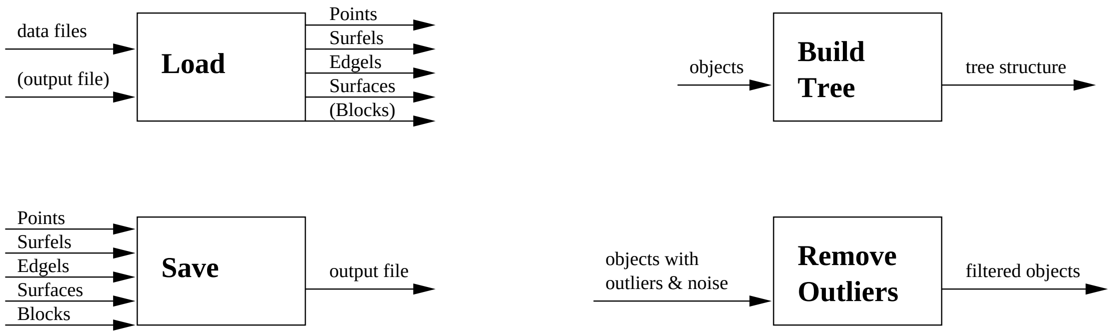
The building fragments generated from photographic imagery described in Chapter 1 are merged into a three-dimensional model by the aggregation algorithm. Figure 3.1 provides a high-level illustration of this algorithm. The data objects (described in Section 3.1) are loaded into the system by the Load module. New objects may be introduced at any time, and the results of a previous aggregation session (shown in parentheses) may be incorporated into the current aggregation. Section 3.2 explains the data structures that are used to efficiently gather neighboring objects. The phases and subphases of aggregation (described in Section 3.3) may be executed individually or iteratively to create surfaces and volumes that best represent buildings and other structures in the environment. Section 3.4 details the methods used for removing outliers and handling noisy data. The Load and Save modules which convert data files to objects, and objects to output files, are discussed in Section 4.5. The aggregation algorithm makes use of global parameters which are described in Appendix A.

3.1 Building Fragment Object Types

Building fragments are represented by five types of objects in the system: Points, Surfels, Edgels, Surfaces, and Blocks. The first four types can come directly from instrumentation on the acquisition cart (for example, point cloud datasets can be acquired with a laser range scanner) or as output from the building fragment generating algorithms of the group described in Section 1.2. Intermediate results of the system are Edgels, Surfaces, and Blocks and the final model output by my program is composed of Blocks which represent the solid



a) The phases of the aggregation algorithm may be performed repeatedly as new objects become available.



b) The file I/O options, Load and Save, and the Build Tree and Remove Outliers operations may be performed at any time.

Figure 3.1: Diagram of the aggregation algorithm.

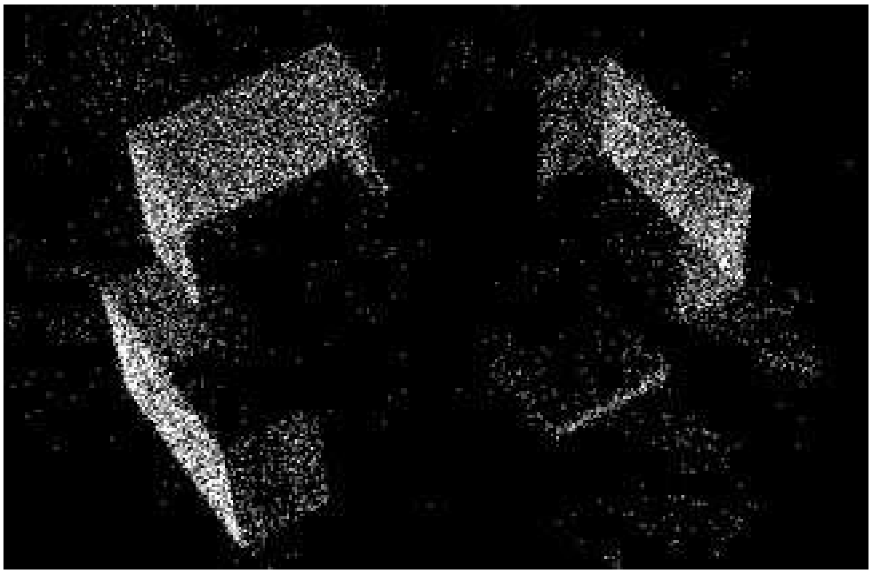


Figure 3.2: Image of approximately 1% of the Points produced by J.P. Mellor's program from the *Synthetic Image Dataset*. Because the cameras for this image dataset were located in a circle around the buildings (see Figure 1.1), nearly all of the Points are on the outwardly facing facades.

spaces occupied by buildings and other structures in the environment. If needed, extensions to each type as well as additions of new types can be handled gracefully by the system. For example, known ground heights, camera locations, or camera paths might need special representations in the system. Section 4.1 in the Implementation Chapter describes the class hierarchy and the fields and methods for each type of object.

3.1.1 Points

Many model reconstruction algorithms begin only with a set of three-dimensional points located in space; likewise, my algorithm can process this type of input data. An early dataset produced from the *Synthetic Image Dataset* by J.P. Mellor's algorithm is the set of Points shown in Figure 3.2.

The Points from J.P. Mellor's dataset each have an estimate of the normal of the surface at that point which he claims are accurate within 30 degrees. The normal is only used to determine the final orientation of the surface (pointing "in" or "out") once it has been recovered from the point cloud. Points are not required to have a normal estimate, but an approximation based on the location of the camera(s) or scanner that observed this data is usually available (see Figure 3.3). If input point cloud datasets have very accurate

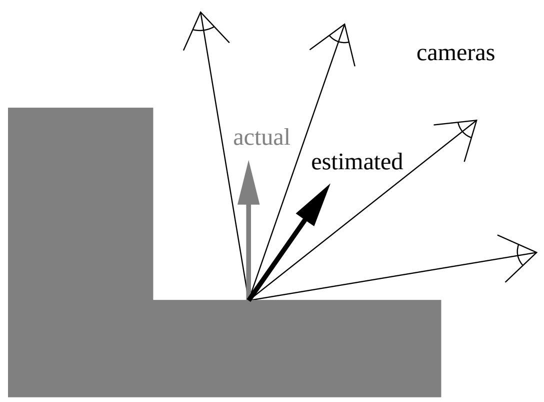


Figure 3.3: Diagram of a Point on the surface of a building with a rough estimate of the surface normal based only on the cameras that observed the point.

normals, techniques similar to those used for Surfels introduced in Section 3.1.2 could take advantage of this information.

The group has plans to mount a long-range laser depth scanner [24] on the Argus acquisition cart [11] which would be able to take "depth map" images simultaneously with the photographic images. These depth maps would provide large quantities of Points with low-accuracy normals. For algorithm testing I can generate synthetic point cloud datasets from Inventor models to simulate laser scanner data (see also Section 4.6).

Challenges in dealing with point cloud datasets include:

- Managing tens of thousands of Points or more.
- Rejecting the outliers and false correlations and minimizing the effects of the noisy data points.
- Finding planes or more complex surfaces which best fit the data.
- Adjusting for a varying density of the samples (more samples closer to the ground, fewer samples behind trees or in tight alleys, etc).

Storing objects in the hierarchical tree structure described in Section 3.2 makes it much more efficient to work with large numbers of Points. Also, by randomly sampling the Points as they are loaded and/or sampling the set of Points in a leaf node, we can work with fewer objects at a time. Techniques for creating surfaces from Points are described in Section 3.3.1.

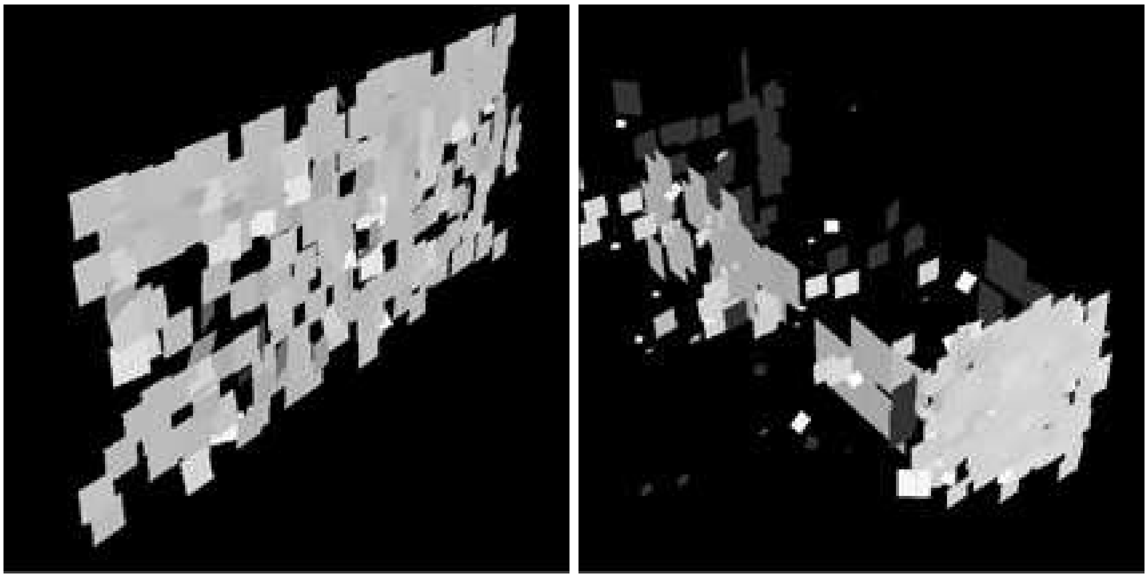


Figure 3.4: Images of Surfels produced by J.P. Mellor's algorithm from the *100 Nodes Dataset*. The left image shows the reconstruction of a single facade, and the right image shows Surfels of varying sizes produced for a rectilinear building in Technology Square.

3.1.2 Surfels

In his more recent work with photos from the *100 Nodes Dataset*, J.P. Mellor produces Surfels shown in Figure 3.4, which are bits of building surface about ten feet square (but this size may vary within the dataset). He estimates the normal of these surfaces to be accurate to within a couple of degrees. The normal accuracy is much more important with the Surfel dataset than with the Point dataset because the algorithm produces far fewer Surfels than Points. Each Surfel has a small planar Surface to represent its area.

Challenges in dealing with the Surfel dataset include:

- Rejecting outliers and noisy Surfels. In initial datasets, over 50% of the Surfels were outliers (see Figure 3.24), though some of this outlier culling is being done in more recent versions of J.P. Mellor's program.
- Eliminating fencepost problems, where the algorithm is fooled into false correspondences by repeated building textures (see Figure 3.5).
- Varying density of the samples.
- Taking advantage of the accurate normals while joining Surfels into larger surfaces which may be planar, spherical, cylindrical, etc.

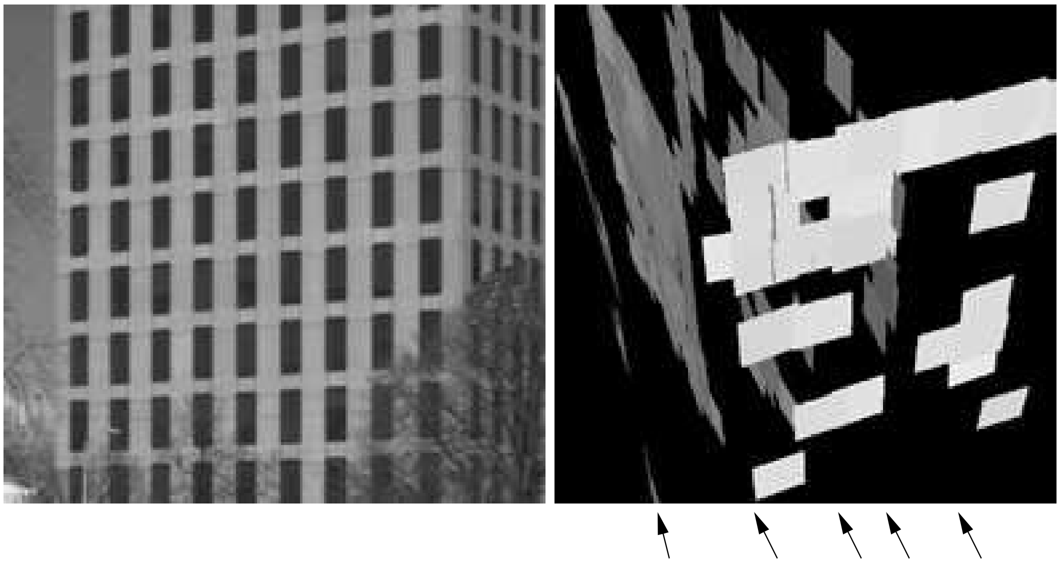


Figure 3.5: Repeated building textures can cause false correlations, a stereo vision fencepost problem. In the reconstruction shown from an early version of J.P. Mellor's algorithm, Surfel evidence at five different plane depths is available for the building facade in the background.

The methods for removing Surfels arising from false correlations are described in Section 3.4.4. The techniques for surface creation give special weight to Surfels based on their area and normals.

3.1.3 Edgels

Both Satyan Coorg's and George Chou's algorithms produce line segments which are represented as Edgels in my program (see Figures 3.6 and 3.7). Some of these segments represent geometrical building edges, but many are just features (window and floor delineations) from the texture of the buildings. Edgels are used to neatly border surfaces and indicate where shallow bends in large surfaces should occur. They are also generated by my algorithm at the intersection of two planar surfaces.

Challenges in dealing with Edgel datasets include:

- Assigning appropriate weights to each Edgel.
- Identifying spurious Edgels which may bend and crop surfaces incorrectly.
- Making full use of Edgels even though they do not inherently contain any information about the orientation of the surface(s) on which they lie.

Satyan Coorg's method for obtaining Edgels begins with line detection from pho-

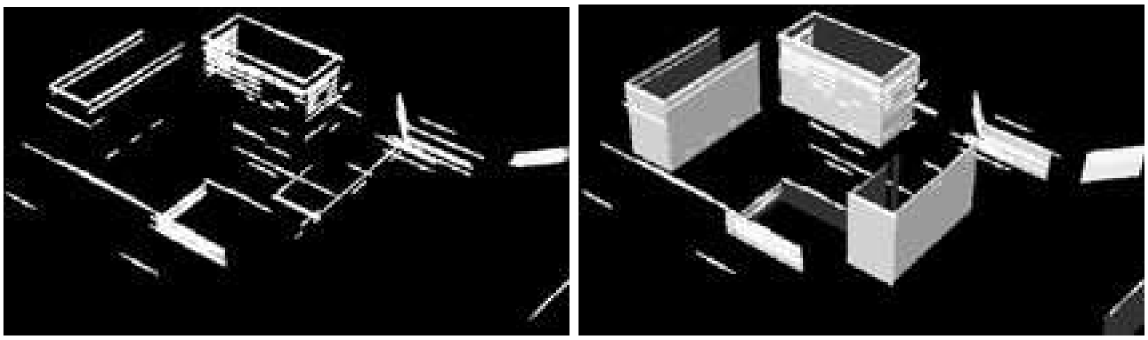


Figure 3.6: Horizontal Edgels identified by Satyan Coorg's algorithm from *100 Nodes Dataset* and the vertical Surfaces created by projecting these line segments to the ground.

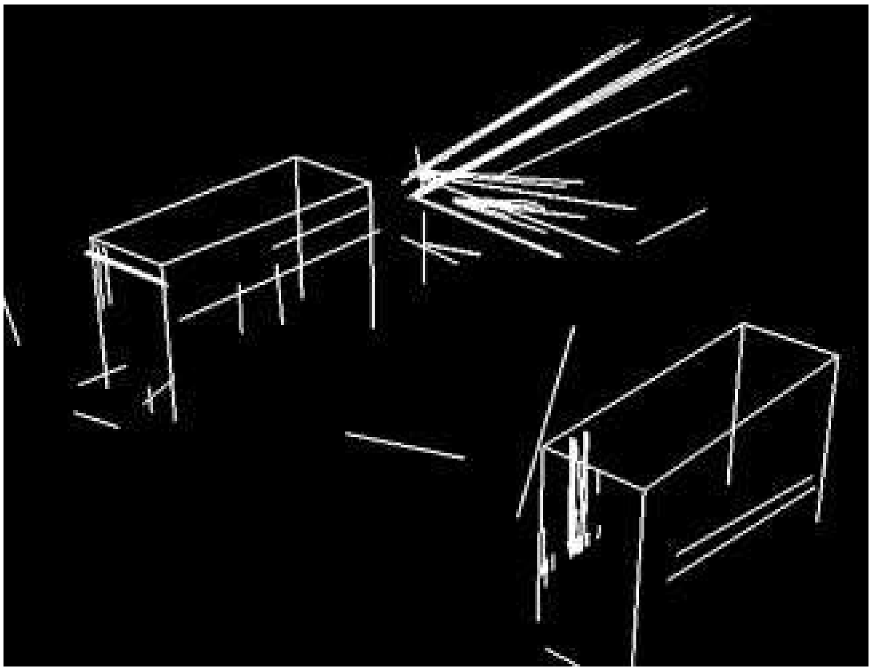


Figure 3.7: Edgels detected by George Chou's algorithm. Spurious Edgels are created when false correspondences are made, but when few images agree with a produced Edgel, its certainty value is low.

tographs and produces possibly hundreds of Edgels per building, while George Chou's algorithm produces a much smaller set of Edgels which are averages of many lines detected in the images. The input file parsing modules for each file type must assign reasonable *certainty* values for Edgels (described in Section 3.4.2) so they can be compared appropriately during the **Create Surfaces**, **Match Edgels**, and **Construct Blocks** phases of aggregation described in Sections 3.3.1, 3.3.4, and 3.3.5.

3.1.4 Surfaces

Surfaces are finite, bounded portions of two-dimensional manifolds representing building facades, roofs, columns, domes, etc. Satyan Coorg's algorithm projects clusters of horizontal segments to the ground plane, producing simple rectangular building facades (see Figure 3.6) which are represented as Surfaces in my program. Surfaces are generated by my algorithm to fit clusters of Points, Surfels, and Edgels. For algorithm testing I create synthetic Surface datasets by sampling surfaces in Inventor models (see Section 4.6).

Planar Surfaces are modeled with the equation $ax + by + cz + d = 0$, which has three degrees of freedom (DOF), and a chain of Edgels to delineate the boundary. Spherical Surfaces are represented with the equation $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$, with four DOF. The system can be extended to include cylindrical Surfaces modeled with two points p_1 and p_2 (representing the endpoints of the cylinder's axis) and a radius r having a total of seven DOF. Constraints on how objects appear in the urban environment can be used to decrease the number of DOF needed to model cylinders. For example, vertical cylinders can be modeled with just five DOF (base point, height, and radius). The system can also represent other types of non-planar surfaces, and clip planes or curved Edgels to specify that only a portion of a sphere, cylinder or other curved surface is present in the environment.

Challenges in dealing with Surface Datasets include:

- Discarding very small surfaces and dealing with improper surfaces (those with negative or zero area, or non-convex polygonal outlines).
- Inaccurate normals from poor point cloud surface fitting.
- Ragged edges bounding surfaces generated from clusters of building fragments.

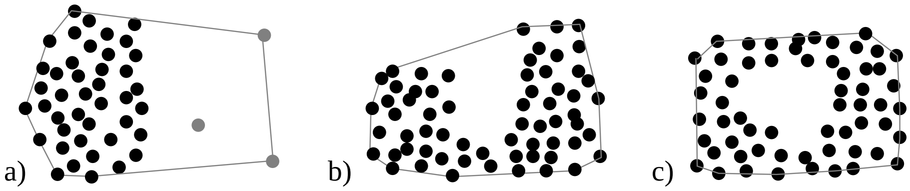


Figure 3.8: Examples of poorly bordered surfaces: *a*) Surface which is too big for the evidence because of outliers (shown in gray), *b*) Points which would fit best to a non-convex surface, and *c*) a set of Point data with a hole in it (the actual surface may or may not have a hole in it).

- Determining if a surface has a hole in it (a hole in the data sampling can indicate either a hole in the surface, or the presence of an occluder such as a tree in front of the surface).
- Detecting and correcting Surfaces that are much larger than the majority of the evidence suggests because outliers were included during their construction.
- Varying levels of success in reconstructing surfaces due to an uneven distribution of data samples and variations in data accuracy.
- Modeling non-convex building facade silhouettes.

The tree data structure (described in Section 3.2) organizes objects by location. Data fragments with few neighbors will not be used to create new Surfaces. The smaller the leaf nodes of the tree, the less likely we are to encounter the oversized surface problems shown in Figure 3.8. Surface density and surface fit criteria (described in Sections 3.4.2 and 3.4.3) are used to identify surfaces which need to be refined or removed. Also, surfaces with very small area relative to the global parameter `average_building_size` can be removed (see also Section 3.4.1). I have chosen to represent only convex surfaces for simplicity of some operations. The system can be extended to manipulate non-convex Surfaces by internally representing them as several smaller convex surfaces.

3.1.5 Blocks

Blocks represent buildings and other solid structure in the urban environment and are the final output of the system. I represent a Block as a set of Surfaces which define a closed region of space. The Surfaces are connected to each other through their Edgels in a half-edge data structure which is sketched in Figure 3.9. This structure can represent convex

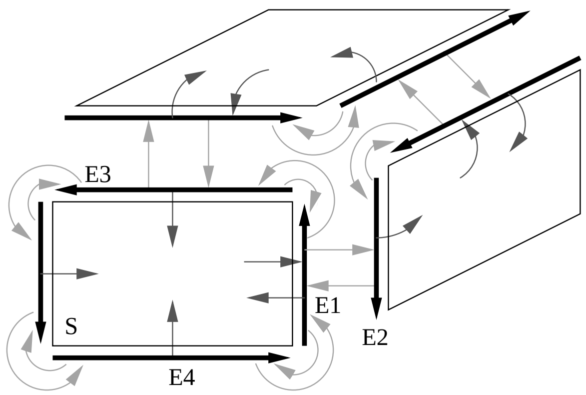


Figure 3.9: The half-edge data structure provides a consistent method for connecting Edgels and Surfaces into a Block. Every Edgel has a pointer to its twin, next, and previous Edgels and its surface; for example, Edgel E1 points to E2, E3, E4, and S respectively. Each Surface has a pointer to one of its bordering Edgels; in this example, S points to E1.

and concave buildings. The vertices may join more than three surfaces.

Challenges in producing and manipulating Blocks include:

- Computations with non-convex polyhedral buildings.
- Altering the dimensions and structure of a Block to adapt to new data.
- Modeling buildings composed of both planar and non-planar (spherical, cylindrical, etc.) Surfaces.

Non-convex Blocks are manipulated internally as several convex blocks to make the intersection operations simpler. Each Surface and Edgel that is part of a Block maintains a list of evidence (if any) that led to its creation. If new evidence is added which may modify the conclusions drawn from the original data, the Block is recomputed.

3.2 Build Tree

An enormous amount of data exists within a city, and each building can be sampled to an arbitrary level of detail. Whether we are reconstructing just one building, one block of buildings, or an entire city, we must use a hierarchical system to manage the data. All objects are stored in a three-dimensional *k-d tree* organized by location to allow fast calculation of each object's nearest neighbors [4]. A k-d tree is a binary tree that is successively split along the longest of its k dimensions resulting in roughly cube-sized leaf nodes. The tree is recomputed automatically as necessary: when new objects are added outside the bounding box of the current tree, when the data has been modified (to focus on a region of interest, or to remove outliers, etc.), or when global tree parameters have been changed.

When the **Build Tree** command is performed, either automatically or as requested by the user, any existing tree structure is deleted, a new empty tree is created, and all current objects are added to the tree. The bounding box of this new tree is set to be 10% larger in each dimension than the bounding box of all the objects in the scene. A reconstructed object may have a bounding box which is larger than that of its evidence; the 10% buffer reduces the chance that we will have to recompute the tree because a newly created object's bounding box is also larger than the current tree.

3.2.1 Tree Size and Structure

The maximum number of nodes in a binary tree is $2^n - 1$, where n is the maximum depth. The size of the tree (number of nodes) is critical for performance of the searching operations. A large tree with many leaves will take longer to build and maintain, while a small tree with fewer leaves will require more linear searching through the multiple elements of the leaf nodes. In order to choose an appropriate tree depth, the program needs information from the user about the base unit of measurement of the input data and the desired level of detail for the reconstruction (a single building, a city block of buildings, or an entire city). This information is conveyed with two global parameters: `average_building_side`, which specifies the dimension of an average size building, and `nodes_per_side`, which specifies the minimum number of leaf nodes to be used per average building length. For example, if the `average_building_side` is 500 units and the user requests 2.0 `nodes_per_side`, the program will divide a scene of size $1000 \times 1000 \times 500$ into 32 leaf nodes as illustrated in Figure 3.10. The necessary tree depth is computed as shown below, by totaling the number of times each dimension d should be split, where Δd is the size of the tree bounding box in that dimension.

$$\text{tree depth} = 1 + \sum_{d=X,Y,Z} \max\left(\left\lceil \log_2 \frac{\texttt{nodes\_per\_side} \times \Delta d}{\texttt{average\_building\_side}} \right\rceil, 0\right)$$

The user can also specify how each intermediate node of the tree should be split: at the middle of the longest dimension or at a randomized point near the middle of the longest dimension (see Figure 3.11). The randomized option simulates tree structures which might be created by a more complicated splitting function implemented in the future that

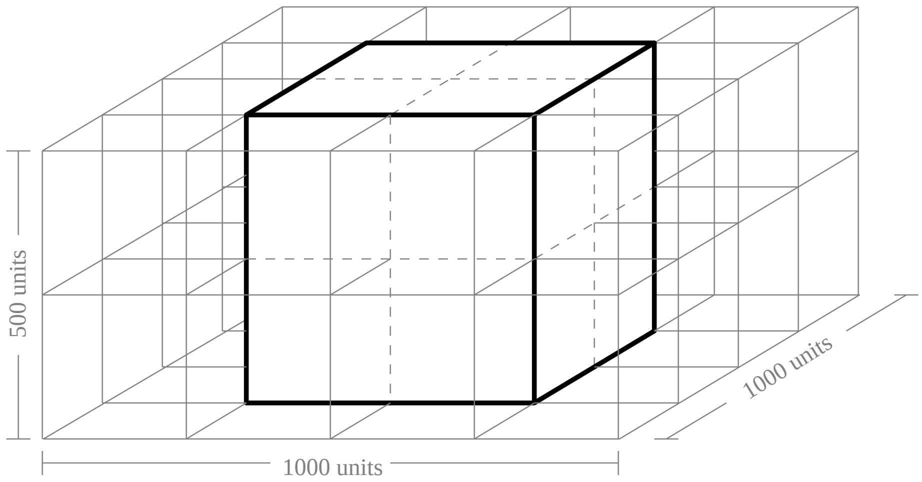


Figure 3.10: A scene of size 1000 x 1000 x 500 units that has been evenly divided into 32 leaf tree nodes based on the parameters: `average_building_side` = 500 units and `nodes_per_side` = 2.0. The tree nodes are drawn in grey, and a cube representing an average building is drawn in black.

maximizes the separation of the objects of the node while minimizing the number of objects intersected and required to be on both sides of the split.

The **Create Surfaces** step of the aggregation algorithm (see Section 3.3.1) is the only phase whose result is directly dependent on the size and structure of the tree. The **Refine Surfaces** step may further subdivide some leaf tree nodes; the maximum number of additional subdivisions allowed is adjusted by the parameter `additional_tree_depth` (see Section 3.3.2).

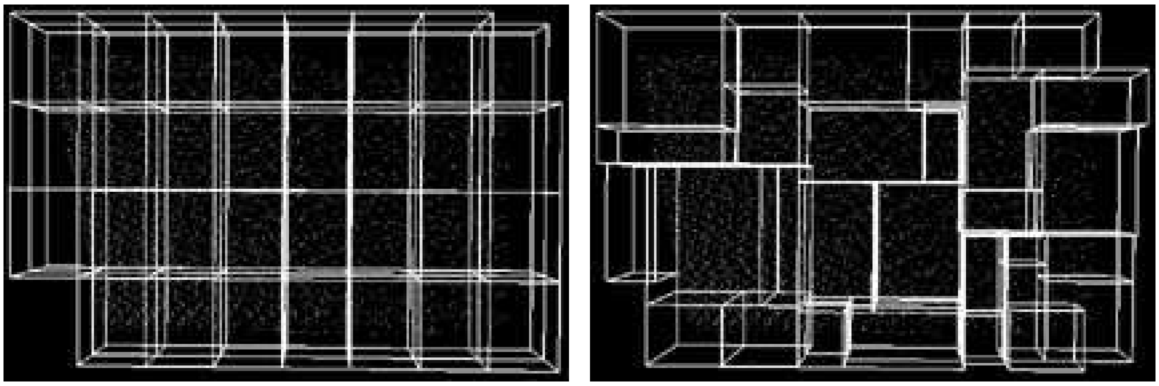


Figure 3.11: Point cloud data set displayed within an evenly split tree and a randomly split tree.

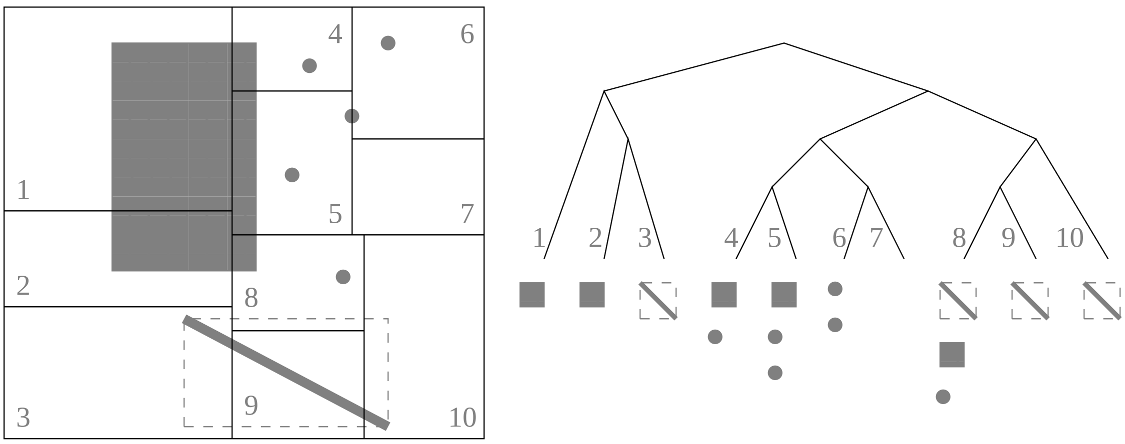


Figure 3.12: Two-dimensional example of how objects are stored in a tree structure. The tree nodes are split at a random point along their longest side until either they have only one object (as for node 1) or the length of each side is small enough (as for node 6). Each leaf node stores the set of all objects whose bounding box at least partially overlaps the node's bounding box. Note that the Edgel is stored in node 8 because its bounding box overlaps the tree node even though the Edgel does not pass through that node. A less conservative method could be used to determine the elements of each leaf node.

3.2.2 Retrieving Nearest Neighbors

Each leaf node of the tree contains a list of all building fragments whose bounding box fully or partially overlaps the tree node (see Figure 3.12). Using this structure, objects may belong to more than one leaf node. When gathering objects from the tree or walking through the tree to perform operations on each element, one must be careful not to double count or process these objects.

Duplicate counting and processing can be avoided by creating a list and verifying with a linear search that each element added is not already in the list. This operation becomes inefficient quickly, $O(n^2)$, when creating large unsorted lists of objects. A better choice is to tag elements as they are added to the list and check for that tag when adding new elements, $O(n)$. Also, instead of constructing a list of all the objects (which is costly for large lists of elements), the tree structure can be *walked*, tagging objects as they are processed. See the Implementation Chapter, Section 4.3 for more details on tagging objects and walking through the tree structure.

The tree structure makes it efficient to collect all objects in a region of space that meet some specified criteria such as object type, whether it was a data or reconstructed element, etc. For example, to conservatively find the nearest neighboring surfaces of an object, we take that object's bounding box and extend it in each dimension by the *relative error distance* which is the minimum of the size of the object and the global parameter `average_building_side`. Then we retrieve all leaf nodes of the tree which contain part of the extended bounding box. And finally we identify all Surfaces (being careful to avoid duplicates as described above) in each of these nodes. We are guaranteed to find all objects that could be within the relative error distance from the initial object, but we will probably return others which will be removed by subsequent operations as necessary. With smaller leaf nodes, fewer non-neighbors are returned.

3.3 Phases of Aggregation

The following sections explain the five phases of the aggregation algorithm (shown in Figure 3.1a). These phases may be executed individually by the user through the graphical user interface (described in Appendix B), which has a button for each phase. A sequence of phases to be executed may be specified in a command script (described in Appendix C). The single command **Aggregate All** executes all five phases.

Surfaces are created and manipulated with the operations **Create Surfaces**, **Refine Surfaces**, and **Extend Surfaces**. These Surfaces are intersected and combined with Edgel data by the operations **Match Edgels** and **Construct Blocks**. The intermediate and final results of these steps can be saved for later aggregation sessions or for viewing with other modeling or visualization programs. By default all subphases of the five phases are executed in the order described below; however, each subphase may be disabled (as described in Appendices B and C) for faster processing (for example, it may be known that there are no quadric surfaces) or for inspection of intermediate results. The final results are not dependent on the order of phase or subphase execution, but it is more efficient to run certain phases first (for example the subphase Extend Excellent Surfaces should be run before Extend All Surfaces).

The phases of aggregation create new objects called *reconstructed* objects which can be modified unlike *data* objects which are never modified. Data objects are deleted only if they are malformed (too small or non-convex, etc.), are outliers as detailed in Section 3.4, or are outside the region of interest. Each reconstructed object stores pointers to its *evidence* (objects that were used to create it) and each object stores pointers to any reconstructed objects to which it *contributes.* Section 3.4 details several important metrics: *relative weight*, *evidence fit*, *surface density*, and *surface fit*.

3.3.1 Create Surfaces

In this phase, the program creates Surfaces to best fit Point, Surfel, and Edgel data which has been sorted into tree nodes. The computation focuses on Points, but Surfels and Edgels are incorporated with special weights. For leaf nodes with a minimum number of data objects (specified by parameter `tree_count`), the algorithm computes a best fit plane using an iterative least squares technique similar to that used by Fua [13] and Hoppe et al. [15]. The **Extend Surfaces** phase will fit objects to non-planar Surfaces as appropriate.

Fitting a Plane to a collection of Points

Each iteration of the plane fitting algorithm proceeds as follows: The centroid $C = (x_c, y_c, z_c)$ is calculated to be the weighted average of the Point positions $P_i = (x_i, y_i, z_i)$. For the first iteration w_i (the weight of Point i) = relative weight$_i$. In subsequent iterations w_i = relative weight$_i$ $\times$ evidence fit$_{i\ to\ p}$, where p is the best fit plane from the previous iteration. The metric *relative weight* is described in Section 3.4.2 and *evidence fit* is described in Section 3.4.3. The n positions P_i are gathered into a 3×3 matrix M:

$$M = \sum_{i=1}^{n} w_i((P_i - C) \times (P_i - C))$$

The values for the coefficients of the plane equation $ax + by + cz + d = 0$ are calculated by finding the eigenvector associated with the largest real eigenvalue λ of matrix M:

$$M \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \lambda \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

The positions P_i were translated by the centroid C so the best fit plane passes through the origin $(0, 0, 0)$ and the coefficient d for the calculated plane is zero. d is calculated for the original positions as follows:

$$d = -(ax_c + by_c + cz_c)$$

The above steps are shown in Figure 3.13 a, b, and c. The number of iterations performed can be adjusted by the global parameter `surface_iterations`. Fua [13] found that three to five iterations were optimal.

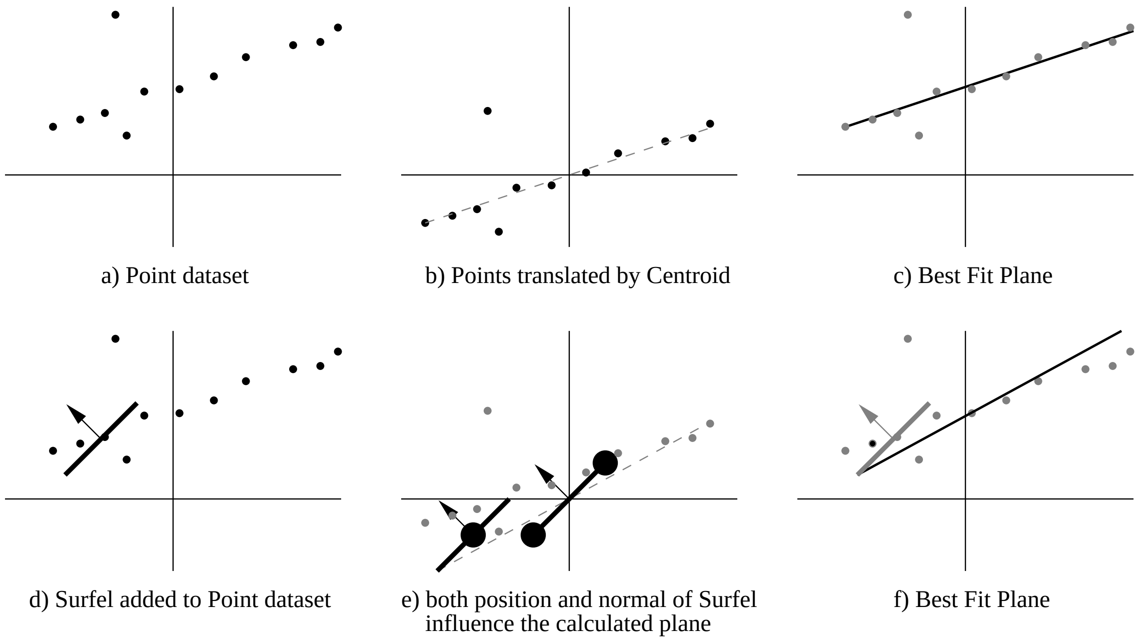


Figure 3.13: Two dimensional example of how to fit a line to a Point and Surfel dataset. a and d show the original objects. In b and e the objects are translated so their centroid is at the origin and c and f show the final planes. In e, the Surfel is shown translated in two ways: first it is translated with the other objects and is drawn with a large black dot for the center of mass, the other is drawn at the origin and shows two black dots which influence the normal of the final surface. Note that the Surfel's normal does not fit the plane very well, so in subsequent iterations the Surfel will be given less influence over the plane created.

Incorporating Surfels and Edgels

A Surfel's position can be treated like a Point in the above algorithm, but its normal needs special consideration. In addition to adding the Surfel's position into the matrix M we add four additional points each one unit away from the origin in the plane perpendicular to the normal.

$$\begin{aligned} P_0 &= \text{Surfel center of mass} \\ \vec{v_1} &= \text{any unit vector} \perp \text{to Surfel normal} \\ \vec{v_2} &= \text{the unit vector} \perp \text{to Surfel normal and } \vec{v_1} \\ P_1 &= \vec{v_1} \\ P_2 &= \vec{v_2} \\ P_3 &= -\vec{v_1} \end{aligned}$$

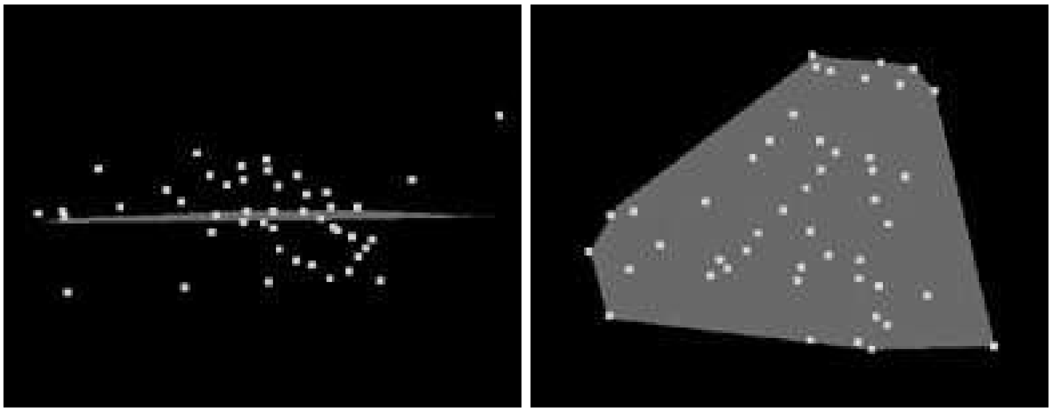


Figure 3.14: A collection of Points from a node of the tree and the convex hull of each Point's projection onto the best fit plane.

$$
\begin{aligned}
P_4 &= -\vec{v_2} \\
w_0 &= \frac{\text{relative weight}_{Surfel}}{2} \\
w_1 = w_2 = w_3 = w_4 &= \frac{\text{relative weight}_{Surfel}}{8}
\end{aligned}
$$

For example, if the Surfel's normal is (0, 0, 1), we add the points (1, 0, 0), (0, 1, 0), (-1, 0, 0), and (0, -1, 0). A two-dimensional example is shown in Figure 3.13 d, e, and f.

Edgels are incorporated into the plane computation described above by simply adding the endpoints of the Edgel to matrix M:

$$
\begin{aligned}
P_1 &= \text{Edgel endpoint}_1 \\
P_2 &= \text{Edgel endpoint}_2 \\
w_1 = w_2 &= \frac{\text{relative weight}_{Edgel}}{2}
\end{aligned}
$$

Determining a Boundary

Once a plane has been estimated for each tree node having a minimum number of data elements, a bounding outline for the Surface is found by projecting the elements of the tree node onto the new plane and computing the convex hull of those two-dimensional points as shown in Figure 3.14. Objects that poorly fit the computed plane are not used in computing this boundary. The final orientation of the Surface (which side is the "front") is determined by computing the weighted average of the normals of all evidence contributing to the surface.

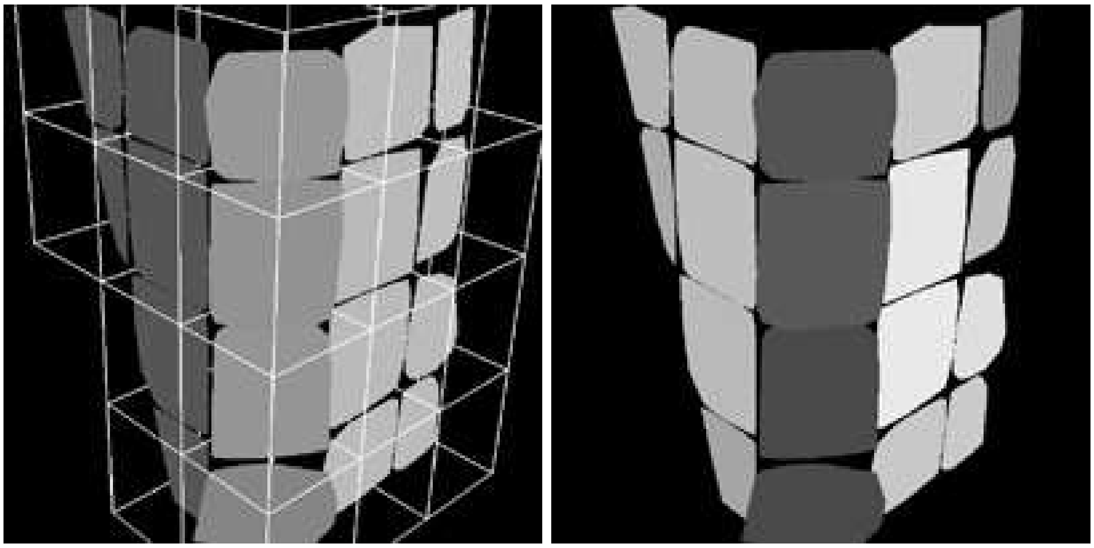
3.3.2 Refine Surfaces

Create Surfaces, the initial phase of aggregation, groups data objects by proximity without any knowledge of their relationship in the real environment. For this reason, some of the resulting surface patches will not be the best fit surfaces to those data elements. The two subphases of **Refine Surfaces** attempt to improve reconstructed Surfaces by changing the set of objects used to calculate each surface. Reconstructed Surfaces are judged using two metrics *surface density* and *surface fit* which are described in Sections 3.4.2 and 3.4.3. Both metrics range from 0.0 (low quality Surface) to 1.0 (high quality Surface).

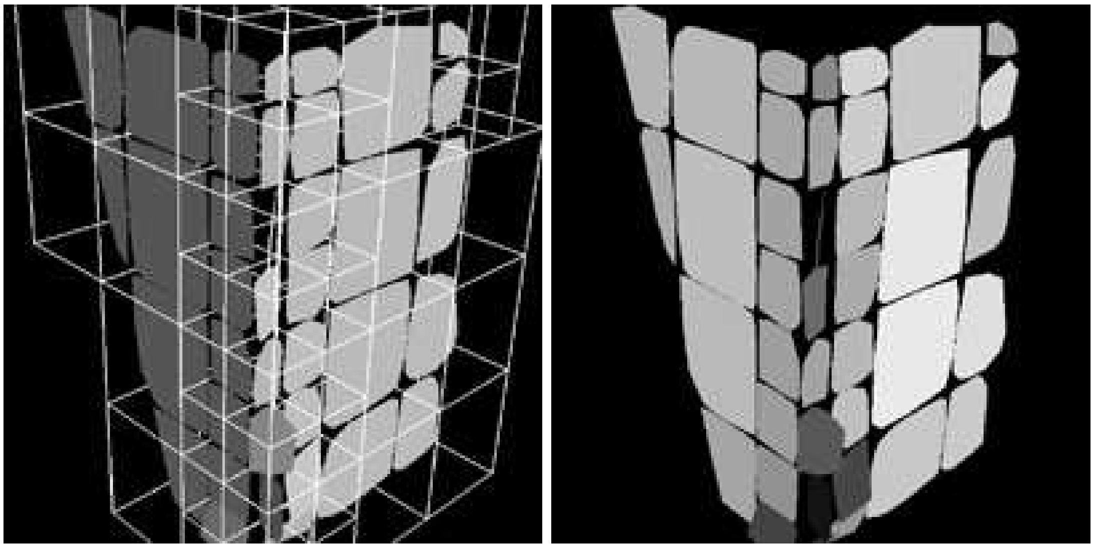
The first subphase of surface refinement, Subdivide Tree, identifies Surfaces created from the data elements of leaf tree nodes that have poor surface fit (< 0.5) but high surface density (> 0.5). Further subdivision of each Surface's corresponding tree node can help separate building fragments which belong to different facades. The parameter `additional_tree_depth` specifies the maximum number of levels of further tree subdivision to be used while attempting to separate Points which lie on different surfaces. Figure 3.15 shows Points from two planar facades which were grouped together in the initial tree structure.

The second subphase of surface refinement, Use Best Fit Data, filters the collection of evidence for Surfaces with high surface fit (> 0.7) and high surface density (> 0.5) values. After filtering out data which least well fits the Surface (objects whose evidence fit value, described in Section 3.4.3, to the surface is < 0.5) the Surface is recomputed; if not enough evidence remains, the Surface is deleted. By releasing poorly matched data objects, other neighboring Surfaces to which the data fits more closely will be able to use this data to improve surface fit and extend their boundaries (see Figure 3.16 and Section 3.3.3).

Not all Surfaces with high surface density values emerge from the **Refine Surfaces** operation with a high surface fit value. There are several reasons this phase of aggregation may not improve surface fit. A high density of outliers or the presence of a complex structure, such as a tree, in the environment could result in an erroneous yet dense surface. A dataset with a high level of noise viewed at close range (for example, a leaf node of size d^3 within a point cloud dataset that has many Points distance d from the surface) can result in a meaningless plane estimate. Also, while most buildings are composed of rectilinear solids, some urban environments contain large non-planar structures. To adequately reconstruct these types of structures, the **Extend Surfaces** phase will also attempt to fit the data

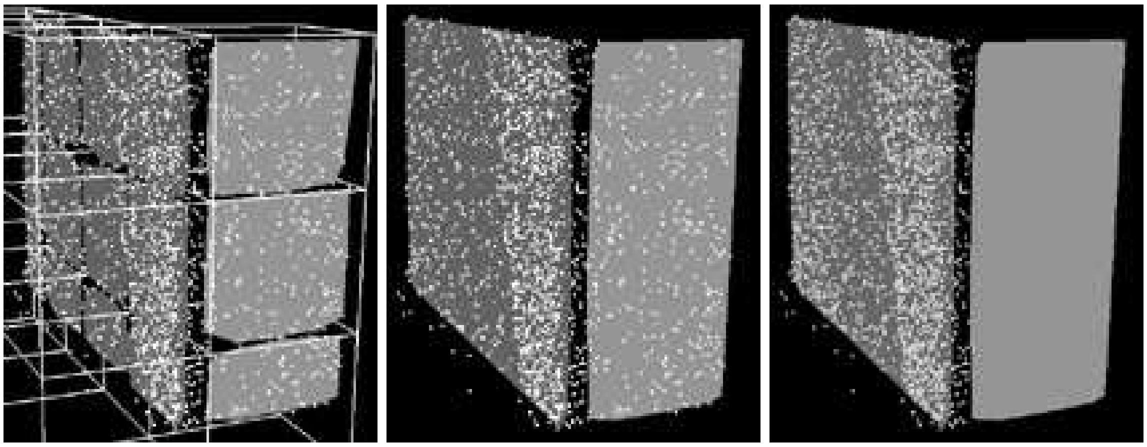


Initial Surfaces fit to point cloud dataset of two distinct facades.

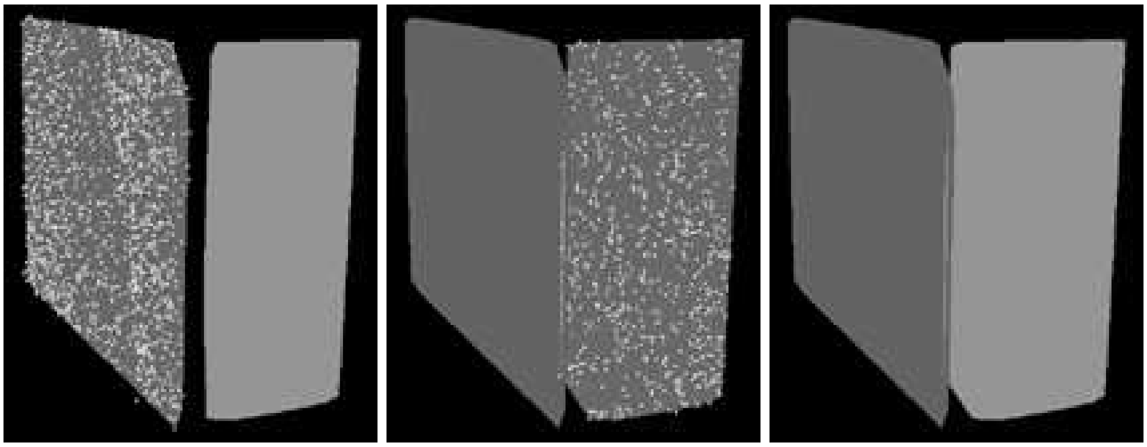


After the Subdivide Tree subphase of **Refine Surfaces**.

Figure 3.15: The initial Surfaces suggest planar facades at three positions. After subdividing leaf nodes with poorly fit Surfaces, the "average" surface disappears. The left images are colored by normal, the right images are shaded by surface fit (lighter Surfaces have higher surface fit).



a) initial Surfaces | *b*) Extend Excellent Surfaces | *c*) evidence for left Surface



d) Use Best Fit Data | *e*) Unclaimed Data | *f*) final Surfaces

Figure 3.16: Sample reconstruction of Point dataset which uses the Use Best Fit Data subphase of **Refine Surfaces** and the Unclaimed Data subphase of **Extend Surfaces**. The original tree structure shown in *a*) groups some of the Points which belong to the right facade with Points from the left facade shown in *b*) and *c*). After the left Surface releases the data Points which do not closely fit the Surface, shown in *d*), the right surface can use the Unclaimed Data to extend its boundary, as shown in *e*) and *f*).

elements to simple quadric surfaces.

3.3.3 Extend Surfaces

Once the program has generated many small planar surface patches with the commands **Make Surfaces** and **Refine Surfaces**, they need to be joined with neighboring patches to make larger surfaces.

Creating Large Planar Surfaces

The first two subphases, Extend Excellent Surfaces and Extend All Surfaces join neighboring surfaces to make larger planar surfaces. Extend Excellent Surfaces only attempts to join the "best" Surfaces and therefore can use a joining method which is more efficient. A new Surface will be created for two or more neighboring planar surfaces if:

- Each Surface is a data object or a reconstructed object with high surface density (> 0.5) and high surface fit (> 0.5) which are described in Sections 3.4.2 and 3.4.3.

- The surfaces when compared to each other lie in nearly the same plane as described in Section 3.4.4.

The new Surface is computed with the algorithm in Section 3.3.1 using the evidence from the original surfaces if they are reconstructed, or simply the original surface if it is a data object. Once joined the following properties must hold or the new Surface is deleted:

- The new Surface's density and fit values are approximately the same or better than the original surfaces.

- The new Surface's area is not too much greater ($< 10\%$ greater) than the sum of their original areas minus the area of overlap (if the original surfaces overlap).

The new Surface is added to the system and the original Surfaces are deleted (if the Surfaces are data objects they are not deleted, but stored as evidence for the new Surface). We recursively join surfaces until there are no pairs of surfaces which meet the above requirements. As this operation generates larger and larger planar surfaces, the number of evidence objects may become unmanageable. However, the set of objects may be randomly sampled to compute the new plane since the original surface fit values were high and the surfaces were nearly the same. Also, it is adequate to use the convex hull of the original

Surfaces' boundaries as the boundary of the new Surface instead of doing a costly convex hull operation of all the evidence objects.

The Extend All Surfaces subphase is more expensive than Extend Excellent Surfaces since it also attempts to join neighboring surfaces whose fit values are low and surfaces which do not lie in nearly the same plane. If **Refine Surfaces** was unsuccessful in improving the surface fit value for a particular surface, then perhaps it is necessary to merge the data from this Surface with another poorly fit neighboring Surface, or donate the Surface's evidence to neighboring Surfaces which already have high fit values.

Extend All Surfaces recursively attempts to join pairs of neighboring reconstructed surfaces that have high density (> 0.5). A new Surface is created from the original Surface's evidence using the algorithm in Section 3.3.1. If the new Surface's fit value is significantly higher than the original Surfaces the new Surface replaces the original Surfaces.

Fitting to Quadric Surfaces

The third subphase of **Extend Surfaces**, Try Quadrics, joins small data and reconstructed planar surfaces into larger non-planar surfaces. Currently this is only implemented for spherical surfaces, but the system can be extended to try cylindrical or other more complex quadric surfaces. Heckbert [14] presents the classes of quadric surfaces and some of their useful computational properties. Figure 3.17 shows planar Surfaces produced by the **Create Surfaces** phase which are joined to make two spherical Surfaces in the Try Quadrics subphase of **Extend Surfaces**.

We examine each planar Surface P_0 that has high surface density (> 0.5) and high surface fit (> 0.5) and does not already contribute to a Sphere. If P_0 fits well to an existing Sphere, we add P_0 and recompute the Sphere (as described below). Otherwise, we compare P_0 to each of its neighboring planar Surfaces that also have high surface density and fit. Figure 3.18a shows P_0 and a neighboring surface P_i with their normals (n_0 and n_i) and centers of mass (m_0 and m_i). A new Sphere is created if at least four of P_0's neighboring surfaces meet the following criteria:

- The lines parallel to the normals n_0 and n_i that pass through the centers of mass m_0 and m_i (nearly) intersect (i.e., these lines are not skew). Call the point of intersection C_i, the pairwise center estimate.

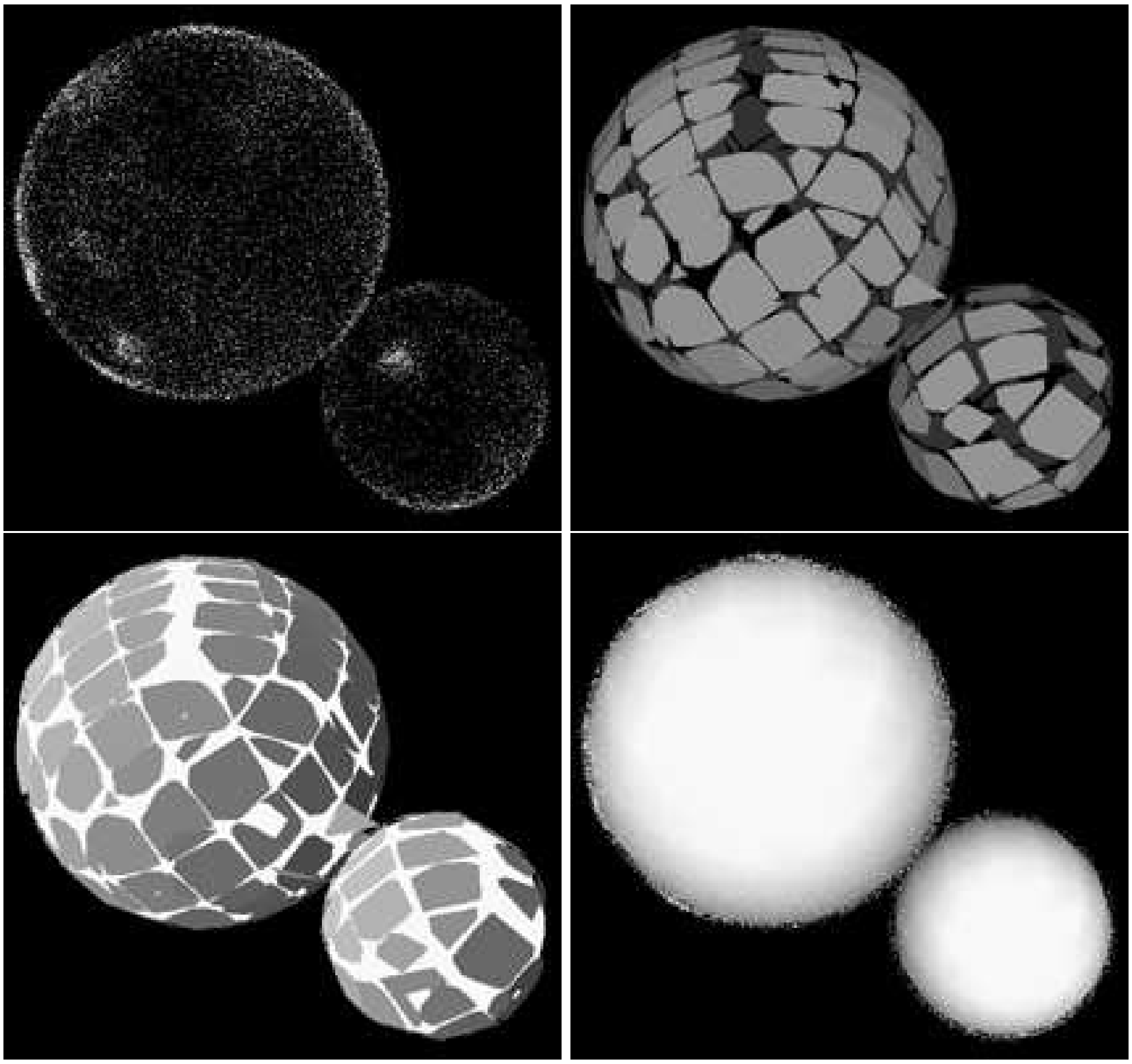


Figure 3.17: Spherical Surfaces fit to synthetic point cloud dataset of approximately 17,000 points.

- The angle between n_0 and n_i, labeled α in the figure, is between 10 and 45 degrees. If it is too narrow, the Surfaces should probably be joined into a planar surface; if it is too wide, the Surfaces probably meet at a sharp corner. Currently, we only reconstruct outwardly facing spherical surfaces, but the same algorithm can work for spherical interiors.

- The areas of P_0 and P_i are within an order of magnitude of each other. We should not try to create a Sphere between a large Plane and a tiny (possibly erroneous) Plane.

- The distance between m_0 and m_i, labeled x in the figure, is within an order of mag-

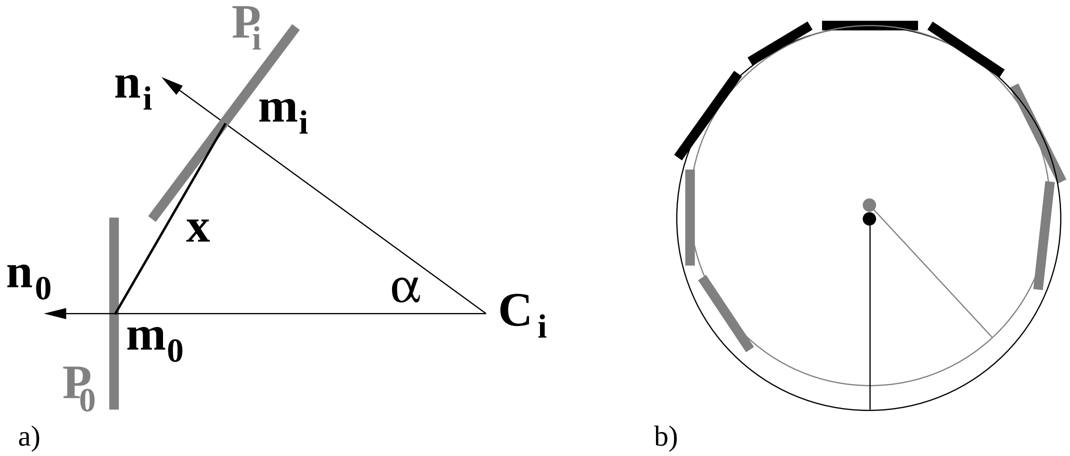


Figure 3.18: *a*) A Sphere can be estimated from just two neighboring planar surface patches. *b*) An existing Sphere (shown in black) can be extended with neighboring surfaces (shown in gray) and recomputed as necessary (the smaller gray circle).

nitude of the square root of the area of P_i. If the centers of mass are too close, then the Surfaces "overlap" too much to make an accurate estimate. If the centers of mass are too far apart, the surfaces might belong to different blocks and we should not join them.

- The distances of m_0 and m_i to C_i, differ by less than 10%.

A new Sphere is created from P_0 and the neighboring surfaces P_i that meet the above criteria (at least five total Surfaces). We use an iterative re-weighting method similar to the Plane fitting algorithm used in the **Create Surfaces** phase (Section 3.3.1). For each iteration, we compute the center C and radius r of the new Sphere. C is the weighted average of the center estimates C_i and r is the weighted average of the distances from each m_i to C.

For the first iteration, C_i is the pairwise center estimate between P_0 and P_i (as shown in Figure 3.18*a*). The weight used for P_i, w_i, is the area of P_i.

For subsequent iterations, C_i is the center of the unique Sphere of radius r_{prev} that is tangent to P_i at the point m_i. r_{prev} is the radius of the Sphere from the previous iteration. w_i = area of $P_i \times$ evidence $\text{fit}_{i\ to\ Sphere}$.

The above computation computes an optimal Sphere which is tangent to each of the planar Surfaces at the center of mass of the Surface. The resulting Sphere is probably too small and should be refined with the original data. As shown in Figure 3.17, the original

Points lie outside of the Sphere. Refinement of this estimate will increase the Sphere radius to fit the original points.

Extending Surfaces with Unclaimed Data

As described in the Section 3.3.2, the Use Best Fit Data subphase of **Refine Surfaces** allows surfaces with high plane fit values to release data fragments which are not a good fit. These data fragments may be collected by other surfaces to which they fit more closely. The final subphase of **Extend Surfaces**, Unclaimed Data, adds to each Surface any nearby data fragments which fit the surface well (have evidence fit values which are > 0.5), and recomputes the Surface. Several iterations of this subphase may be necessary since each Surface will only collect objects which are close to its current boundary.

For optimal results, execution of the **Refine Surfaces** and **Extend Surfaces** phases can be alternated iteratively. As we construct larger Surfaces, the surface fit value should increase (become a better fit) since data objects are required to fit more closely to smaller surfaces (see Section 3.4.3). If iterated surface refinement and extension do not result in quality surfaces which use most of the available data fragments and cover the region of interest, then it is probable that the data for this portion of the environment is too noisy or sparse for reconstruction. A feedback loop to the data collection and analysis systems can signal to gather more data and/or put more effort into generating building fragments for this area of the environment.

3.3.4 Match Edgels

The **Match Edgels** phase of aggregation joins Edgels to make longer Edgels and attaches Edgels to Surfaces to create hinges which suggest solid three-dimensional structure in the environment.

In the Intersect Surfaces subphase, we create an Edgel for each pair of neighboring planar Surfaces with high surface density (> 0.5) and high surface fit (> 0.5) that meet the following conditions:

- The angle between the Surface's normals is greater than 20 degrees.
- Most of each Surface lies on one "side" of the other Surface. The intersection of the planes (a line) and the distance from the line to each of the Surface's center of mass

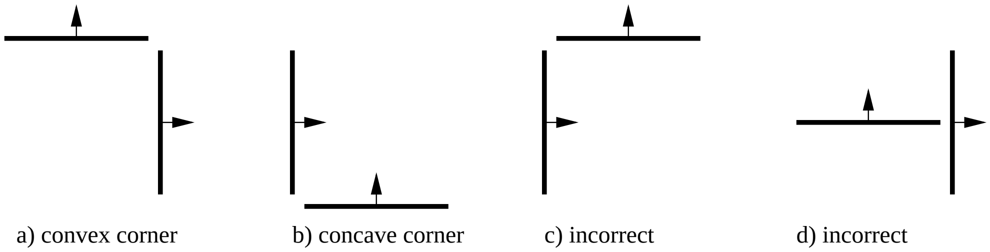


Figure 3.19: Edgels are created for neighboring Surfaces which form a consistent corner as in a and b. Edgels are not created for Surfaces c) which do not agree on orientation or d) whose intersection passes near the center of the one of the Surfaces.

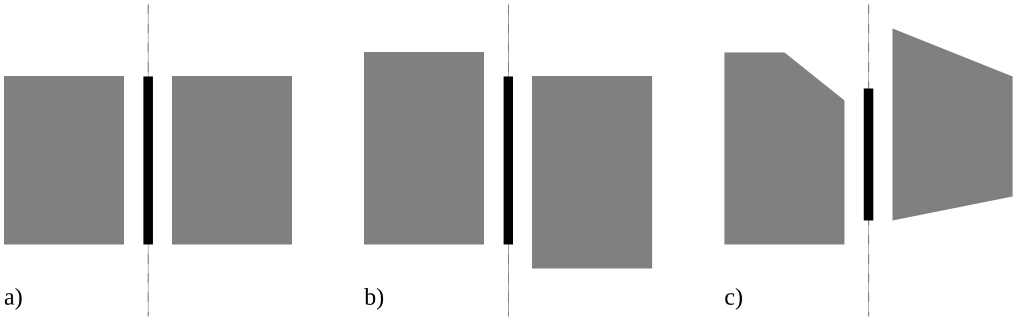


Figure 3.20: The intersection line is trimmed to the longest length that is still within a minimum distance of each surface.

is computed. The distance from each center of mass to the intersection line should be within an order of magnitude of the square root of the Surface's area.

- The normals are both oriented in the same way, either they point outward or inward (as shown in Figure 3.19).
- The surfaces intersect, or are very close to intersecting, relative to their size. The distance from each Surface to the Edgel should be less than the square root of the Surface area.

If these conditions hold, an Edgel is constructed by trimming the intersection line (as shown in Figure 3.20) so that the distance from any point on the Edgel to some point on each Surface is small relative to the area of the Surface.

In the Join Edgels subphase of **Match Edgels**, neighboring data and reconstructed Edgels which are nearly parallel are joined together. The Edgel detection algorithms may find only short portions of longer building edges at a time due to occluders. Data Edgels can be matched with reconstructed Edgels. Data Edgels can be used to neatly trim the

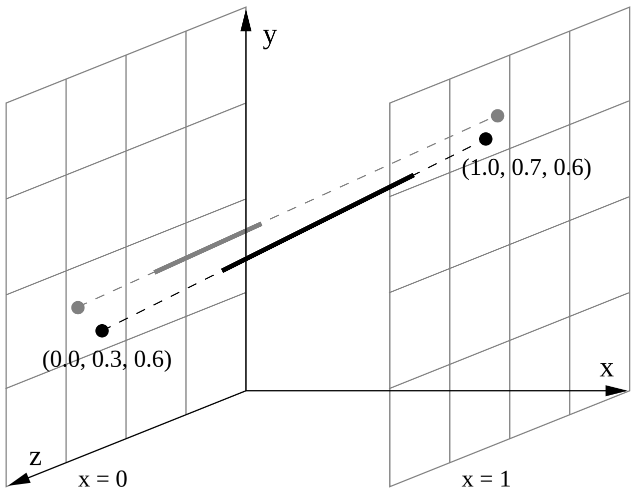


Figure 3.21: Two Edgels and their intercepts on the planes of the cube from (0, 0, 0) to (1, 1, 1). The black Edgel is most perpendicular to the x-axis and intersects the $x = 0$ plane at (0.0, 0.3, 0.6) and the $x = 1$ plane at (1.0, 0.7, 0.6). This Edgel is then inserted into the x-plane's four-dimensional tree with the values [0.3, 0.6, 0.7, 0.6]. The gray Edgel has intercepts in the same quadrants of the tree and is therefore nearly parallel to the black Edgel.

ragged edges of Surfaces shown in Figure 3.8. Edgels can suggest shallow angle bends in a large surface that does not fit its data well.

If there are many Edgels in the system, it may be more efficient to find neighboring parallel Edgels by first organizing all Edgels into a four-dimensional tree which is a duality of their slopes in three-dimensional space [2], as follows: Each Edgel in the system is projected to the two parallel planes of the scene bounding box to which it is most perpendicular (see Figure 3.21). Each Edgel is inserted into an adaptively-subdivided four-dimensional tree based on these intercepts. Finding other parallel Edgels involves searching the neighboring spaces in this tree.

3.3.5 Construct Blocks

An Edgel which joins two surfaces is evidence of solid urban structure in the environment. In the Create/Add To Blocks subphase of **Construct Blocks**, Edgels and Surfaces are used to Create new Blocks or added to existing Blocks as appropriate.

First, the Edgels that join two Surfaces at a convex corner (see Figure 3.19) and do not fit to any existing Block are processed. A new Block is created for the Edgel by making a convex parallelpiped between the Edgel, and two vectors perpendicular to the Edgel along

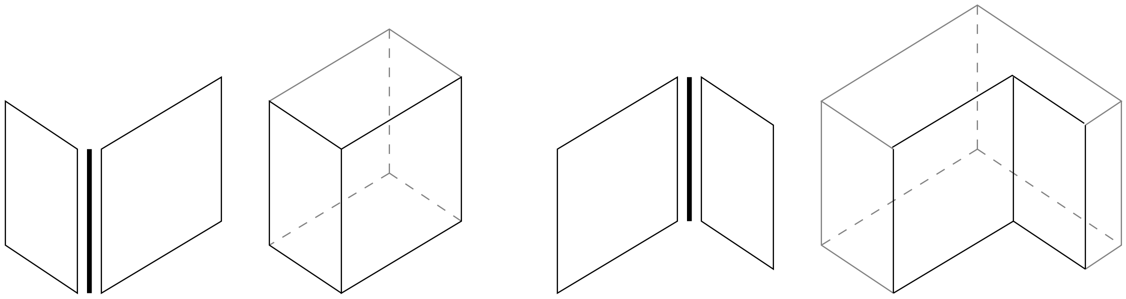


Figure 3.22: Concave and convex blocks each made from an Edgel joining two Surfaces.

the length of each Surface (see Figure 3.22a). Then for each Edgel that joins two Surfaces at a concave corner (and cannot be added to an existing Block or Blocks) we create an L-shaped Block as shown in Figure 3.22b. The depth of each arm of the Block is based on the dimensions of the corresponding Surface. The remaining Surfaces and Edgels are created and linked into the Block, but tagged as hypotheses so they can be easily modified as information is discovered for the other sides of the building.

Large (nearly) rectangular Surfaces on the boundary of the scene may not have any Edgel evidence attached to them. Blocks are also created for these Surfaces by creating a rectilinear solid whose depth proportional to the area of the Surface.

Normally, the system will add evidence to an existing Block instead of creating a new Block and merging it with neighboring Blocks; however, when merging the results of previously saved aggregation sessions we will likely need to join Blocks which were created from different evidence. The Join Blocks phase of **Construct Blocks** merges Block structures that intersect or overlap.

3.4 Remove Outliers

Not all building fragment data is equally useful. In fact, some of these objects should be discarded because they are outliers or are exceptionally noisy. Outliers are data fragments arising from false correlations where no building structure exists. This can happen because of coincidental agreement in images, repetitious facades, trees, or other minor urban structure. Noise is imprecision in the data introduced by imperfections in the camera lens or interior calibration, errors in the camera pose estimates, and mathematical approximations in the building fragment algorithms, etc.

Non-sensical Surfaces created from outliers and noisy data should be removed from the system. When objects are removed from the system, reconstructed objects which use them as evidence need to be recomputed or deleted if adequate evidence does not remain. In the following sections I describe my methods for handling outliers, noise, and the reconstructed objects they influence. These techniques are called the **Remove Outliers** options and the metrics and parameters described in the following sections are used in the phases of aggregation to weight the use of data objects and to regulate the creation of new objects. Appendix A discusses appropriate values for each parameter.

3.4.1 Minimum Size Requirements

The parameters `average_building_side` and `nodes_per_side` (described in Appendix A.1) specify the relative size of buildings in the environment and the level of detail expected to be reconstructed from the available data. Sparse or noisy datasets can provide information only for coarsely detailed models. Spurious building fragment data and reconstructed objects can be eliminated by setting minimum requirements on the length, area, and volume of Edgels, Surfaces, and Blocks. These minimum values are specified by the user as a percentage of the `average_building_size`, `average_building_size` squared, and `average_building_size` cubed respectively. The system could be modified to estimate these parameters iteratively.

3.4.2 Surface Density

The *density* of the n Point, Surfel, Edgel, and Surface data objects which are evidence for a reconstructed Surface can be used as a measurement of the quality of that surface. The global parameters `average_building_side` and `average_points_per_building_side` are used to compare the densities of Surfaces created from Points.

$$\text{surface density}_{Surface} = \frac{\sum_{i=1}^{n} \text{relative weight}_i}{\text{Surface area}}$$

The relative weight of the different data objects in the system are computed as follows:

$$\begin{aligned}
\text{relative weight}_{Point} &= \frac{(\texttt{average_building_side})^2}{\texttt{average_points_per_building_side}} \\
\text{relative weight}_{Surfel} &= \text{Surfel area} \\
\text{relative weight}_{Edgel} &= \text{Edgel certainty} \times \text{Edgel length} \times \text{Edgel width}
\end{aligned}$$

$$\text{Edgel width} = \frac{\min(\text{Edgel length}, \texttt{average\_building\_side})}{10}$$

$$\text{relative weight}_{Surface} = \text{Surface area}$$

The weights for Points, Surfels and Surfaces are straightforward comparisons of their areas. Edgels are difficult to weight appropriately for a number of reasons.

- Different algorithms produce Edgels in different ways. Satyan Coorg's algorithm generates an Edgel for every line detected in each image, while George Chou's program generates an Edgel for each building feature which is an average of many agreeing observations and discards observations which do not have many matches. I assume that the file parsing modules for each different type of input data has given each Edgel an appropriate *Edgel certainty* value which varies from 0.0 to 1.0, and can be used to compare Edgels produced from different algorithms. For example, an Edgel from Satyan's program might receive a weight of 0.1, since few cameras observed it, while Edgels from George Chou's program would usually be weighted between 0.5 and 1.0, depending on how many cameras observed the data (this information is available in George Chou's program output file format).

- There is no obvious way to compare a one dimensional object (a line) to a two dimensional object (a plane). I created a measurement called *Edgel width* which is one-tenth the size of the Edgel's length (or the `average_building_size`, whichever is smaller). Thus the ratio of weights of an Edgel of length l and certainty 1.0 to a Surface with area $l \times l$ is 1:10.

The formula for the relative weight of an Edgel may need to be adjusted as more Edgel datasets and more information about their creation becomes available.

The metric for surface density could be altered to minimize the effects of uneven data distribution (Figure 3.8 illustrates several Surfaces with uneven Point distribution). For example, we could define the density of a Surface as follows: divide the Surface into n equal subsurfaces; compute the density of each subsurface using the above method; and define the density of the Surface to be the minimum of these subsurface densities. This method prevents highly sampled regions of Surface from expanding unchecked into sparsely sampled regions.

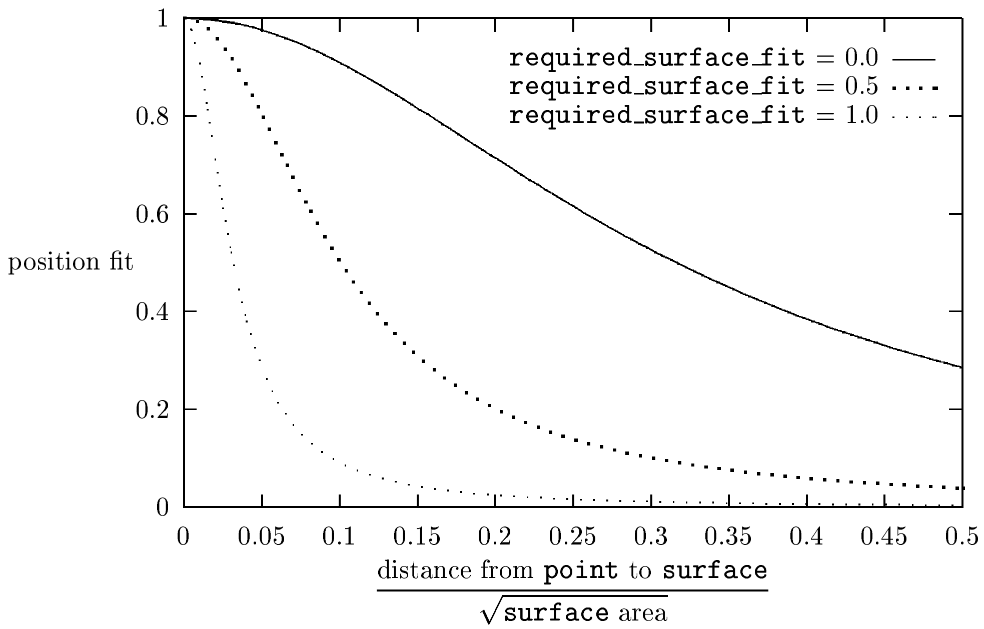
3.4.3 Surface Fit

How well a reconstructed surface fits its corresponding evidence is a measurement of the quality of that surface and its normal. The *surface fit* of a Surface is a weighted sum of the *evidence fit* values for each of its n evidence objects.

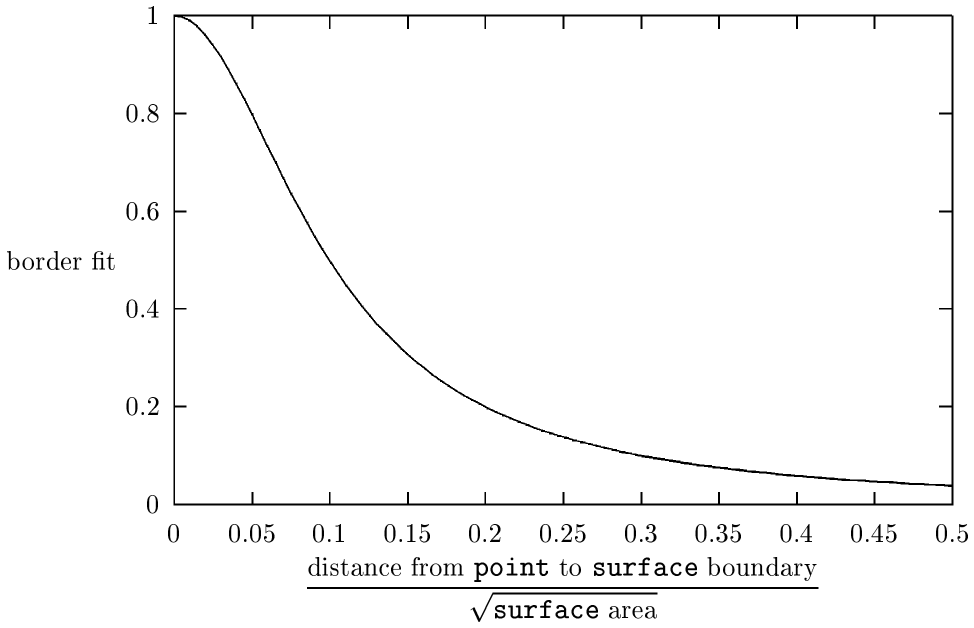
$$\text{surface fit}_{Surface} = \frac{\sum_{i=1}^{n} \text{relative weight}_i \times \text{evidence fit}_{i \ to \ Surface}}{\sum_{i=1}^{n} \text{relative weight}_i}$$

The evidence fit of an object to a Surface is the product of three metrics, each of which range from 0.0 to 1.0: position fit measures how well a point fits the equation of a Surface (1.0 indicates perfect fit); border fit indicates whether a point is within the current boundary of the Surface (1.0) or how close it is to the boundary; and normal fit evaluates how well the normal fits the Surface at a particular point (1.0 for perfect fit). These metrics are methods of the Surface class (see Section 4.1). Below are the equations for each metric and their respective graphs. These equations were chosen to produce desired graph shapes, their exact formulas are not important.

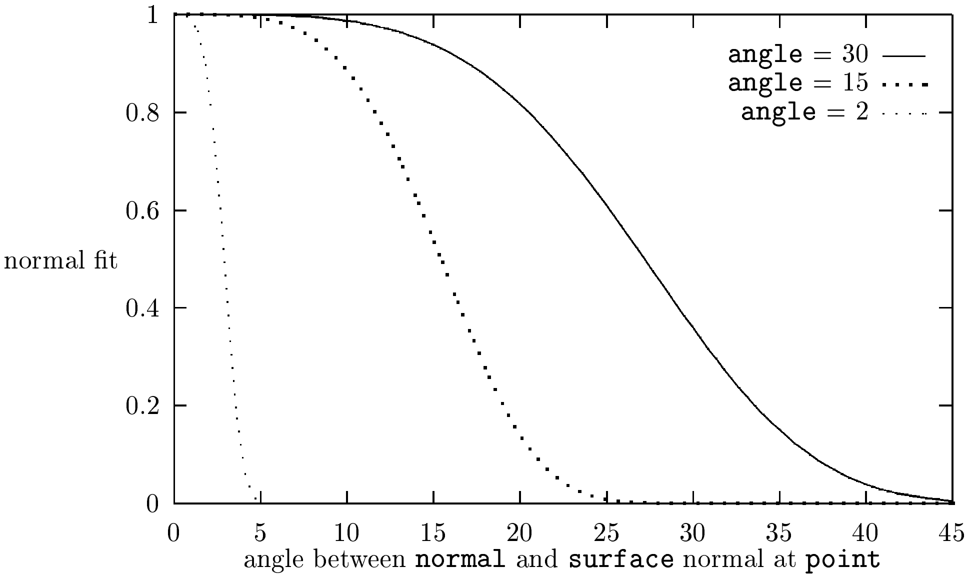
$$\texttt{surface->position\_fit(point)} = \frac{1}{1 + 10^{(2 \cdot \texttt{required\_surface\_fit} + 1)} \cdot \left(\frac{\text{distance from } \texttt{point} \text{ to } \texttt{surface}}{\sqrt{\texttt{surface} \text{ area}}}\right)^2}$$



$$\texttt{surface->border\_fit(point)} = \frac{1}{1 \ + \ 100 \cdot \left(\frac{\text{distance from } \texttt{point} \text{ to } \texttt{surface} \text{ boundary}}{\sqrt{\texttt{surface} \text{ area}}} \right)^2}$$



$$\texttt{surface->normal\_fit(point, normal, angle)} = e^{\left(\frac{(\text{angle between } \texttt{normal} \text{ and } \texttt{surface} \text{ normal at } \texttt{point})^4}{1 \ + \ 10^{\log_2 \texttt{angle}}} \right)}$$



The evidence fit equations for each type of object fits are presented below. The formula describing how well a planar surface fits a spherical surface includes a ratio of their areas; a large Plane cannot be used as evidence for a much smaller Sphere.

evidence $\text{fit}_{Point\ to\ Surface} =$

Surface->`position_fit`(Point position) $\times$

Surface->`border_fit`(Point position) $\times$

Surface->`normal_fit`(Point position, Point normal, 30 degrees)

evidence $\text{fit}_{Surfel\ to\ Surface} =$

Surface->`position_fit`(Surfel position) $\times$

Surface->`border_fit`(Surfel position) $\times$

Surface->`normal_fit`(Surfel position, Surfel normal, 2 degrees)

evidence $\text{fit}_{Edgel\ to\ Surface} =$

Surface->`position_fit`(Edgel endpoint_1) $\times$

Surface->`position_fit`(Edgel endpoint_2) $\times$

Surface->`border_fit`(Edgel endpoint_1) $\times$

Surface->`border_fit`(Edgel endpoint_2)

evidence $\text{fit}_{Plane\ to\ Plane'} =$

Plane′->`position_fit`(Plane center of mass) $\times$

Plane′->`border_fit`(Plane center of mass) $\times$

Plane′->`normal_fit`(Plane center of mass, Plane normal, 2 degrees)

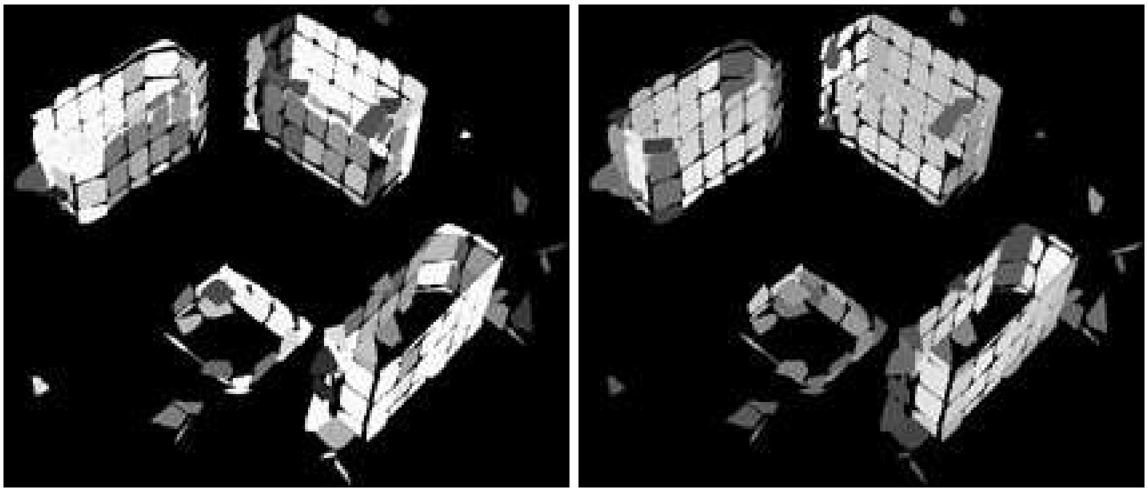
evidence $\text{fit}_{Plane\ to\ Sphere} =$

Sphere->`position_fit`(Plane center of mass) $\times$

Sphere->`border_fit`(Plane center of mass) $\times$

Sphere->`normal_fit`(Plane center of mass, Plane normal, 2 degrees) $\times$

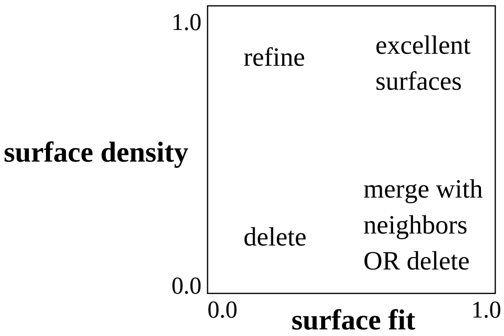
$$\max\left(1,\ \frac{\text{Sphere area}}{4\pi \cdot \text{Plane area}}\right)$$



shaded by surface density　　　　shaded by surface fit

Figure 3.23: Comparing Surfaces using the surface density and surface fit metrics (lighter surfaces have higher values). The interior Surfaces have lower density since fewer Points were available for those Surfaces, and the Surfaces that contain evidence from more than one facade have lower surface fit.

Figure 3.23 shows a collection of reconstructed Surfaces shaded by the surface density and surface fit metrics. The following diagram illustrates how these metrics are used to process reconstructed Surfaces.



Surfaces with poor density and poor fit should be removed. Surfaces with high density but poor fit can be refined (as decried in Section 3.3.2). Surfaces with poor density and high fit that are not successfully merged with neighboring Surfaces in the **Extend Surfaces** phase (Section 3.3.3) should be removed because they were probably created from outliers and noisy data.

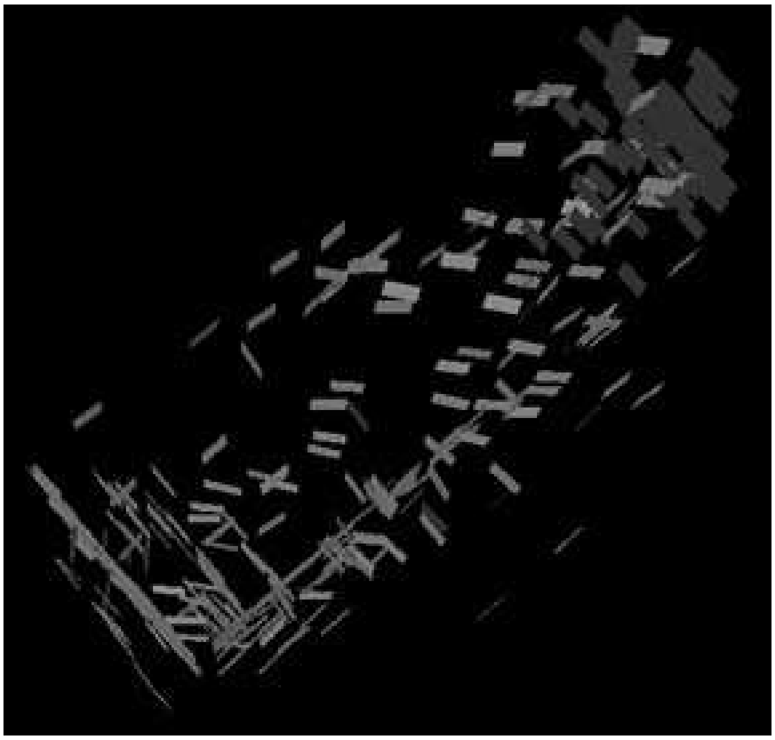


Figure 3.24: View from above of a Surfel dataset from J.P. Mellor's algorithm which contains many outliers. Note the abundance of Surfels at a 45 degree angle to the actual facades of the building.

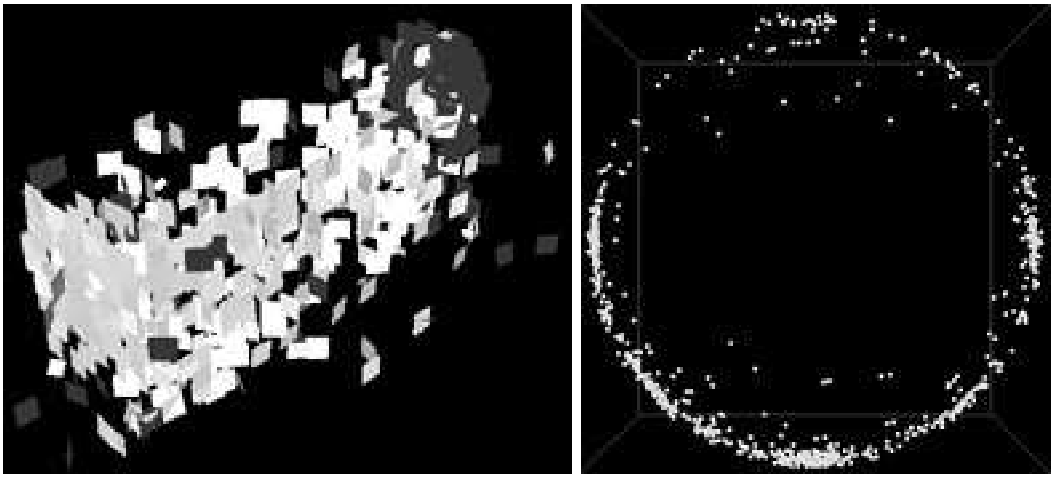
3.4.4 Surface Normal

The Surfel datasets produced by J.P. Mellor's algorithm have many outliers (see Figure 3.24) and I have developed several metrics to eliminate the erroneous Surfels. First, I plotted the normals of all the Surfels as points on the unit sphere shown in Figure 3.25a and eliminated Surfels which did not have many neighbors in the sphere plot.

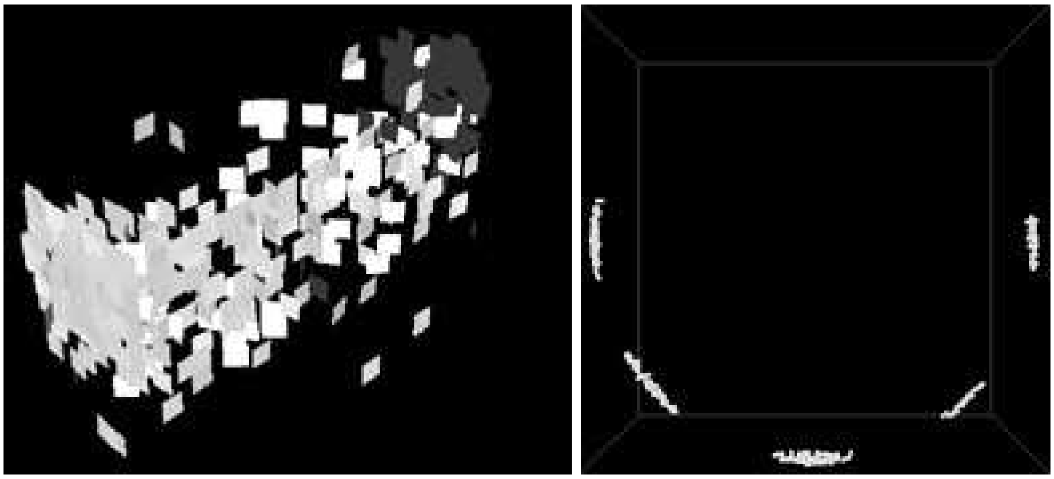
As expected, the remaining Surfel normals clustered in the directions of the four vertical facades, but they also clustered at positions suggesting Surfaces at 45 degree angles to the actual facades. Even though many Surfels gave evidence for planes with these incorrect normals, they did not agree about the depth (i.e. the offset of the plane from the origin); they proposed parallel planes (see Figure 3.25b). I was able to eliminate these outliers by requiring Surfels with similar normals to also have a minimum number of neighbors which agreed about the plane depth.

The final result is shown in Figure 3.25c. Fencepost problems which remain can be resolved by comparing the density of evidence for competing parallel planes.

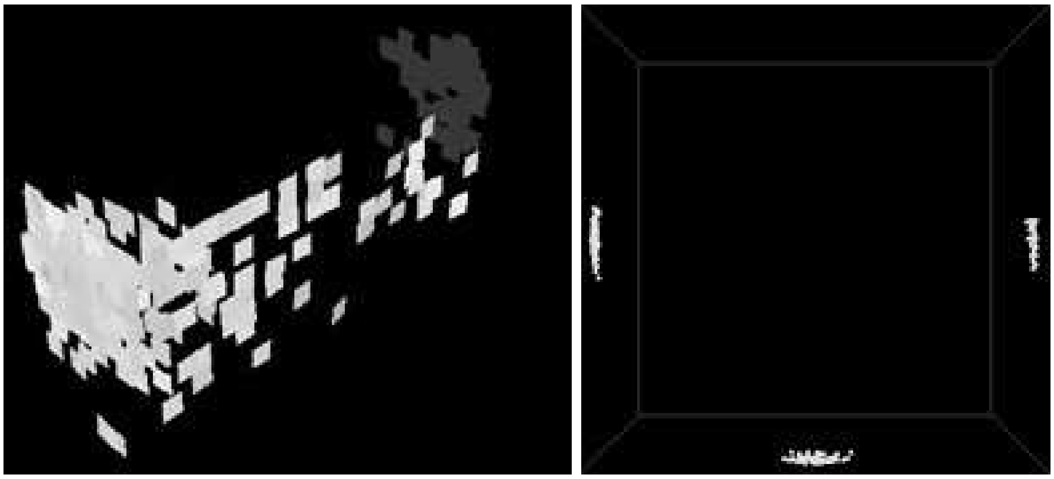
The above ideas are grouped together in the Remove Outliers option Surface Normal



a) Original dataset which contains many outliers.



b) After culling by normal.



c) After culling Surfels which do not agree in plane depth.

Figure 3.25: Surfel dataset (left images) and plot of Surfel normals on the unit sphere viewed from above (right images).

which removes Surfels from the system which do not have at least `normal_count` neighbors which lie on "nearly the same surface". Two Surfels or planar Surfaces are said to be nearly the same Surface if the angle between their normals is small relative to the `surface_normal` parameter, and the distance from the center of mass of each Surface to the equation of the other Surface is less than one tenth the square root of the area of the of the other Surface. If some Surfels are so large that they might individually represent facades (this should be apparent from the `average_building_size` parameter) then they should not be culled for the above reasons alone. This technique needs modification before it can be used on a Surfel dataset of a spherical Surface since these Surfels will have very few neighbors in the spherical plot by normal.

Chapter 4

Implementation

This Chapter presents the important aspects of my implementation of the aggregation algorithm described in Chapter 3. Section 4.1 describes the C++ class hierarchy that implements the types of building fragments used by my program. Section 4.2 explains my memory management scheme since C++ does not have garbage collection. Section 4.3 describes how the tree structure is walked. Section 4.4 discusses the system modules and interfaces. Section 4.5 describes how objects are loaded into the system from files, and the output file format used to save the complete system state. Finally, Section 4.6 describes how synthetic building fragments for are generated for testing.

4.1 Object Hierarchy

The building fragment objects described in Section 3.1 are organized in the C++ class hierarchy shown in Figure 4.1*a*. An Object is a building fragment that was loaded from a file (a *data* object), created during a phase of aggregation from other objects (a *reconstructed* object), or synthetically created by my program to be used for testing (a *generated* object).

The implementation of the Object and its subclasses uses the following simple classes: Three, a vector of three floating point numbers; Bag, a templated set; Segment, two endpoints representing a finite line segment; and Bounding Box, the smallest rectangular axis-aligned region of space which completely contains an Object or TreeNode.
Object has the following fields:

`int id`
: A unique identifier for the object.

`char creator[]`

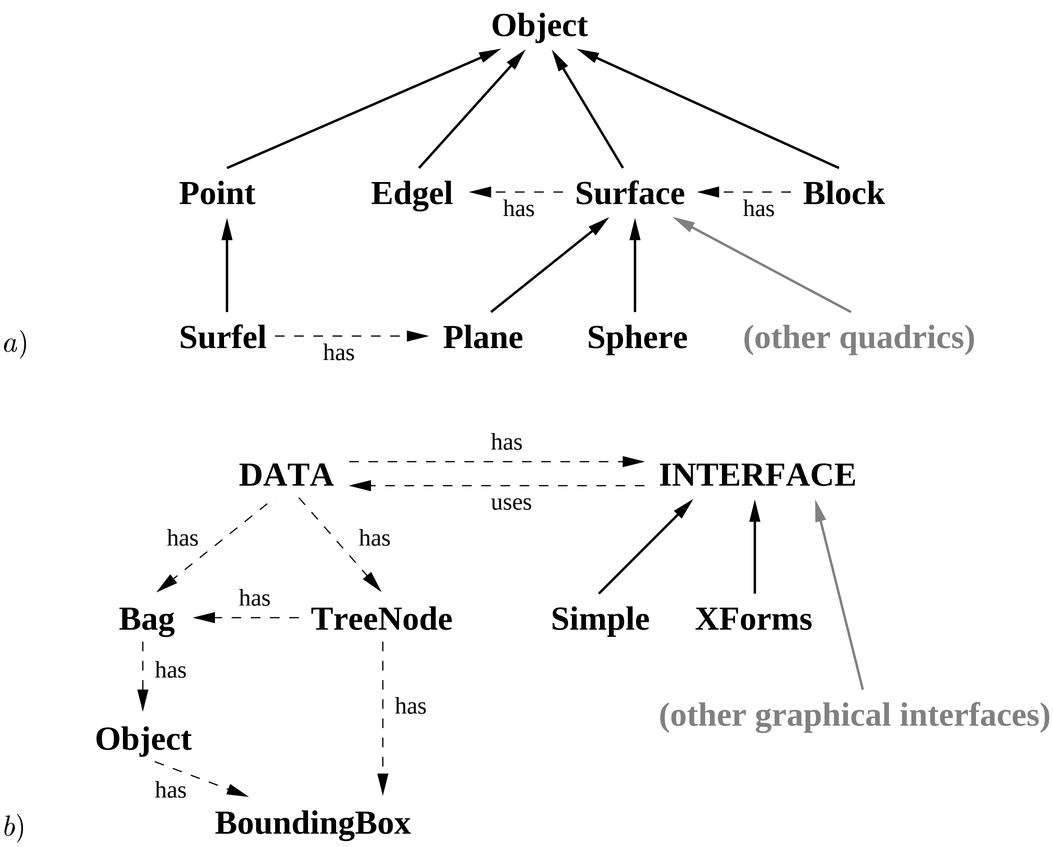


Figure 4.1: Diagram of the a) object class hierarchy and b) main program modules and data structures. Inheritance relationships are drawn with thick solid arrows from subclass to superclass and special relationships are labeled and drawn with dashed lines.

For data objects this is the name of the file from which it was loaded.

BoundingBox *bb
: The maximum and minimum x, y, and z values of the object.

Bag<Object> *evidence
: For a reconstructed object this is the collection of objects that were used to create it.

Bag<Object> *contributes
: The collection of reconstructed objects that have been created using this object as evidence (if any). $a \in b$->contributes iff $b \in a$->evidence.

int treetag
: Used when walking the tree (described in Section 4.3).

Each class derived from Object must implement the following methods:

int intersectsBB(Boundingbox *bb)
: Returns 1 if this object at least partially intersects bb, returns 0 otherwise.

float weight()

Returns the relative weight$_{object}$ (described in Section 3.4.2).

`float evidencefit(Surface *s)`
Returns the evidence fit$_{object\ to\ s}$ (described in Section 3.4.3).

`float intersectionWithRay(Segment *s, Three *&point)`
Computes the (near) intersection of object with `s` (or NULL if they do not intersect) and stores it in `point`. Returns the distance between `point` and object (i.e. the error of intersection).

`void save(FILE *file)`
Writes this object to `file`.

`void print()`
Displays information about object to `stdout`.

`void draw()`
Executes OpenGL commands used to render the object.

`Three* color(enum COLOR_MODE mode)`
Returns the RGB color which should be used to render the object based on the current `mode` (where mode is one of `DEFAULT`, `NORMAL`, `DENSITY`, or `SURFACEFIT`).

The class Point (see Section 3.1.1) has the following additional fields:

`Three *position`
The location of the Point.

`Three *normal`
The estimate of the surface normal at this Point (if available).

The class Surfel (see Section 3.1.2) is derived from Point. The `normal` estimate of a Surfel is required and should be very accurate (within a few degrees). Surfel has the following additional field:

`Plane *plane`
The size and shape of the Surfel.

The class Edgel (see Section 3.1.3) has the following additional fields:

`Three *endpoint1,   Three *endpoint2`
The endpoints of this Edgel.

`float certainty`
A number between 0.0 and 1.0 which indicates how this Edgel should be weighted (see Section 3.4.2).

`Edgel *twin,   Edgel *prev,   Edgel *next,   Surface *surface`
The connections of the half-edge data structure shown in Figure 3.9.

`enum FOLD fold`
: If this Edgel is connected to two surfaces, indicates whether this is a `CONVEX` or `CONCAVE` building corner.

Each Surface (see Section 3.1.4) is actually an instance of one of the subclasses, Plane or Sphere. The system can easily be extended with cylinders or other more complex surfaces. Surfaces have the following fields and methods:

`float area`
: The surface area.

`Block *block`
: The block to which this surface belongs (if any).

`BoundingBox *treebb`
: If this surface was created from a set of objects in a tree node (described in the **Create Surfaces** phase, Section 3.3.1), `treebb` is the BoundingBox of that tree node. Note: Since the tree might be rebuilt after the creation of this object, we cannot point to the tree node.

`static Surface* Create(Bag<Object> *objects)`
: Creates a surface to best fit the `objects`.

`int recompute()`
: To be used after adding or deleting objects from the evidence list.

`float PositionFit(Three *point)`
: Returns a value between 0.0 and 1.0 indicating how well `point` fits the equation of this Surface relative to the parameter `required_surface_fit` (see Section 3.4.3).

`float BorderFit(Three *point)`
: Returns 1.0 is `point` is within the boundary of this Surface, otherwise returns a value between 0.0 and 1.0 indicating how close `point` is to the boundary of this Surface (see Section 3.4.3).

`float NormalFit(Three *point, Three *normal, float angle)`
: Returns a value between 0.0 and 1.0 indicating how closely `normal` fits the Surface at `point` relative to `angle` (see Section 3.4.3).

`Edgel* intersects(Surface *neighbor)`
: Returns the Edgel that represents the intersection of this surface with `neighbor`.

`enum FOLD GoodCornerMatch(Surface *neighbor)`
: Returns `CONVEX` or `CONCAVE` as appropriate if this surface and `neighbor` intersect (as described in Section 3.3.4), otherwise returns UNKNOWN.

The representation of a Plane has the following fields:

`Three *normal`
: The normal (a, b, c) of this Plane. $a^2 + b^2 + c^2 = 1$.

`float d`
: The offset of the plane from the origin, i.e., the last coefficient of the plane equation $ax + by + cz + d = 0$.

`Edgel *edgel`
: A member of the chain of Edgels that form the boundary of the surface.

`Three *centerOfMass`
: The point on which the Plane would balance if it were made of a uniform material.

The representation of a Sphere has the following fields:

`Three *center`
: The center of the sphere.

`float radius`
: The radius of the sphere.

A Block (see Section 3.1.5) has the following fields and methods:

`Bag<Surface> *surfaces`
: The set of all the Surfaces that form the Block. The surfaces are connected in a half-edge data structure so all Surfaces and Edgels can be accessed from any Surface or Edgel.

`float volume`
: The volume of the region enclosed by `surfaces`.

`static Block* CreateBlock(Edgel *edgel)`
: Creates a new Block from this Edgel which must be connected to two Surfaces so we can hypothesize a rectilinear solid from it.

`static Block* CreateBlock(Surface *surface)`
: Creates a Block from a single Surface.

`int AddToBlock(Edgel *edgel)`
: Returns `1` if `edgel` is successfully added to the structure, `0` otherwise.

`int AddToBlock(Surface *surface)`
: Returns `1` if `surface` is successfully added to the structure, `0` otherwise.

`int JoinBlock(Block *block)`
: Returns `1` if `block` is successfully joined to this Block's structure, `0` otherwise.

4.2 Object Allocation and Registration

In order to keep track of all the objects and the memory allocated to store the objects, every geometrical object that is created and used in the aggregation algorithm is registered in the system. The central registration unit is then responsible for making sure that each object is added to the tree (as described in Section 3.2) so it is available for nearest neighbor computations. The official copies of all objects are stored by type in unordered linked lists, called Bags. Objects can also be deleted from the system using the corresponding unregister command which will carefully clean up the memory associated with that object.

Each reconstructed object maintains a list of its evidence and each object stores a list of all the reconstructed objects to which it contributes. When reconstructed objects are used to create a new object, the new object may either:

- Use the evidence of the reconstructed objects as its own evidence, and delete the reconstructed objects. For example, the Extend Excellent Surfaces subphase of **Extend Surfaces** (see Section 3.3.3) destroys the intermediate reconstructed Surfaces.

- Use the reconstructed objects as evidence. For example, Spheres created in the Try Quadrics subphase of **Extend Surfaces** use both data and reconstructed planar Surfaces as evidence.

The evidence and contributes lists for each object are difficult and expensive to maintain as we recompute and delete reconstructed Surfaces and remove outliers. For this reason, some Bags of objects are actually maintained as Bags of object identifiers (the class variable `id`) which are looked up in a hash table. If the id is not in the hash table, the object must have been deleted from the system, so the reference is updated.

4.3 Using the Object Tree

The object tree described in Section 3.2 is implemented with the TreeNode class. Each TreeNode has a pointer to its parent and children nodes and each leaf TreeNode maintains a list of all objects whose Bounding Box at least partially overlaps its own BoundingBox.

The object tree is used when executing each phase of aggregation. The **Create Surfaces** phase uses the tree structure directly by creating a Surface for each leaf node's set of objects. The other phases perform operations on objects that meet certain criteria. The objects are processed by walking through the tree structure in a depth first manner, tagging objects as they are processed to ensure that we do not repeat work for objects whose BoundingBox overlaps more than one leaf node. Because each operation walks through the tree structure, we can use our current position in the tree to estimate the percentage of computation that has been completed.

The tree structure is also used to implement object selection with the mouse. A single click with the left mouse button initiates a tree search. A ray beginning at the point on the near plane where the mouse is clicked and ending at the far plane is sent through the tree. The leaf nodes along this ray are walked from near to far. The elements of each node are individually intersected with the ray. If there are any intersection points that lie within the node the closest intersection point is returned. If there are no such intersections, the tree walk continues.

4.4 Program Interfaces

The static DATA class (shown in Figure 4.1b) organizes the global system state including all registered objects (described in Section 4.2), the values of the parameters described in Appendix A, and the current Object Tree.

DATA has a link to an INTERFACE which implements how to change parameter values, execute phases of the algorithm, and display the results. The INTERFACE is an instance of one of the subclasses, Simple or XForms. The Simple INTERFACE operates in non-graphical batch mode, reading parameter values and aggregation commands from a command script (described in Appendix C) and printing output messages to the screen. The XForms INTERFACE is graphical, allowing the user to adjust parameters by setting slider and counter values in its GUI and execute commands with button presses (see Appendix B). The XForms INTERFACE also renders three-dimensional objects to Open GL canvases. Graphical interfaces for other platforms can be added as subclasses of INTERFACE.

The XForms INTERFACE's graphical user interface was built with the XForms GUI builder `fdesign` [30], and uses OpenGL [21] to render the building fragments and the computed model. I used interface code for XForms and OpenGL developed by Eric Amram while he was a student in the M.I.T. Computer Graphics Group, including mouse view control, object picking, and template classes for the rendering canvases.

The current implementation is compiled with CC, the SGI C++ compiler, on the Irix 6.2 platform and runs on SGI O2 and Indy machines. Purify, a memory management verifier by Rational Software, Inc. [22], was helpful for finding memory leaks and other deallocation errors in my program. However, the XForms code contains an illegal (but not fatal) memory exception during its initialization that Purify cannot ignore in order to proceed with the execution of my program. Due to this error, I could use Purify only with the non-graphical version of my program.

4.5 File Input and Output

The intermediate and final results of an aggregation can be saved to a file to be viewed by another program or loaded back into the aggregation program for further processing. There are four options for saving files: **Save_All**, which saves all of the objects; **Save_Reconstructed**, which saves reconstructed Edgels, Surfaces and Blocks; **Save_Final**, which saves the Blocks only; and **Save_Generated**, which saves the generated test data objects (see Section 4.6).

The object geometry is stored with the Inventor file format [25] using the point, line, face, and Sphere datatypes. Although not fully implemented, the **Save_All** option will also record the complete state of the system. The additional system information (the links between reconstructed objects and their evidence, the originating file name for data objects, etc.) can be stored as comments within the Inventor format. This augmented file format allows the program to completely recover all system state, and may be helpful for implementing an UNDO option, while still allowing the output file to be viewed as geometry by other programs.

I have implemented a limited Inventor parser which can handle the augmented format described above as well as the syntax used in the surveyed Technology Square CAD model and the output from Rebecca Xiong's City Generator described in Section 4.6. As mentioned in Section 1.2, a file parser must be provided for each new datatype. Currently, I have special-purpose routines to parse J.P. Mellor's Point and Surfel datasets, George Chou's Edgel datasets and Satyan Coorg's Edgel and Surface datasets. To avoid the need to implement and maintain additional parsers for the aggregation program, it would be easiest to require that all building fragments presented to the system are first converted into a common format, such as the augmented Inventor file format described above.

4.6 Generating Test Data

In Chapter 1, I described the building fragment data produced from the photographic images by the algorithms of J.P. Mellor, Satyan Coorg, and George Chou. The datasets currently

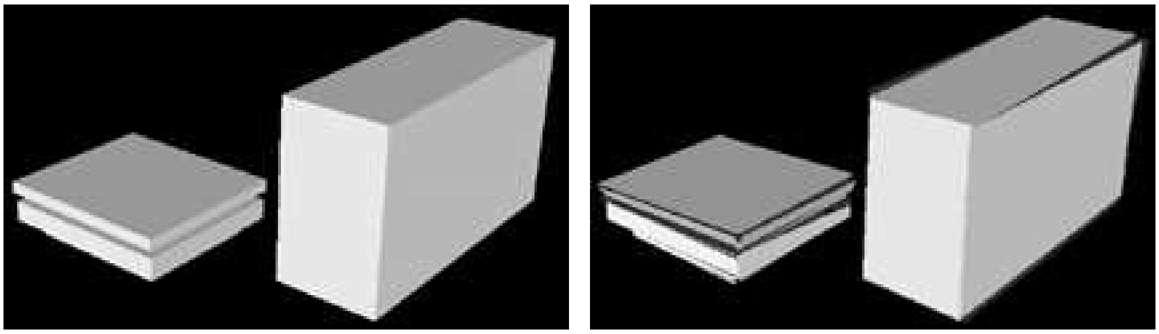


Figure 4.2: An Inventor model of Technology Square and the result of jittering each Surface. The jittered surfaces are no longer rectangular and do not meet each other neatly at 90 degree angles.

available contain building fragments from the environment near Technology Square which has a simple arrangement of large, flat-facaded rectilinear buildings that are nearly axis-aligned with the *Technology Square coordinate system.*

In more general urban environments the buildings may consist of non-convex shapes with walls that do not meet at 90 degree angles. The buildings may have domes or angular roofs and the facades may have columns or deep window ledges. And most importantly, building fragments will not be uniformly available for all surfaces in the environment. I have used a number of techniques to generate test datasets with these properties, including: sampling, jittering, transformation matrices, and ray casting.

The `load_sample` parameter can be used to create sparse Surface datasets. For example, if we load a Surface dataset with this parameter set to 0.5, approximately one-half of the Surfaces will be loaded.

The `data_jitter` parameter can be used to add noise to point cloud datasets and to perturb Edgels and Surfaces (see Figure 4.2). Each object type is represented by a set of three-dimensional points (described in Section 4.1). The data jitter option shifts each of these points by a small randomly oriented vector whose length is controlled by the value of this parameter. A randomly directed ray of length 1 is created by generating three uniformly distributed floating point numbers on the interval [-1, 1]. If the magnitude of this vector is too small (i.e. all three numbers are very nearly zero), repeat. Otherwise normalize this vector. The four vertices of each square Surface in Figure 4.2 have been individually jittered. The resulting vertices will probably no longer be co-planar, so the Surface constructor must first find the best fit plane to the new vertices.

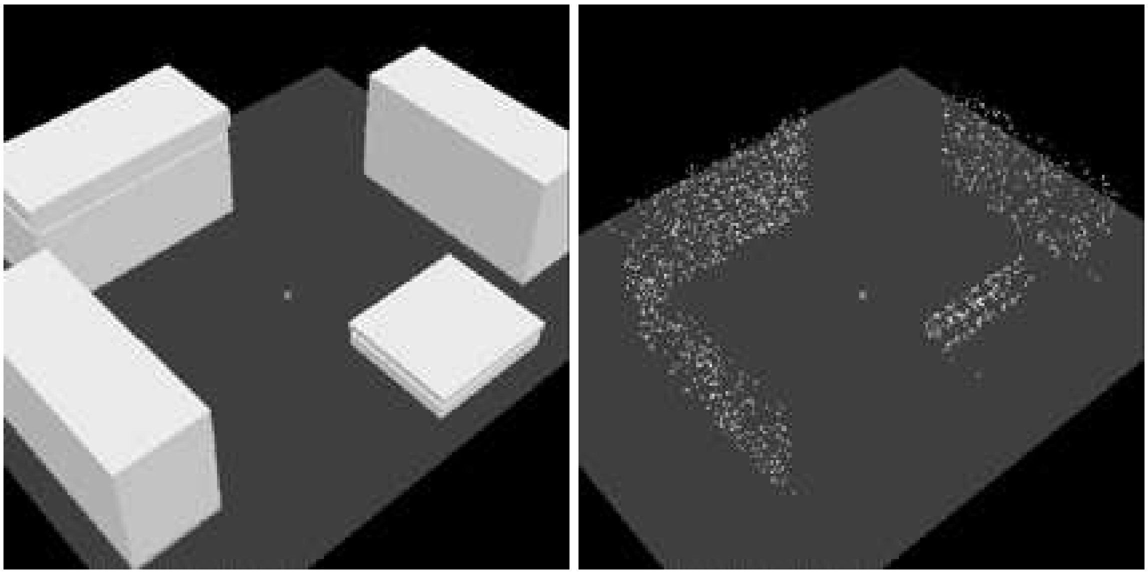


Figure 4.3: A single "camera" placed in the center of an Inventor model and the resulting synthetic dataset of approximately 1000 Points.

The transformation matrix can be used to make datasets which are no longer axis-aligned. I used this option to make sure that the aggregation algorithm is not dependent on a particular tree structure (see also Section 3.2).

My program can also be used to generate synthetic point cloud datasets from the building fragments currently loaded. The user specifies the positions of camera or laser scanning devices within the model. Randomly directed rays are cast from each camera position into the surrounding environment. A Point is created on the first object the ray intersects (if any). This method of sampling three dimensional objects creates uneven distribution and simulates how point cloud datasets are actually acquired. The user has control over the arrangement and density of cameras among the buildings. Very few Points will be generated for tight alleyways unless cameras are placed in them. Cameras placed very close to a surface result in a distinctive density pattern. If a regular sampling had been used across every surface these variations would not occur. This method could also be used to generate test Surfel datasets.

I apply the above techniques to new cities I created using Rebecca Xiong's Random City Model Generator [31]. Her program creates cities with varied building density, height, size and shape (rectangular solids, pyramids and cylinders). A few snapshots of a city from her program are shown in Figure 4.4.

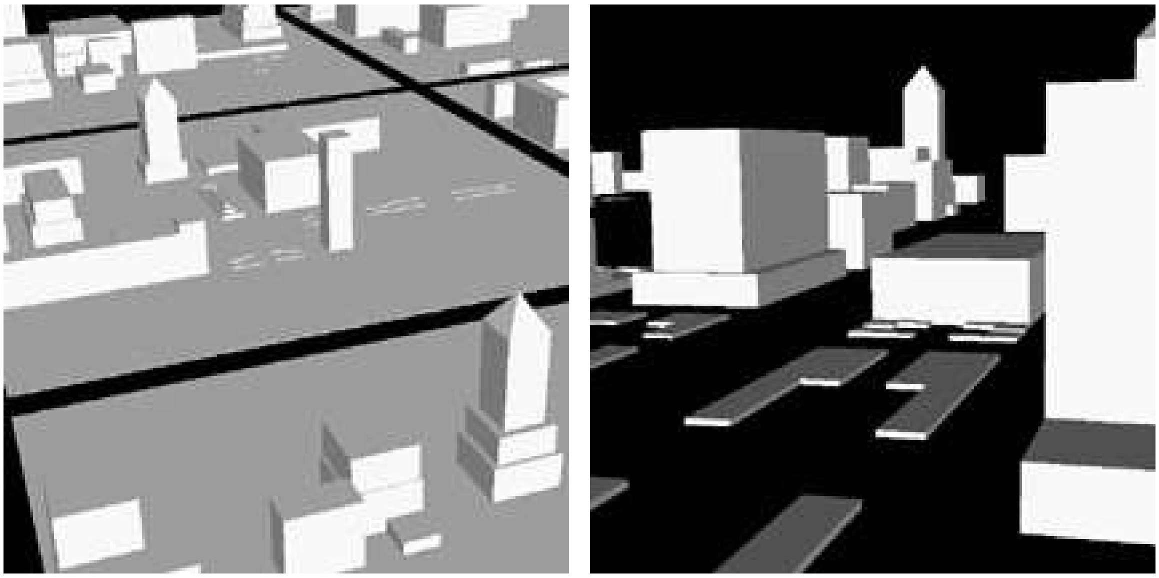


Figure 4.4: Images of a city from Rebecca Xiong's Random City Model Generator.

Chapter 5

Results

In this Chapter I present the results of my algorithm on the available datasets and on test datasets that were generated using techniques mentioned in Section 4.6.

5.1 Generalized Tree Structures

The **Create Surfaces** phase of the aggregation algorithm (described in Section 3.3.1) is heavily dependent on the exact structure of the tree (see Section 3.2.1). For this reason, it is important to test it with both axis-aligned and non-axis-aligned datasets and evenly and randomly split trees.

The rectilinear buildings of Technology Square are axis-aligned to the *Technology Square coordinate system*. Using the matrix transformation option, I rotated a small point cloud dataset from J.P. Mellor's program by the Euler angles 10 degrees, 20 degrees, and 30 degrees. Figure 5.1 shows this dataset and the resulting Surfaces. Although nearest neighbor computations may become more expensive (since the BoundingBox for a diagonal Surface must be larger than that of an axis-aligned Surface of the same size), none of the phases of aggregation appear to perform less well on non-axis aligned datasets.

To verify that the methods used to create and join small Surfaces will work with irregularly structured trees, I tested these phases of aggregation on a point cloud dataset organized in a randomly split tree. The `random_split` option for building the tree simulates tree structures that might arise if leaf nodes are split to optimally separate the contents of the node (as described in Section 3.2). The resulting Surfaces are shown in Figure 5.2. None of the phases of aggregation appear to be negatively impacted by randomly split tree

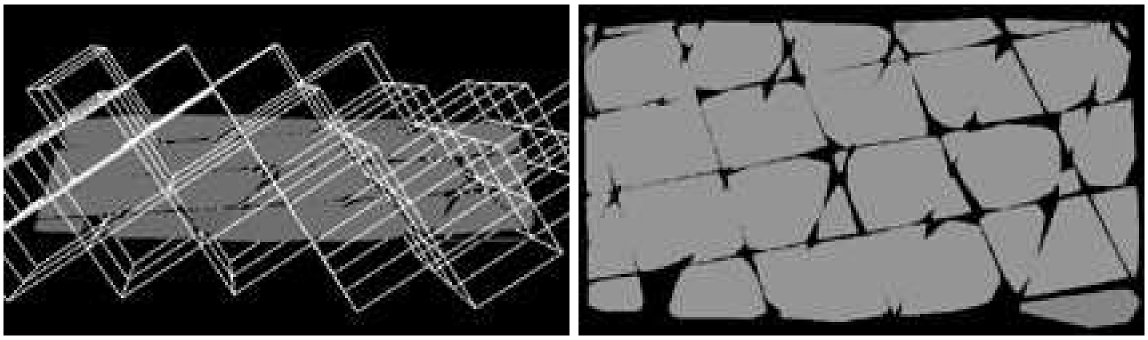


Figure 5.1: A non-axis-aligned point cloud dataset shown from above with the tree structure and the corresponding irregularly shaped Surfaces.

structures.

5.2 Point Datasets

The first dataset I worked with extensively was J.P. Mellor's point cloud dataset from the *Synthetic Image Dataset*. Most of the outliers and the worst of the noise were successfully culled by creating Surfaces only for tree nodes with more than a minimum number of Points. However, a strange false correlation caused problems. The dataset contains clusters of Points that are less than an inch away from each other. Once I loaded all of the Point dataset files I noticed that these clusters of Points formed a circle around the buildings. J.P. Mellor's program estimated the depth of some image pixels to be zero (on the image plane of the camera) which placed them in very close three-dimensional proximity to each other. Figure 5.3 shows the misleading Point clusters.

These clusters of Points result in tiny surfaces whose surface density (Points/area) is several orders of magnitude larger than that of surface patches representing actual facades. My original density metric compared each Surface's density to the greatest surface density currently in the system. Removing surfaces with poor density (according to this metric) caused correct surface patches to be culled in favor of these very dense, but incorrect, tiny surfaces near each camera.

I introduced the minimum Surface area requirement described in Section 3.4.1 to fix this problem. Also, my new surface density metric compares surface densities using the global parameter `average_points_per_building_side` instead of the greatest surface density in the system. Surface densities much greater than the global parameter could

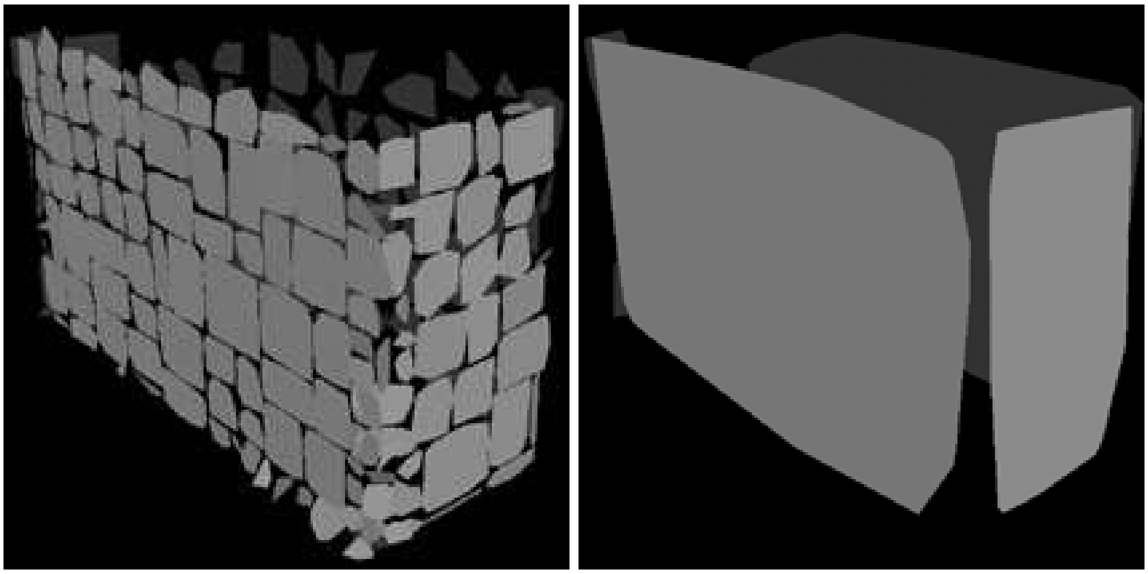


Figure 5.2: Surface patches fit to a point cloud dataset organized in a randomly split tree and the large surfaces resulting from the Extend Excellent Surfaces subphase of **Extend Surfaces**.

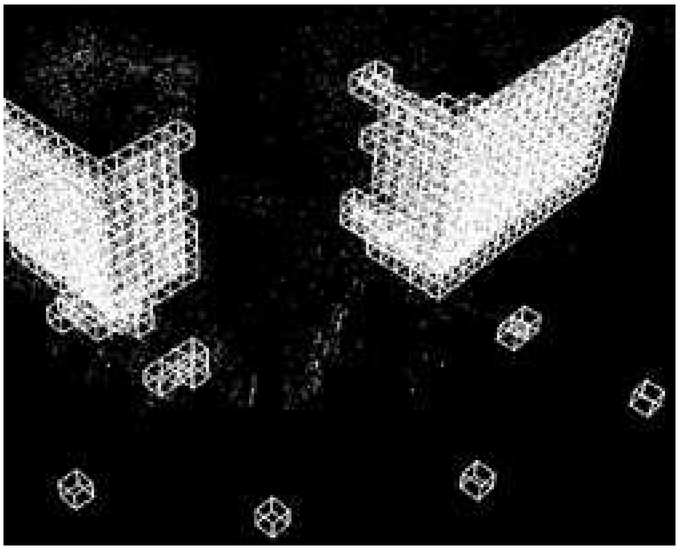


Figure 5.3: A Point cloud dataset within a finely subdivided tree. All leaf tree nodes containing at least 20 Points are displayed. The tree nodes containing the four cameras are drawn (in the foreground of this image) because at least 20 pixels from each image are calculated to have depth 0.

indicate that this user-defined variable is incorrect for the current dataset. In addition, the system could use information about the camera to immediately discard all building fragments which are too close to any of the camera positions, since the cameras must be located in open space.

The results of the aggregation algorithm on J.P. Mellor's point cloud dataset of Technology are shown in Figure 5.4. Figure 5.5 shows the aggregation of a test point cloud dataset created from a non-axis-aligned city from Rebecca Xiong's Random City Model Generator [31].

5.3 Surfel Datasets from the *100 Nodes Dataset*

The Surfel datasets from J.P. Mellor's program have many erroneous Surfels. Removing Surfels without a minimum number of neighboring Surfels that agree about normal and plane position (as described in Section 3.4.4) works well for these datasets. Figure 5.6 shows a dataset of Surfels that cover three facades of a rectilinear building. Very few Surfels were available for the fourth facade (which faces the tree– and foliage–rich courtyard). The few Surfels on this facade were all removed since they had few neighbors. Sampling variations like this in the data will always be a problem; the challenge is to automatically choose correct values for the parameters of the remove outliers algorithm.

5.4 Surface Datasets

The Surfaces produced by Satyan Coorg's algorithm from the *100 Nodes Dataset* consist of vertical facades. Some of the patches are narrow and should be joined with neighboring facades, and some Surfaces are very small and short compared to the other Surfaces. They were probably created for vertical lines corresponding to curbs, steps, etc. Figure 5.7 shows the results of the aggregation algorithm on this dataset.

I also tested the Intersect Surfaces subphase of the **Match Edgels** phase of aggregation on a non-axis-aligned test Surface dataset that was created by jittering the surfaces of a model from Rebecca Xiong's Random City Model Generator [31]. See Figure 5.8.

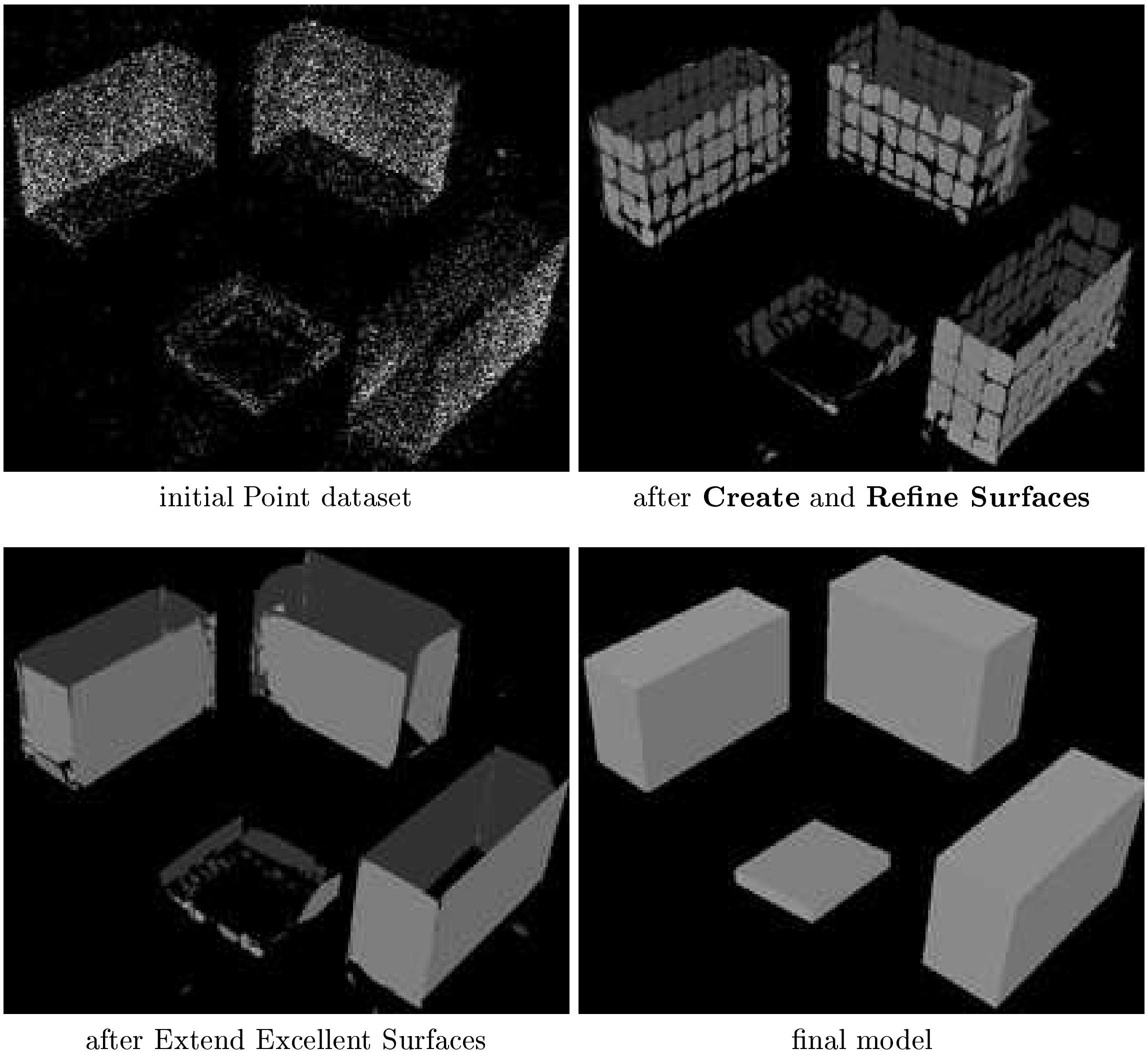
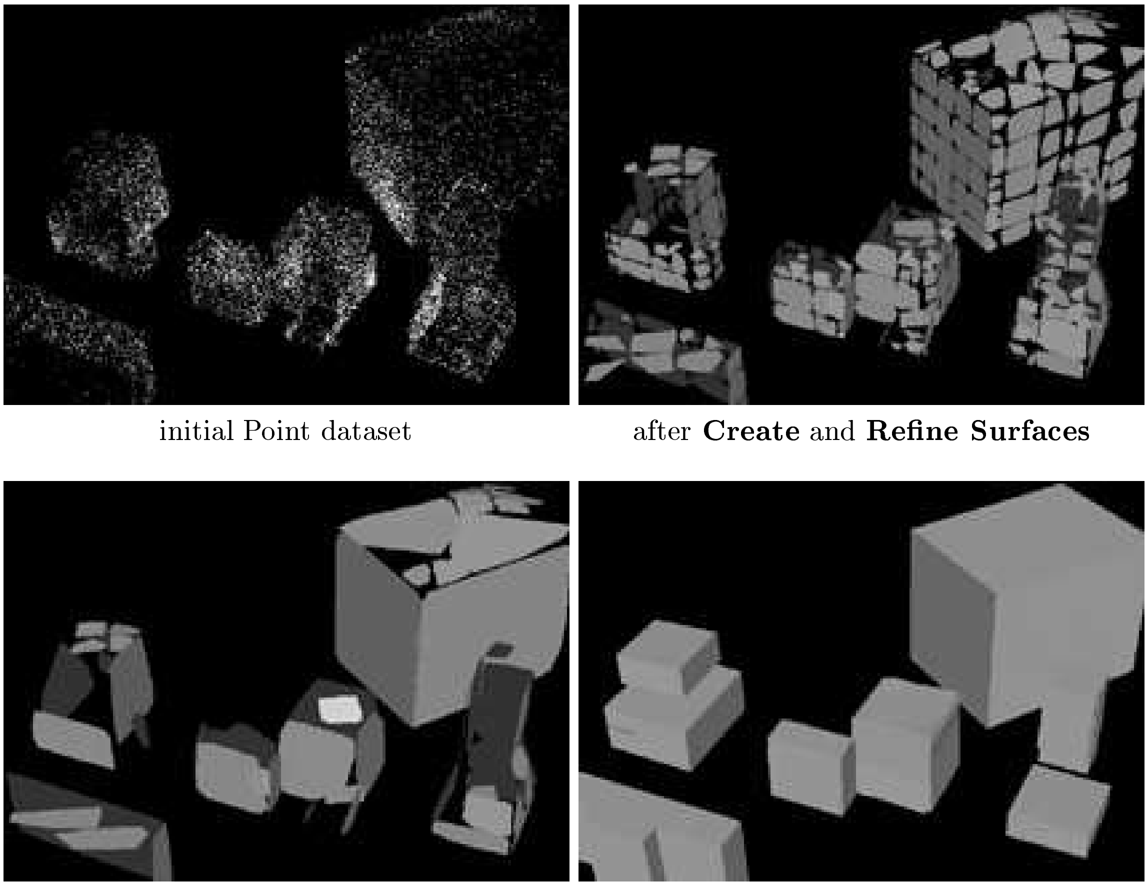


Figure 5.4: Results of the aggregation algorithm on a dataset of 30,000 points. The dataset includes about 2% of the Points J.P. Mellor's algorithm recovered from the *Synthetic Image Dataset* and approximately 8,000 Points which I generated from the surveyed model to cover the inwardly facing facades.



initial Point dataset — after **Create** and **Refine Surfaces**

after **Extend Surfaces** — final model

Figure 5.5: Aggregation of a synthetic 25,000 Point dataset. Some of the facades and roofs of the model were sparsely sampled, but the final Block model is mostly complete. Figure 5.8 shows a jittered version of the Surface dataset from which this point cloud dataset was generated.

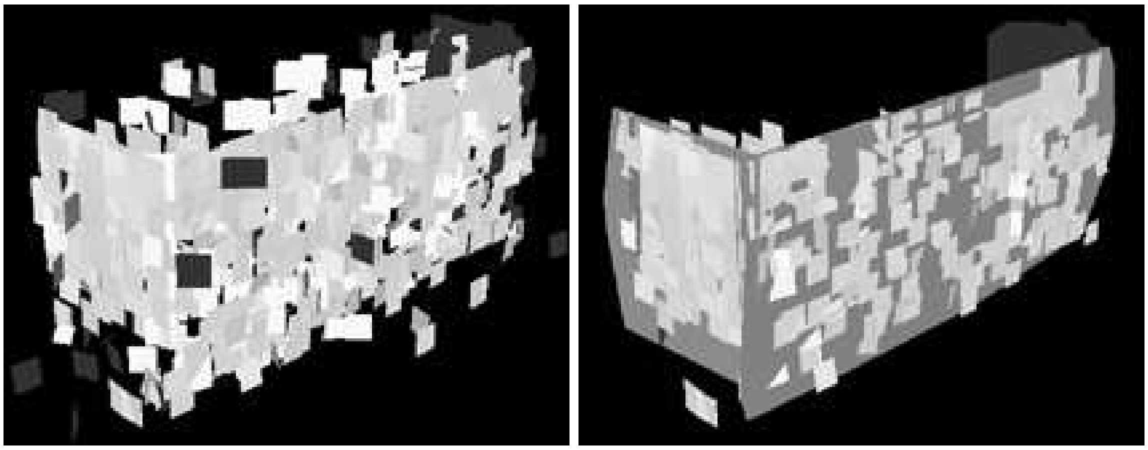
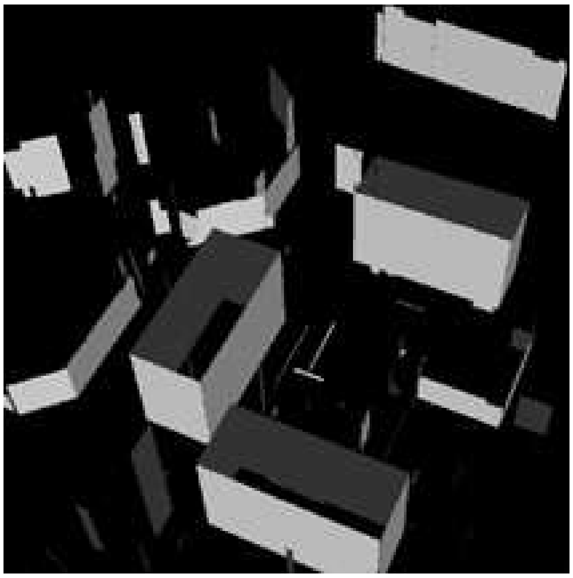
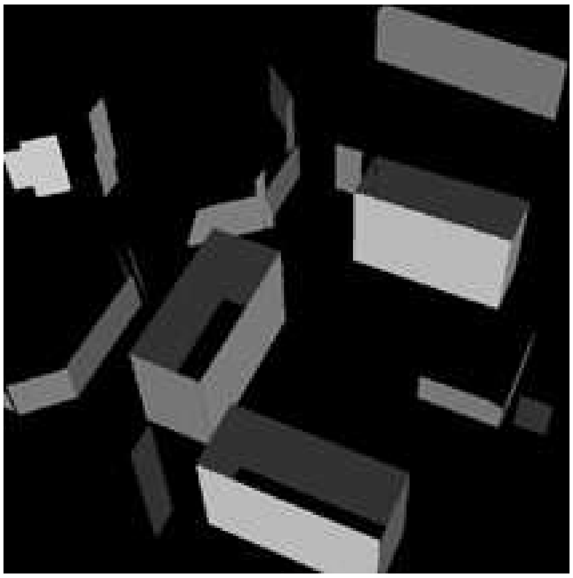


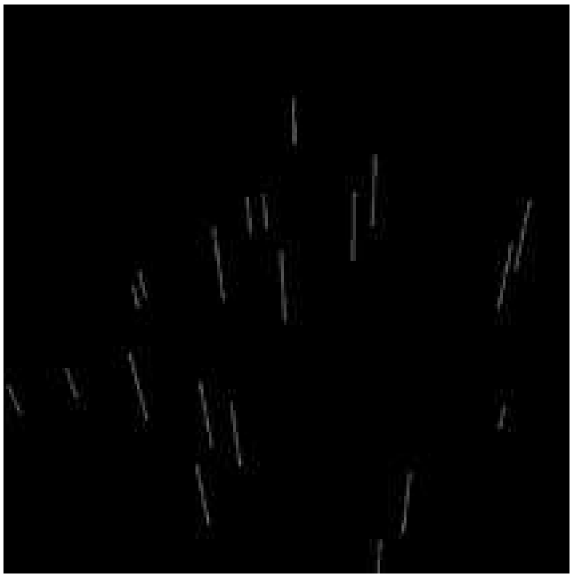
Figure 5.6: Image of the approximately 1,500 original Surfels and the Surfaces fit to three of the facades after removing approximately 700 noisy Surfels.



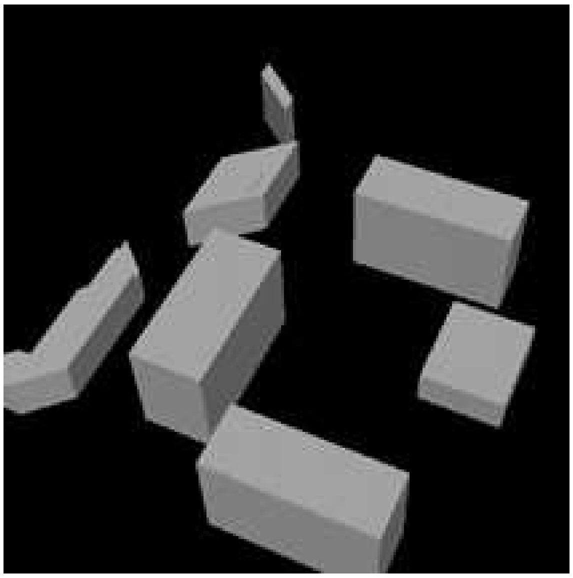
initial Surface dataset



after Extend Excellent Surfaces
and remove small surfaces



after Intersect Surfaces (Edgels only)



final model

Figure 5.7: Satyan Coorg's Surface dataset after various phases of the aggregation algorithm.

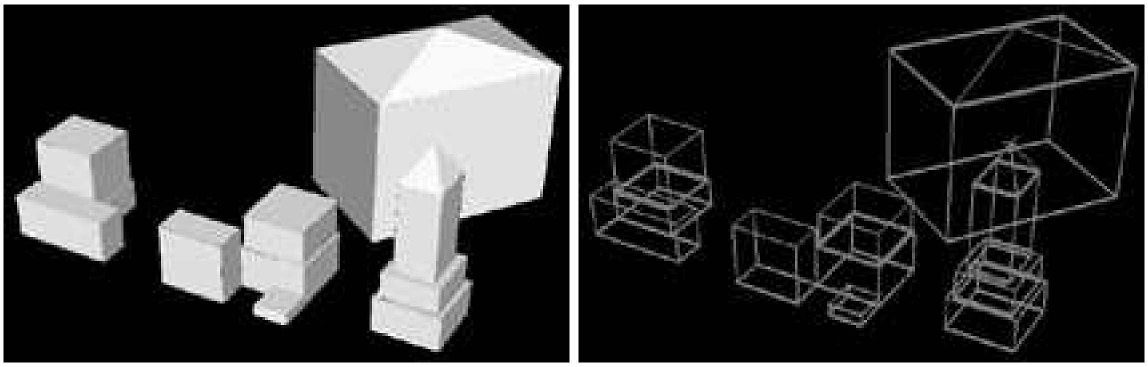


Figure 5.8: A test Surface dataset of a city skyline which is not axis-aligned and the Edgels created by intersecting neighboring Surfaces.

5.5 Aggregating Planar and Spherical Surfaces

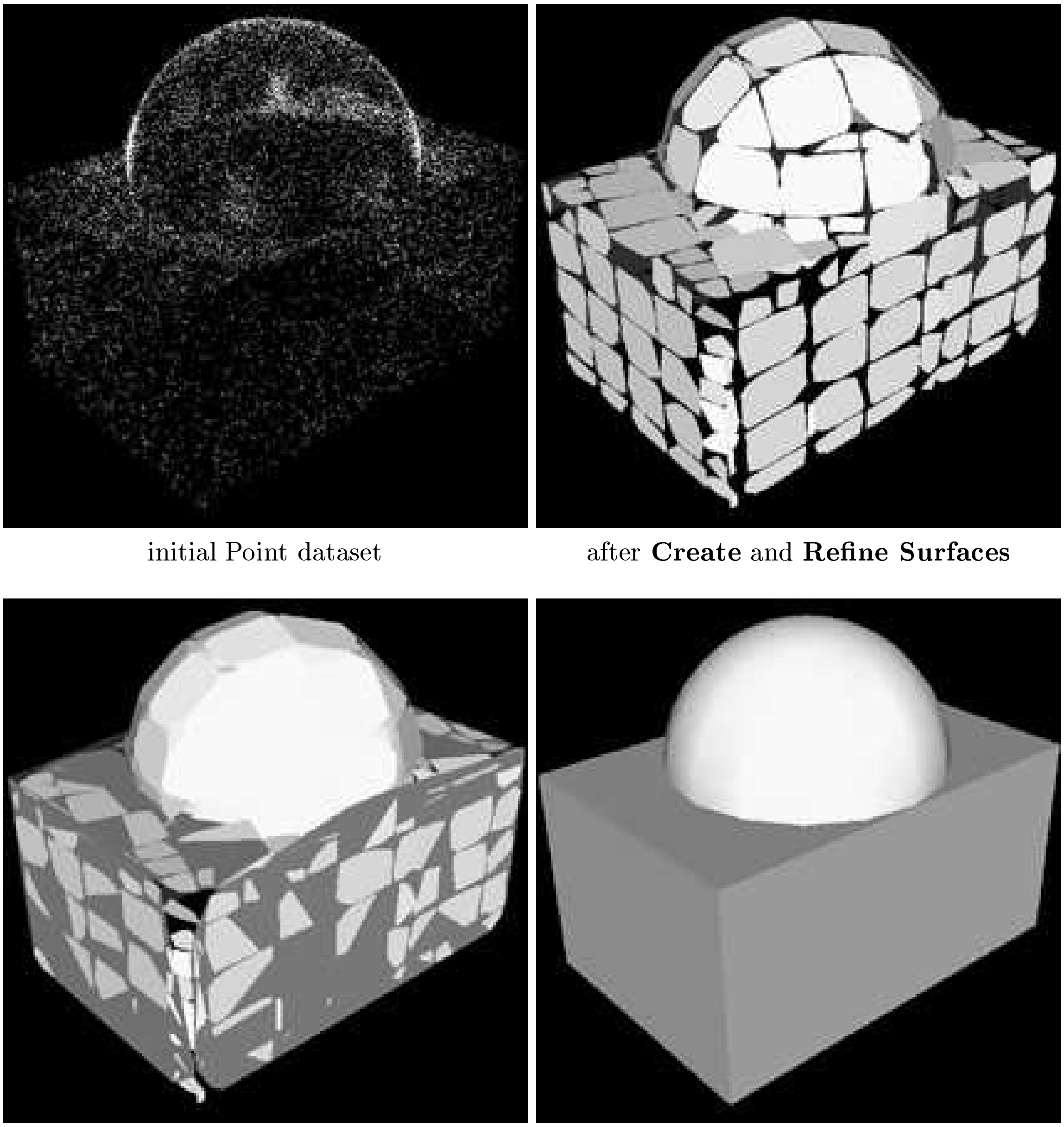
To test the aggregation of planar and spherical surfaces, and the interaction between the Extend Excellent Surfaces and Try Quadrics subphases of **Extend Surfaces**, I created a simple building model which has a rectilinear base and a spherical dome. I created a point cloud dataset of approximately 17,000 Points from this model and ran the aggregation algorithm on this dataset.

Creating surfaces in an environment which has both planar and spherical datasets is complicated since the algorithm must correctly identify which data elements fit each type of Surface. The requirements listed for the Try Quadrics subphase of **Extend Surfaces** in Section 3.3.3 try to do just that. The Use Best Fit Data subphase of **Refine Surfaces** also helps by removing surface patches that have been inadvertently added to the wrong Surface.

5.6 Aggregating Points, Surfels and Surfaces

In the previous sections I have shown the results of reconstructing Surfaces and Blocks from single types of data. I have also created test datasets that simulate how several types of data combined would be aggregated. Figure 5.10 shows a collection of data objects — Points, Surfels, and Surfaces — that were produced by J.P. Mellor and Satyan Coorg's algorithms.

The diagram of the aggregation algorithm (Chapter 3, Figure 3.1) shows the phases of aggregation listed in a particular order, and the subphases for each phase described in Section 3.3 are presented and executed in a particular order also. However, the algorithm



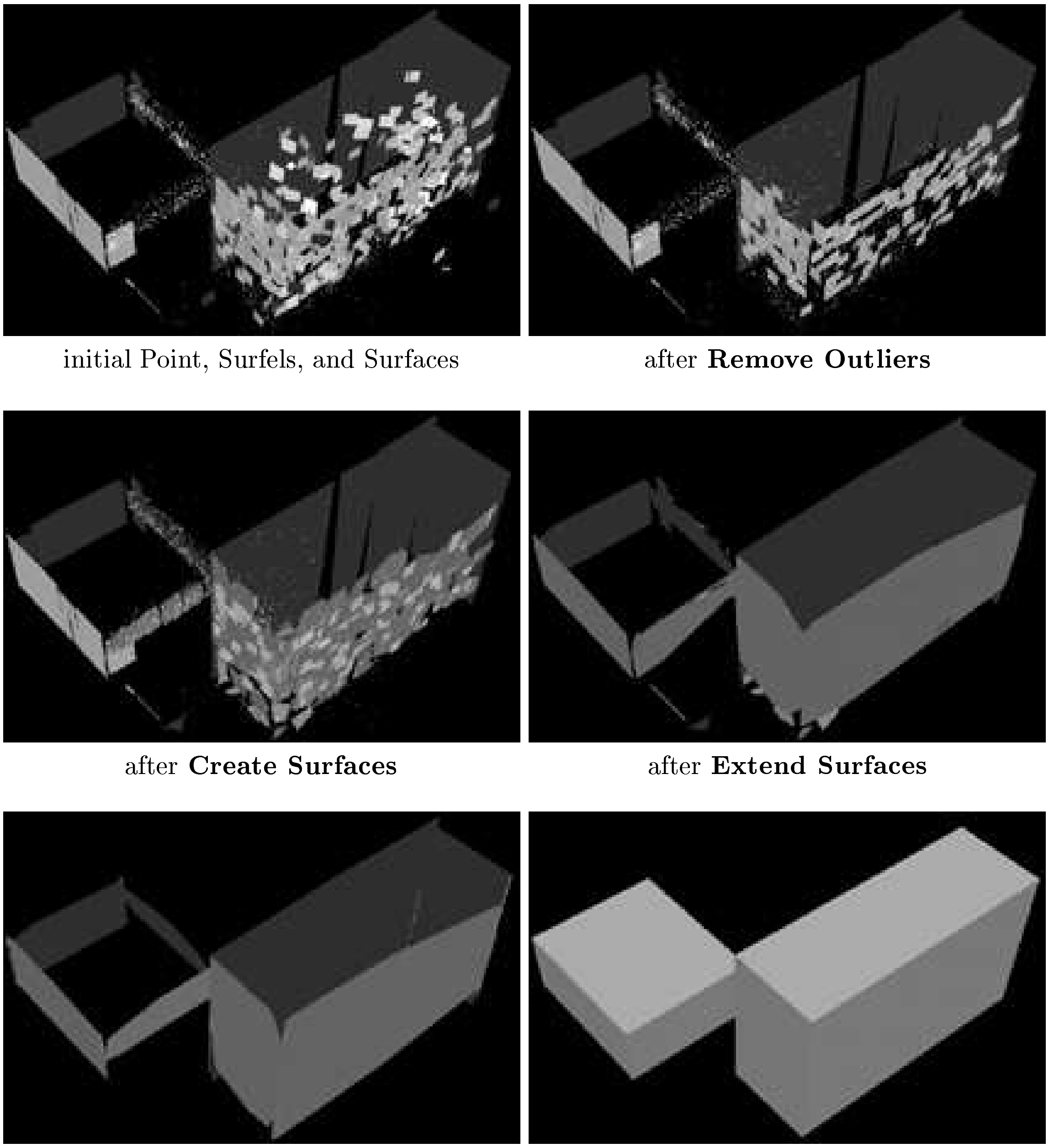
initial Point dataset

after **Create** and **Refine Surfaces**

Extend Excellent Surfaces and Try Quadrics

final model

Figure 5.9: Detecting both planar and spherical Surfaces from a synthetic dataset of approximately 20,000 Points.



initial Point, Surfels, and Surfaces — after **Remove Outliers**

after **Create Surfaces** — after **Extend Surfaces**

after **Match Edgels** — final model

Figure 5.10: Aggregation of Points, Surfels, and Surfaces from J.P. Mellor and Satyan Coorg's programs.

results are not directly dependent on the order of these operations. Also, data can be loaded during the middle of an aggregation session and merged with the current model. It is important for certain operations to happen before others (i.e., if aggregating a dataset of Points and Surfels, one must run **Create Surfaces** before **Extend Surfaces** or there will be no Surfaces to extend). The operations may also be performed repeatedly. Some of the phases work best when performed in alternation with other phases; for example, the Use Best Fit Data subphase of **Refine Surfaces** works well in combination with the Unclaimed Data subphase of **Extend Surfaces**. I have tested different orderings of these phases and they all work fine, as long as the operations are performed repeatedly so that everything converges to a stable final result.

It is also important for the system to easily add to or merge the results of previous aggregations. Currently, the difference between aggregating everything at once and adding to a previously saved aggregation is that the reconstructed objects cannot be refined since their evidence and the important metrics surface density, surface fit, etc. are not available. Once the **Save All** option (described in Section 4.5) is fully implemented, and the complete system state can be reloaded from a saved aggregation session, adding to a previous reconstruction will be no different than adding data during the middle of the aggregation. Merging two or more previous aggregations is more complicated since it may involve joining Surfaces or Blocks that contain disagreeing data.

5.7 Future Work

I have many ideas for small modifications to the program which I did not have time to implement. These include:

- Split tree nodes at a point that maximizes the separation of objects while minimizing the number of objects that must be on both sides of the split (Section 3.2.1).
- Implement the modified surface density metric (Section 3.4.2).
- Improve the limited Inventor parser and implement a VRML parser (Section 4.5).
- Implement the **Save All** operation so that the complete system state is saved (Section 4.5).
- Add UNDO (Section 4.5).

- Add a "record" option to save (in command script syntax) the parameters and commands used during a graphical session (Appendix C).
- Improve the use of color and transparency in the graphical visualization. For example, color could be used to visualize the how well evidence fits to the final model. Enabling higher quality transparency can add more possibilities for visualization, for example, differentiating known empty space and unknown space.

Some more involved ideas for future work include:

- Represent non-convex Surfaces and Blocks by internally manipulating them as several convex objects (Sections 3.1.4 and 3.1.5).
- Fit Point, Surfel and Edgel data to more general quadric surfaces (Sections 3.1.4 and 3.3.3).
- Remove the program's dependence on the 0.5 cut–off value for the surface density and surface fit metrics to distinguish "excellent" Surfaces from "poor" Surfaces.
- The aggregation algorithm currently relies on a large number of parameters, some of which must be manually adjusted for new types of datasets. Appendix C describes modifications to the system that would warn the user if the values might be inappropriate and suggest more reasonable values. Further work could develop methods to iteratively estimate values for these parameters from the data alone.
- Represent empty space, the areas that are known to not contain any structure. Camera positions can do this; perhaps other things can also indicate when something is known to be empty. This will probably help a lot when reconstructing interior spaces.
- Project the original photographic images onto the current model and analyze how well things line up.

Chapter 6

Conclusion

The goal of the City Scanning Project is to build a system that automatically creates accurate three-dimensional models of urban environments. Several user-driven programs to build such models are currently available from other groups but these programs require too much interaction and familiarity with the environment to be adapted to our problem.

Image analyzing programs being developed within the group take different approaches to environment reconstruction and produce different types of building fragments. The aggregation algorithm presented merges these fragments, combining the strengths of each to produce a more complete model than any of them could create alone. The aggregation algorithm can handle incomplete datasets, which is important since the image analyzing programs may have trouble with foliage or unusual structures in the environment. The algorithm joins building fragments into a final model, appropriately weighting disagreeing fragments and removing duplicates when fragments agree.

The phases of the aggregation algorithm can be used iteratively to create, refine, and extend the surfaces and volumes that compose the final model. These objects are organized by position in a k-d tree to make searching operations more efficient and to adaptively separate and group data that gives evidence for structures in the urban environment. The system uses configurable metrics to weight data evidence and evaluate the objects reconstructed from the data. The graphical interface of the program facilitates comparison of building fragments produced by the image analyzing programs and inspection of how they are used to create the final model. The aggregation algorithm creates a cohesive model where previously the group had only the results of several isolated programs.

Appendix A

Aggregation Parameters

Global parameters are mentioned throughout the description of the aggregation algorithm in Chapter 3. Some of these parameters are necessary for the algorithm to produce meaningful results, while other parameters are provided to optimize the aggregation algorithm both during development and during future use to adjust to data with varying levels of noise and outliers. Each parameter is set in the input specification file (described in Appendix C) or from the GUI interface (described in Appendix B). If a parameter is unspecified, a default value defined in the C++ header file `"defaults.h"` is used. These default values can be changed by modifying this file and recompiling the executable.

Below I describe each of these parameters, and give their type (integer, floating-point decimal, or boolean), range of legal values, and default value. The parameters marked with ▷ should be reviewed when incorporating new types of datasets or data from new environments, as the default values may not be appropriate.

A.1 Relative Size and Level of Detail

My aggregation algorithm operates independently of the scale or orientation of the building fragments to the urban environment. Instead, it relies on the following parameters to indicate the scale of the environment to be reconstructed — a single building, a city block of buildings, or an entire city — and the desired quality of the model to be created.

▷ `average_building_side` type: `float` range: $(0, \infty)$ default value: 1000.0

Indicates the average size of the buildings in the scene relative to the unit of measurement. For example, the Technology Square environment contains large ten-story office buildings and our current datasets are measured in inches, so the value 1000 is appropriate for these datasets. However, for a suburban environment of small single-family homes measured in feet, a value of 50 would be more reasonable.

▷ `nodes_per_side` type: `float` range: $(0, \infty)$ default value: 2.0

Specifies the minimum number of leaf tree nodes that should cover a length of size `average_building_side` (see description in Section 3.2.1). Larger values of this parameter instruct the system to reconstruct the environment to a finer level of detail, provided that adequate data is available.

`random_split` type: `bool` range: 0 or 1 default value: 0

If `random_split` = 0, the program subdivides tree nodes at the midpoint of their largest dimension. If `random_split` = 1, the program splits tree nodes at a point selected from a Gaussian distribution with around the midpoint.

A visualization of the current values of these parameters is available in the GUI by viewing the *Average Building* (see Figure B.6 in Appendix B). A transparent cube of volume `average_building_side`^{3} is drawn with a wireframe cube inside it that represents the largest tree node possible (according to `nodes_per_side`). The value of `average_points_per_building_side` (described in Section A.4) is shown by randomly placing that many Points on one face of the Average Building. Also, an estimate of the necessary tree depth for the current scene boundingbox is displayed (described in Section 3.2). The system could be modified to warn the user if the proposed tree depth is too large for practical computation.

If the algorithm fails to produce satisfactory results for the data presented, it is likely that the current value for one of these parameters is inappropriate, or that the data available is incomplete or too noisy. The system could be modified to notify the user if the density values for Surfaces created from Point clouds are not within an order of magnitude of the expected density. The program could also suggest more appropriate parameter values for consideration.

A.2 Load File Options

The following options are available when loading a building fragment dataset into the system, as described in Section 4.5.

▷ `load_sample` type: `float` range: $(0, 1]$ default value: 1.0

Indicates that the system should use only a fraction of the building fragments contained in a particular data file. A value of 1.0 indicates that all objects should be loaded, while a value of 0.2, for example, indicates that approximately one-fifth of the objects should be loaded. The sampling is done randomly. This option can be used to reduce the computation necessary to process extremely large Point datasets or create sparse synthetic Surface datasets from CAD models.

`data_jitter` type: `float` range: $[0, 1]$ default value: 0.0

This parameter can be used to create synthetic noisy datasets from geometrically precise models. Each 3D point used to describe each data fragment is jittered by adding a randomly directed 3D vector whose magnitude is a Gaussian distribution of normal $0.1 \times$ `data_jitter` $\times$ `nodes_per_side` $\times$ `average_building_side`. See also Figure 4.2 in Appendix B.

A.3 Algorithm parameters

The following parameters are used by specified phases of the aggregation algorithm, which are described in Section 3.3.

`tree_count` type: `int` range: $[1, \infty)$ default value: 10

The **Create Surfaces** phase of aggregation (see Section 3.3.1) creates a planar Surface for each leaf tree node that contains a minimum number of data elements. The `tree_count` parameter specifies this minimum. When creating Surfaces from point cloud datasets with many outliers, it is practical to increase this parameter to decrease the number of false Surfaces created. The system could be modified to suggest an appropriate value for this parameter as a function of the parameters described in Section A.1. For example, the following equation calculates the number of data points needed to produce a high density Surface with area that "fills" 10% of a leaf node.

$$\texttt{tree\_count} = \frac{\texttt{average\_building\_side} \times \texttt{average\_points\_per\_building\_side}}{10 \times \texttt{nodes\_per\_side}^2}$$

`surface_iterations` type: `int` range: $[1, \infty)$ default value: 2

Specifies the number of iterations to be performed in the iterative re-weighted least squares algorithm used in the **Create Surfaces** phase of aggregation. Fua [13] found that five iterations were usually sufficient to reach a stable solution. Since my algorithm also performs **Refine Surfaces** and **Extend Surfaces** phases (Section 3.3.2 and 3.3.3), I have not found it necessary to perform as many iterations.

`additional_tree_depth` type: `int` range: $[0, \infty)$ default value: 3

This parameter controls the number of additional tree subdivisions allowed during the **Refine Surfaces** subphase Subdivide Tree. Environments with a great disparity between the size of the largest and smallest buildings, or having varying levels of sampling or detail should use a higher value for this parameter.

A.4 Noise and Outlier Error Bounds

The following parameters control how noise and outliers are removed from the system (described in Section 3.4). Since the building fragment datasets are acquired and generated with different methods, the types of noise and outliers present will vary. These parameters may need to be tuned to the different types of data files available. The user specifies which of these criteria should be used with checkbuttons in the graphical interface (Appendix B) or with boolean flags in the command script (Appendix C): `use_minimum_length`, `use_minimum_area`, `use_minimum_volume`, `use_surface_normal`, `use_surface_density`, and `use_surface_fit`. .

`minimum_length` type: `float` range: $[0, 1]$ default value: 0.3

If `use_minimum_length` = 1, the program removes all Edgels of length less than `minimum_length` × `average_building_side`.

`minimum_area` type: `float` range: $[0, 1]$ default value: 0.2

If `use_minimum_area` = 1, the program removes all Surfaces with area less than `minimum_area` × `average_building_side`^{2}.

`minimum_volume` type: `float` range: $[0, 1]$ default value: 0.1

If `use_minimum_volume` $= 1$, the program removes all Blocks having volume less than `minimum_volume` $\times$ `average_building_side`^{3}.

The minimum length of Edgels, minimum area of Surfaces, and minimum volume of Blocks could be combined in a single "minimum size" variable which depends on the level of detail parameter `nodes_per_side`. For example, if `nodes_per_side` $= 3.0$ then reasonable values for minimum length, area, and volume as a fraction of the `average_building_side` might be $\frac{1}{3}$, $\frac{1}{9}$, $\frac{1}{27}$ respectively.

`surface_normal` type: `float` range: $(0, 1)$ default value: 0.25

`normal_count` type: `int` range: $(0, \infty)$ default value: 4

If `use_surface_normal` $= 1$, the program removes all Surfels which do not have at least `normal_count` other Surfels nearby with nearly the same normal. If the angle between the Surfels is small relative to `surface_normal`, they are said to have nearly the same normal. See also Section 3.4.4.

▷ `average_points_per_building_side` type: `int` range: $[1, \infty)$ default value: 500

Estimates the number of point samples that can be expected on a building facade with area `average_building_side`^{2}. The expected density (points/area), described in Section 3.4.2, is used to compare Surfaces created from Points with Surfaces created from Surfel, Edgel, and Surface data. J.P. Mellor's point cloud datasets sampled at 5% have about 1000 points per 1000 x 1000 inch Surface, while some of the synthetic point cloud datasets generated from Inventor models have about 500 points per 1000 x 1000 inch Surface. If `use_surface_density` $= 1$, the program removes all reconstructed Surfaces whose surface density value (described in Section 3.4.3) is less than 0.5.

`surface_fit` type: `float` range: $[0, 1)$ default value: 0.5

If `use_surface_fit` $= 1$, the program removes all reconstructed Surfaces whose surface fit value (described in Section 3.4.3) is less than `surface_fit`.

Appendix B

Program Interface

In this Appendix I describe how to use the graphical implementation of the aggregation algorithm presented in Chapters 3 and 4. The program interface is organized into six windows: **Display**, **Load Files**, **Aggregation Options**, **Sphere Visualization**, **Matrix**, and **Bounding Box**. All non-display options available through the graphical user interface (GUI) are also available in the command script which is described in Appendix C.

A graphical aggregation session may be initialized by a command script that sets parameter values and performs aggregation actions. After the command script has completed, the user can visually inspect the results and perform further operations. Also, the GUI may be used to inspect the saved results of a non-graphical session.

B.1 File Options

The aggregation process begins by loading the available building fragment data files. From the **Load Files** window (shown in Figure B.1) the user may:

- Load a new file and use the following options:
  - Apply a transformation matrix to each file to put it in the common coordinate system. The **Matrix** window (shown in Figure B.2) allows the user to specify the translation, rotation of the matrix, or input the matrix directly.
  - Randomly sample the file, by setting the `load_sample` parameter. For example, if this value is set to 0.2, approximately one fifth of all the data elements in the file will be loaded. This option allows the user to more efficiently process datasets that contain many more data elements than needed to reconstruct a

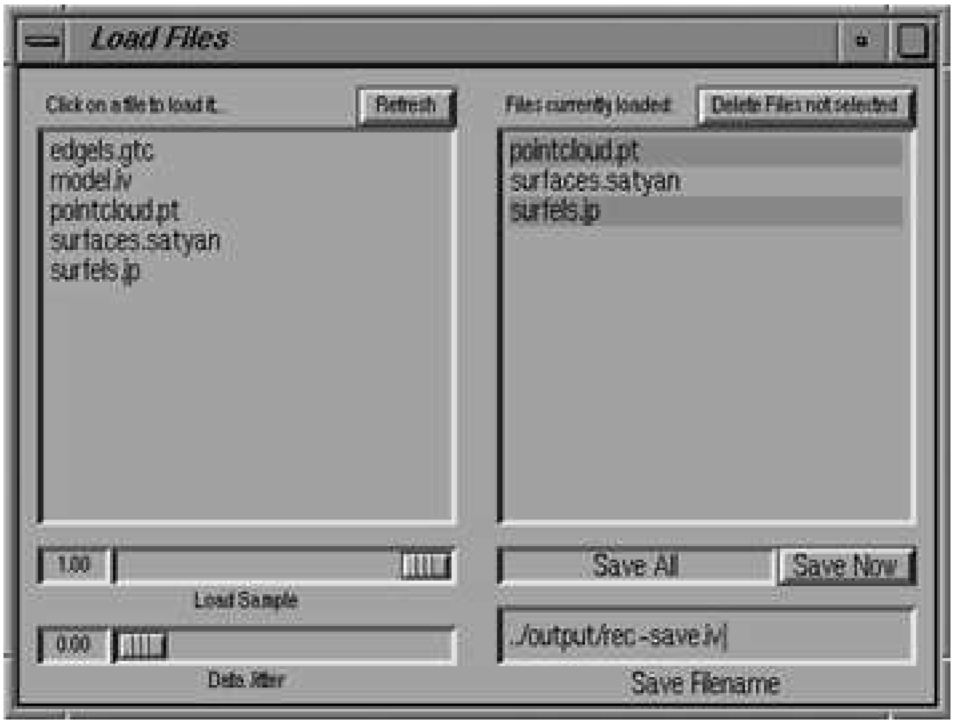


Figure B.1: Snapshot of the **Load Files** window. Three files have been loaded but only two of them are currently selected for display.

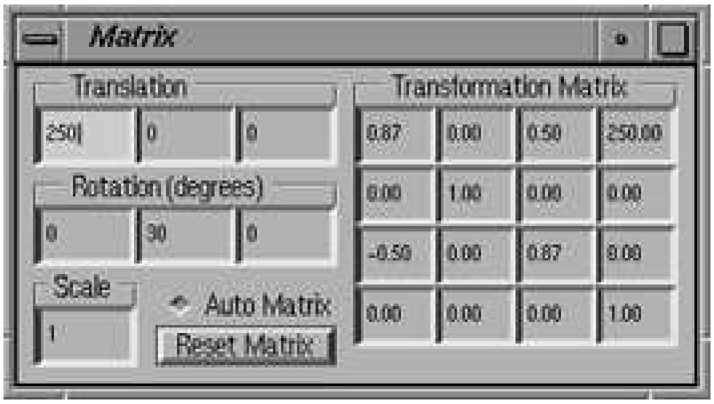


Figure B.2: Snapshot of the **Matrix** window which has been set to translate loaded files by 250 units along the positive x-axis and rotate 30 degrees around the y-axis. When the "Auto Matrix" check button is not selected, a transformation matrix may be input directly.

scene to the desired level of detail. For example, a spare sampling of point cloud datasets generated by long-range laser scanners may be used to create a rough sketch of the building masses in an environment. Then, after aggregation of other available building fragments, the point cloud files can be sampled more densely, as needed, to create a more detailed model.

  – Add Gaussian noise to the data elements with the Data Jitter option. Each 3D point used to describe each data fragment (Point, Surfel, Edgel or Surface) is jittered by a randomly oriented vector whose magnitude is normally distributed around the specified percentage of the `average_building_side` (see also Section A.2). See also Section 4.6 and Figure 4.2.

- Selectively view a subset of all files currently loaded. Each object is tagged with a file identifier so it can be manipulated later by its originating file. Reconstructed objects and objects generated for testing (see Section 4.6) have special file identifiers.

- Delete (unload) files from the system that are not currently selected for display. For example, the user can delete all objects belonging to a particular file if the file was improperly loaded.

- The objects currently in the system may be saved by using one of the following save options: **Save All**, **Save Reconstructed**, **Save Generated** and **Save Final** (described in Section 4.5).

B.2 Display Options

The main display window (shown in Figure B.3) of my program has a large OpenGL canvas in which the data and reconstructed objects are visualized. The view can be translated, rotated, and zoomed in and out with the mouse. The following options are available on the **Display** window:

- The objects currently in the system can be viewed by object type by selecting the appropriate checkboxes (Point Data, Surfel Data, Edgel Data, Surface Data, Reconstructed Edgels, Reconstructed Surfaces, and/or Reconstructed Blocks).

- Objects that are evidence for reconstructed objects can be hidden so only *top level* objects are displayed. The remaining *unclaimed* data objects and reconstructed objects

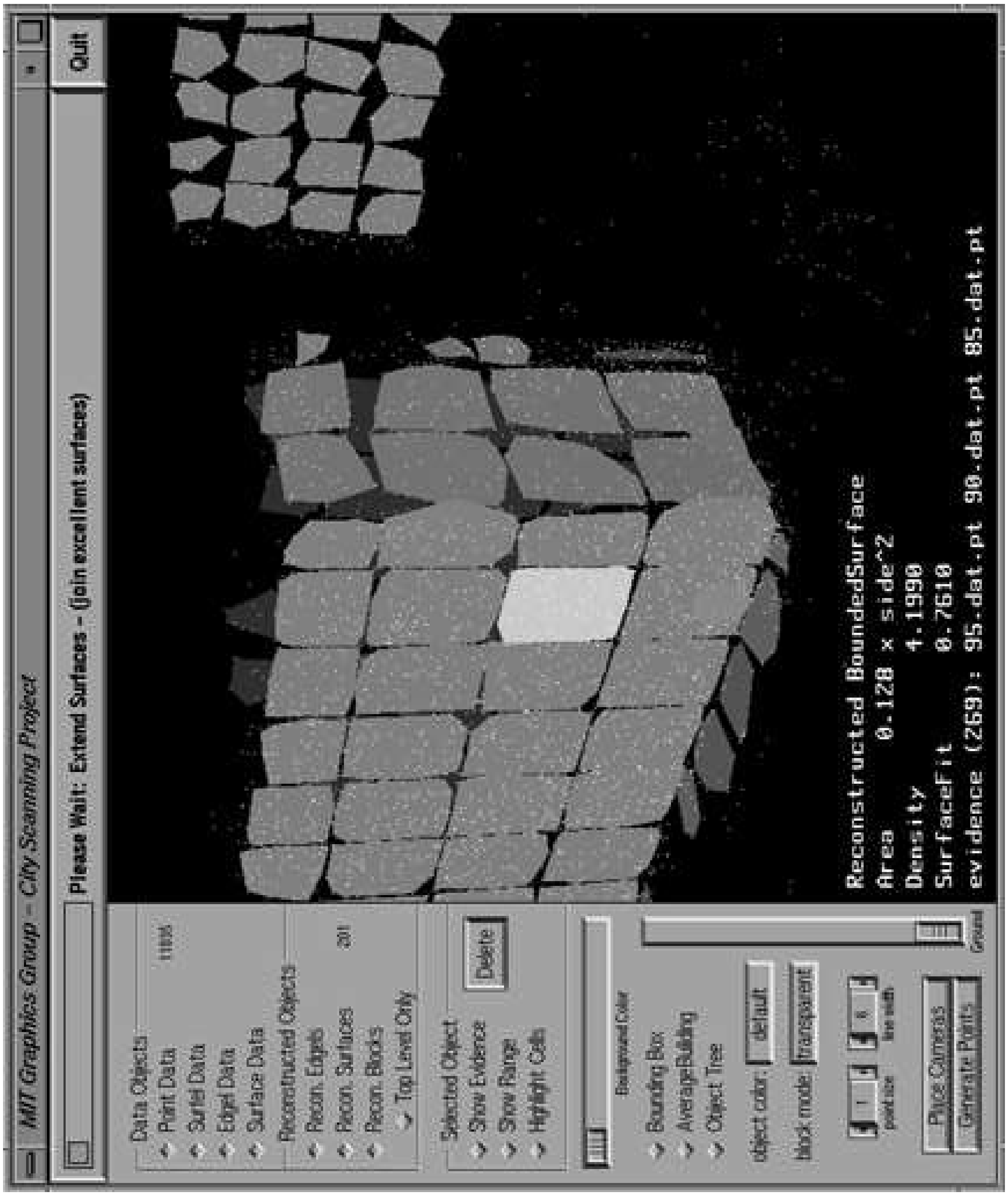


Figure B.3: Screen capture of the **Display** window showing initial surface patches fit to a Point cloud dataset. The selected object is a reconstructed Bounded Surface created from data found in three files.

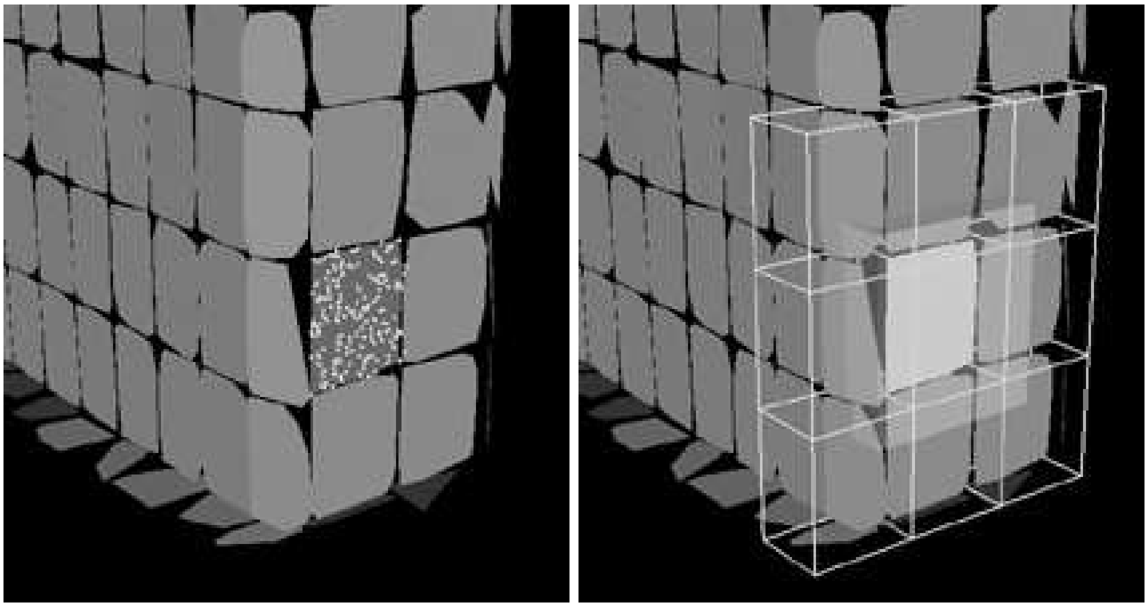


Figure B.4: A selected Surface shown with its evidence and with its neighboring region and tree nodes highlighted.

are displayed.

- The user can select an object with the left mouse button and information about the object is displayed in the lower left corner of the display. This information includes which file the object was loaded from (if it is a data object) or which files contained the data that was used to create the object (if it is reconstructed). The length of Edgels, the area of Surfels and Bounded Surfaces, and the volume of Blocks are displayed. Also the plane fit and density values for reconstructed Surfaces are displayed. Several options are available for the selected object:
  - Display the evidence for the selected object (if it is reconstructed).
  - Highlight the nodes of the tree and the region of space that will be searched when looking for neighbors of the selected object (see Figure B.4).
  - Delete the selected object. This option is *not* intended to be used to remove outliers during aggregation, but rather to be used during inspection of the data and results to remove objects that are "in the way".
- The background color is by default black, but to more easily view some of the coloring scheme options the color can be changed with a slider from black to blue to white.

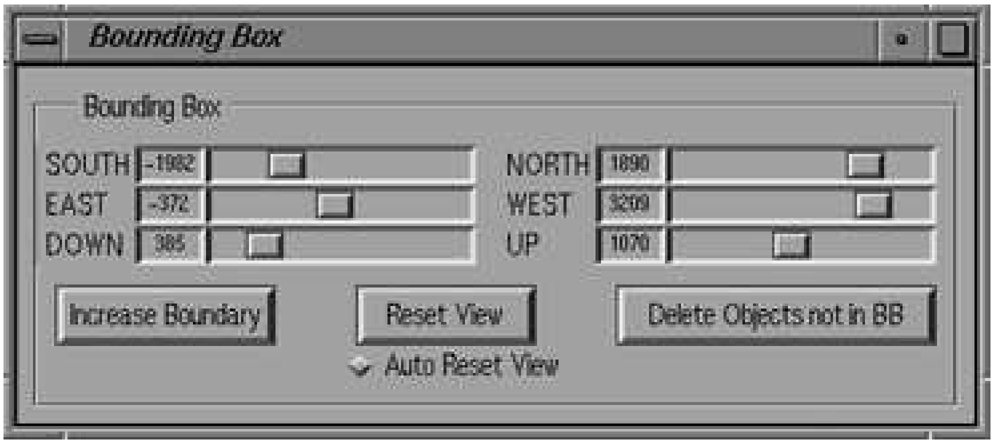


Figure B.5: The **Bounding Box** window has sliders for the upper and lower bounds in each dimension. The minimum and maximum values of each slider are automatically set to the bounding box of all objects in the scene.

- The system maintains two bounding boxes, a scene bounding box which fully contains all objects currently in the scene, and a selected bounding box which is by default equal to the scene bounding box. The user can display and adjust the selected bounding box with the **Bounding Box** window (shown in Figure B.5). By default the **Display** window focuses on the selected bounding box.

- View a transparent block of size `average_building_side`^{3} with a wireframe box showing the maximal tree leaf node size (as specified by `nodes_per_side`) inside of the block and `average_points_per_building_side` Points displayed on one face of the block (see Figure B.6). This option is very helpful in verifying that the current values for these three parameters are appropriate.

- View which leaf nodes of the object tree have at least `tree_count` number of elements.

- The coloring scheme can be adjusted:

  - By default the data objects are white, the reconstructed Edgels and Surfaces are green, the reconstructed Blocks are purple and the selected object (if any) is yellow.

  - The objects may be colored by normal with the RGB values of the color set to the XYZ components of the normal vector.

  - The surfaces may be shaded by surface density (described in Section 3.4.2).

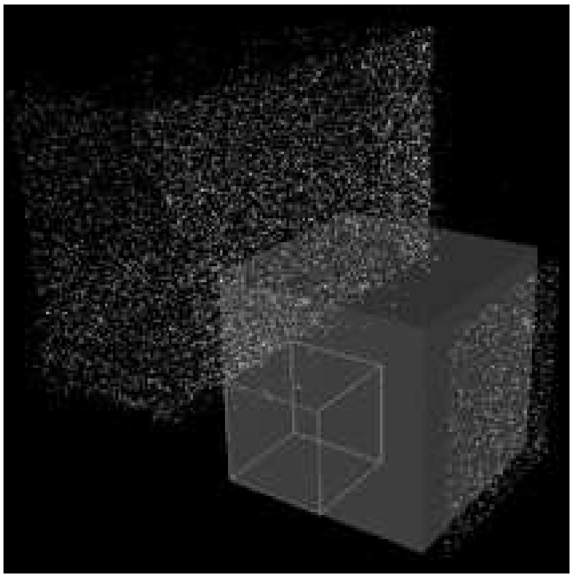


Figure B.6: A Point dataset is displayed in the background of this image. In the foreground is a transparent block representing the average building (the parameter `average_building_side` is 1000 units). A wireframe cube inside the building shows the maximal leaf node of the tree (2.0 `nodes_per_side`), and 1500 points are displayed near one side of the building to visualize the parameter `average_points_per_building_side`.

  – The surfaces may be shaded by surface fit (described in Section 3.4.3).

- The block display mode can be changed. By default the hypothesized block surfaces are shown transparently. The blocks can also be displayed as solid (all surfaces opaque) or open (with only the known surfaces drawn).

- By default Points are drawn with a single pixel and if the selected object is an Edgel, it is drawn six pixels wide. The size of a Point and the width of the selected Edgel can be adjusted.

- Generate synthetic Point cloud datasets (see Section 4.6). First the user presses the *Place Cameras* button and a ground plane appears which can be adjusted with the Ground Height slider. Next, the user selects "camera" locations on the ground plane with the left mouse button. Then the user presses the *Generate Points* button and 1000 randomly directed rays are emitted from each camera point. A Point is created at the intersection point of each ray with the nearest object. If a ray does not intersect any object, no Point is generated. Figure 4.3 shows a synthetic dataset generated from a single camera. The Generate Points button was selected several times to generate roughly 1000 Points.

- A system message bar at the top of the **Display** window shows which phase and subphase of aggregation is currently being executed.

- A progress meter estimates how much of the current phase has been completed.

B.3 Aggregation Options

The **Aggregation Options** window (shown in Figure B.7) allows the user to change the values of the important aggregation parameters listed in Appendix A. The phases of the aggregation algorithm described in Chapter 3 can be executed by pressing the corresponding buttons.

- The user controls the tree size and structure with four tree parameters: `average_building_side`, `nodes_per_side`, `average_points_per_building_side`, and `additional_tree_depth`. The user can also select `random_split` to indicate that the tree should not to be split evenly. The estimated tree depth based on the current scene boundingbox and the values of the above parameters is displayed. This number helps the user check to see if the settings are appropriate. Also a message "Tree Valid" or "Tree Invalid" indicates whether the tree must be built before performing the next tree-dependent operation. The system will automatically build the tree when necessary.

- The user can choose which of the **Remove Outliers** criteria should be used by selecting the checkboxes to the left of each parameter value. The normal fit criteria can be viewed in the **Sphere Visualization** window (Figure B.8) which displays the normals of Points, Surfels and Surfaces as points on the unit sphere.

- The user can individually execute the five aggregation phases (**Create Surfaces**, **Refine Surfaces**, **Extend Surfaces**, **Match Edgels**, and **Construct Blocks**) or execute the entire sequence in order with **Reconstruct All**. Each phase is organized into a few subphases which may be disabled by de-selecting the appropriate checkbutton. These phases and subphases are described in detail in Section 3.3.

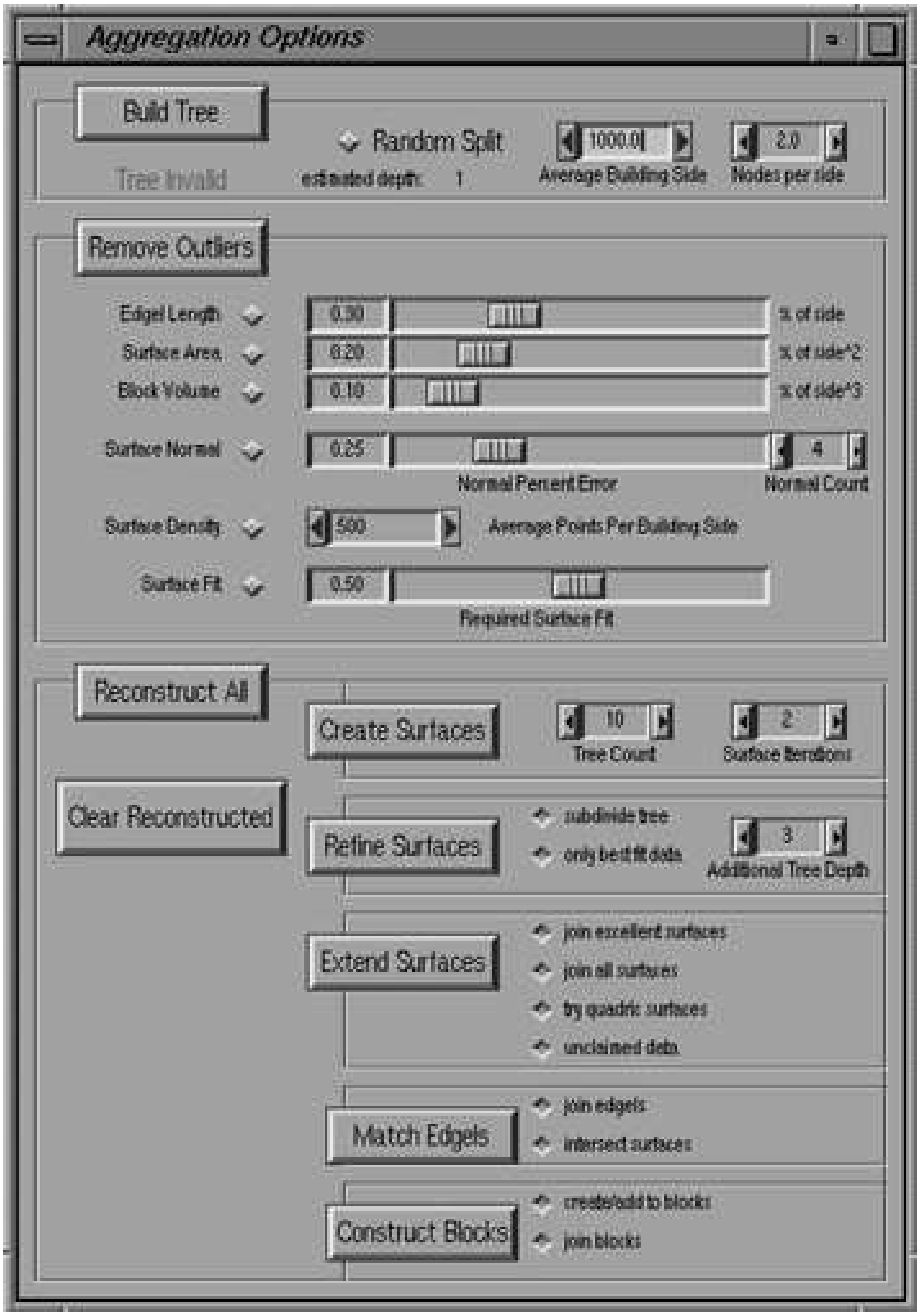


Figure B.7: The **Aggregation Options** window contains the important aggregation parameters and buttons for each phase and subphase of the algorithm.

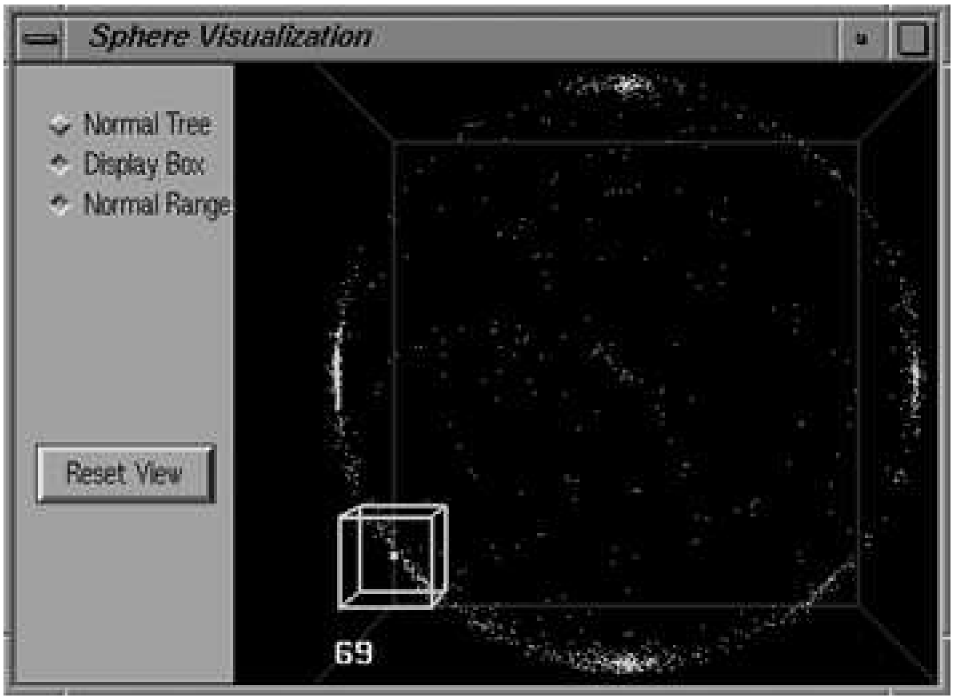


Figure B.8: The **Sphere Visualization** window displays each Object's normal as a point on the unit sphere. The normal of the selected object is shown in yellow, and a wireframe box collects other objects whose normal is approximately the same as the selected object's normal (according to the parameter `surface_normal`). The number in the lower left of the display is a count of these objects.

Appendix C

Command Script

The aggregation program requires a command script as input when run in non-graphical (batch) mode. A command script may also be used to initialize a graphical session (which is described in Appendix B). The script file lists values for the reconstruction parameters, the data files that should be loaded, a region of interest (bounding box), the phases and subphases of aggregation that should be executed, which remove outliers options should be used, and how the results should be saved.

The command script imitates how a user interacts with the graphical version of the aggregation algorithm. Parameters may be set several times and commands may be executed more than once. Parameters may be set and files may be loaded after phases of aggregation have been executed. The most recently set value of each relevant parameter is used when executing a particular command. The program could easily be modified to record (in the command script syntax) the parameters and commands used for a particular graphical aggregation session.

A command script is a sequence of instructions, which each consist of one or more tokens (sequences of non-whitespace characters). They do not need to be on separate lines. The character '`#`' marks a comment; text beyond it on a line is ignored. Figures C.1, C.2, and C.3 show sample command scripts. These are the legal instructions:

parameter `<value>`

Sets parameter to `value`. Appendix A lists the legal and default values for each parameter.

parameter `::=`

```
average_building_side | average_points_per_building_side |
nodes_per_side | random_split | load_sample | data_jitter |
tree_count | surface_iterations | additional_tree_depth |
```

minimum_length | minimum_area | minimum_volume | surface_normal | normal_count | surface_fit

matrix <v0> <v1> ... <v15>

Transforms each object as it is loaded into the system by the matrix M.

$$M = \begin{bmatrix} \texttt{v0} & \texttt{v1} & \texttt{v2} & \texttt{v3} \\ \texttt{v4} & \texttt{v5} & \texttt{v6} & \texttt{v7} \\ \texttt{v8} & \texttt{v9} & \texttt{v10} & \texttt{v11} \\ \texttt{v12} & \texttt{v13} & \texttt{v14} & \texttt{v15} \end{bmatrix}$$

By default, M is the identity matrix.

file <filename>

Loads the file named filename (described in Section 4.5).

bounding_box <v0> <v1> <v2> <v3> <v4> <v5>

Deletes any objects currently loaded that do not at least partially intersect the bounding box (v0, v1, v2) → (v3, v4, v5).

aggregation phase

Executes the aggregation phase (described in Section 3.3).

aggregation phase ::=
create_surfaces | refine_surfaces | extend_surfaces |
match_edgels | construct_blocks

aggregate_all

Executes all of the aggregation phases.

aggregation subphase [0 | 1]

The aggregation subphase is disabled with 0 and enabled with 1. By default, all subphases of aggregation are enabled.

aggregation subphase ::=
refine surfaces subphase | *extend surfaces subphase* |
match edgels subphase | *construct blocks subphase*

refine surfaces subphase ::=
subdivide_tree | use_best_fit_data

extend surfaces subphase ::=
extend_excellent_surfaces | extend_all_surfaces |
try_quadric_surfaces | unclaimed_data

match edgels subphase ::=
intersect_surfaces | join_edgels

construct blocks subphase ::=
create_add_to_blocks | join_blocks

remove outliers option [0 | 1]

The remove outliers option is disabled with 0 and enabled with 1. By default, all remove outliers options (described in Section 3.4) are disabled.

remove outliers option `::=`

`use_minimum_length | use_minimum_area | use_minimum_volume |`
`use_surface_normal | use_surface_density | use_surface_fit`

`remove_outliers`

Performs the enabled remove outliers options.

save command `<filename>`

Saves the current objects (as described in Section 4.5) to the file `filename`.

save command `::=`

`save_all | save_reconstructed | save_final`

```
# set parameter values
average_building_side 950.0
nodes_per_side 2.2
average_points_per_building_side 1500
random_split 1
tree_count 15
surface_iterations 2
additional_tree_depth 6

# load files
load_sample 0.05
file pointcloud.pt
load_sample 1.0
data_jitter 0.1
matrix 1 0 0 500
       0 1 0   0
       0 0 1   0
       0 0 0   1
file surfaces.iv

# delete objects outside bounding box
bounding_box -1000 -1000 -1000 1000 1000 1000

# phases of aggregation
aggregate_all

# save results
save_all output.iv
```

Figure C.1: A sample command script which uses non-default parameter values. Two files are loaded: approximately 5% of the first dataset is loaded, and the second dataset is translated 500 units along the positive x-axis and a small amount of noise is added. All objects outside of the specified bounding box are removed, all the aggregation phases are executed, and all of the data and reconstructed objects are saved.

```
# set parameter values
average_building_side 1000
nodes_per_side 2.0
average_points_per_building_side 500
load_sample 1.0

# load files
file pointcloud.pt

# phases of aggregation
build_tree
create_surfaces
 subdivide_tree 0
 use_best_fit_data 1
refine_surfaces
 extend_excellent_surfaces 1
 extend_all_surfaces 0
 try_quadrics 0
 unclaimed_data 1
extend_surfaces
 intersect_surfaces 1
 join_edgels 0
match_edgels
 create_add_to_blocks 1
 join_blocks 0
construct_blocks

# save results
save_final final.iv
```

Figure C.2: A sample command script which disables some of the aggregation subphases. The reconstructed Blocks are saved to the specified output file.

```
# set parameter values
average_building_side 1000
nodes_per_side 2.0
average_points_per_building_side 500
load_sample 1.0

# load files
file pointcloud.pt

# remove outliers options
use_minimum_length 1
 minimum_length 10
use_minimum_area 1
 minimum_area 1.0
use_minimum_volume 1
 minimum_volume 1.0
use_surface_normal 1
 surface_normal 10.0
 normal_count 5
use_surface_density 1
use_surface_fit 1
 surface_fit 0.1

# phases of aggregation
aggregate_all
remove_outliers

# save results
save_reconstructed reconstruct.iv
```

Figure C.3: A sample command script which enables the remove outliers options, and saves only the reconstructed objects.

Bibliography

[1] Ascender. Knowledge Directed Image Understanding for Site Reconstruction. University of Massachusetts.
`http://vis-www.cs.umass.edu/projects/ascender/ascender.html`

[2] Bala, Kavita, Julie Dorsey, and Seth Teller. Bounded-Error Interactive Ray Tracing. To appear in *ACM Transactions on Graphics,* 1999.

[3] Becker, Shawn and V. Michael Bove, Jr. Semiautomatic 3-D Model Extraction From Uncalibrated 2-D Camera Views. Presented at *SPIE Symposium on Electronic Imaging: Science & Technology*, San Jose, February 5–10, 1995.
`http://www.media.mit.edu/people/sbeck/research.html`

[4] Bentley, J. L. Multidimensional binary search trees used for associative searching. Communications of the ACM, Volume 18,9 pages 509–517, September 1975.

[5] Cao, Xingping, Neelima Shrikhande, Gongzhu Hu. Approximate orthogonal distance regression method for fitting quadric surfaces to range data. *Pattern Recognition Letters*, August 1994, Volume 15, pp 781–796.

[6] Chou, George T. and Seth Teller. Multi-Level 3D Reconstruction with Visibility Constraints. *Proceedings of Image Understanding Workshop*, 1998.

[7] City Scanning Project. Automated System to Produce CAD Models from Images of an Urban Environment. MIT.
`http://graphics.lcs.mit.edu/city/city.html`

[8] Coorg, Satyan. *Pose Imagery and Automated 3-D Modeling of Urban Environments.* PhD. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. September, 1998.
`http://graphics.lcs.mit.edu/~satyan/`

[9] Coorg, Satyan, Neel Master, and Seth Teller. Acquisition of a large pose-mosaic dataset. *Computer Vision and Pattern Recognition.*, 1998, pages 872–878.

[10] Debevec, Paul E., Camillo J. Taylor, and Jitendra Malik. *Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach.* Technical Report UCB//CSD-96-893. Computer Science Division, University of Califonia at Berkeley. January 19, 1996. Façade.
`http://www.cs.berkeley.edu/~debevec/Research/`

[11] De Couto, Douglas S. J. *Instrumentation for Rapidly Acquiring Pose-Imagery.* Master's of Engineering Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1998.

[12] FotoG. Close-range Photogrammetry Software. Vexcel Corporation.
`http://www.vexcel.com/fotog/index.html`

[13] Fua, Pascal. Reconstructing Complex Surfaces from Multiple Stereo Views. *International Conference on Computer Vision.* Cambridge, MA. June 1995. Also available as Tech Note 550, Artificial Intelligence Center, SRI International.
`http://www.ai.sri.com/~fua/`

[14] Heckbert, Paul S. The Mathematics of Quadric Surface Rendering and SOID. New York Institute of Technology. Computer Graphics Laboratory. 3-D Technical Memo No. 4. July 1984.

[15] Hoppe, Hugues; Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface Reconstruction from Unorganized Points. *Computer Graphics* 26, 2, July 1992, pages 71–77.

[16] Lorenen, W. E., and H. E. Cline. Marching cubes: a high resolution 3d surface reconstruction algorithm. *Computer Graphics* 21, July 1987, pages 163–169.

[17] McIvor, Dr Alan M. and Robert J. Valkenburg. A Comparison of Local Surface Geometry Estimation Methods. *Machine Vision and Applications*, Volume 10, Issue 1, 1997.

[18] McIvor, Dr Alan M. and Robert J. Valkenburg. Principal frame and principal quadric estimation. *Image and Vision Computing*, New Zealand, Lower Hutt, August 1996, pp 55–60.

[19] Mellor, J.P., Seth Teller, and Tomas Lozano-Perez. Dense Depth Maps for Epipolar Images. *Proceedings of the 1997 Image Understanding Workshop* and M.I.T. AI Lab Technical Memo 1593.

[20] Mundy, J. and P. Vrobel. The Role of IU Technology in RADIUS Phase II. GE Corporate Research and Development. Schenetady, NY. May 1994.

[21] OpenGL Programming Guide & OpenGL Reference Manual. Addison-Wesley Publishing Co.

[22] Purify. Run-time Error and Memory Leak Detection Program. Rational Software Corporation.
`http://www.rational.com`

[23] RADIUS Home Page. Research and Development for Image Understanding Systems.
`http://www.ai.sri.com/~radius/`

[24] Riegl Laser Measurement Systems. 3D Imaging Sensor & 2D Long Range Laser Scanner.
`http://www.riegl.co.at`

[25] Sceneviewer & Inventor. Three-dimensional Model Viewer and File Format (.iv). Silicon Graphics, Inc.
`http://www.sgi.com/`

[26] Teller, Seth. Automatic Acquisition of Hierarchical, Textured 3D Geometric Models of Urban Environments: Project Plan. *Proceedings of the 1997 Image Understanding Workshop.*

[27] Teller, Seth. Automated Urban Model Acquisition: Project Rationale and Status. *Proceedings of the 1998 Image Understanding Workshop.*

[28] WALKTHRU-PROJECT. Interactive Virtual Building Environments and Simulation-based Design. University of California, Berkeley. `http://www.cs.berkeley.edu/~sequin/WALKTHRU/index.html`

[29] Wrap. Point Cloud Software. Raindrop Geomagic Inc.
`http://geomagic.com`

[30] XForms & `fdesign`. Graphical user interface tools.
`ftp://bloch.phys.uwm.edu/pub/xforms`
`http://bragg.phys.uwm.edu/xforms`

[31] Xiong, Rebecca. *A Stratified Rendering Algorithm for Virtual Walkthroughs of Large Environments.* Master of Science Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. June 1996.
Random City Model Generator for testing CityScape Walkthrough System.
`http://graphics.lcs.mit.edu/~becca/`