024

TR-H-280

# Point Cloud Mesh Generation and Rendering.

Jean-Pierre DOYON

1999.12.20

ATR人間情報通信研究所
〒619-0288　京都府相楽郡精華町光台2-2　TEL: 0774-95-1011

ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan
Telephone: +81-774-95-1011
Fax　　　: +81-774-95-1008

# Point Cloud Mesh Generation and Rendering

Jean-Pierre Doyon
e-mail: jipi@iname.com
ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan
Telephone: +81-774-95-1011
Fax: +81-774-95-1008

December 17, 1999

**Abstract**

The general goal of this project is to develop a system able to reconstruct the 3D structure of an object, with only minimal knowledge about the camera, the object, and the relative position of the two; namely, the object is textured and rigid and assuming a perspective camera model. This system is able to acquire images from an uncalibrated, general purpose camera, extract feature points in the acquired images and track those points in the stream of images. Those points that have been tracked for a long period of time are then used to reconstruct the 3D structure of the object in the form of a 3D point cloud. The images, the tracked points for each image and the 3D reconstructed points are then used to create a realistic rendering of the object. This report will focus on the last part of the project, namely the realistic rendering of the object. It relates all the steps taken to achieve the final goal, including numerous wrong paths. The reasons for taking these paths as well as the explanations of why these were wrong are also given.

## 1 Introduction: Understanding the problem

Visualization of data is an important aspect of today's research. Of course, fundamental research yields mostly mathematical formulas, which are pretty hard to explain to non-scientific people. But more applied research, such as the subject of this report, can be better exposed through the use of graphical representation. Therefore, a good representation can make the difference between a successful presentation and a partial failure. Although the most interesting part of research is most likely not working on some interface for displaying results, but how to generate those results, the former is still very important. This project has a little of both. Given the reconstruction of a 3D structure, to create an appealing and intuitive representation for visualizing the data. And also, if possible, improve the quality of the display so it matches as much as possible the actual object.

As will be discussed in greater detail later in this document, obtaining a visually appealing interface for visualizing the data is quite simple and straightforward to implement. But the enhancement of this display, so that the displayed object matches as closely as possible the original object was a little more problematic. Although many ideas were found to solve this problem, most of them led to bad results and had to be discarded.

### 1.1 What was already done

The 3D reconstruction process of an object, from an uncalibrated camera, as explained and designed in [1], generates 3 types of data. It generates a set of $n$ images, from different views of the object, a set of $n$ 2D points files, containing coordinates of the points tracked from on image to the next, and used to generate the 3D reconstruction and one 3D point file, containing the reconstructed coordinates in 3D space of the tracked 2D points. Every point file (2D and 3D) has the same number of points, and point correspondence across files is determined by the order of the points. So this is the starting point for the generation of a rendered version of the data.

### 1.2 The goal of the project

The main goal of this project was to develop a program that would allow the reconstruction/display of any rigid, textured 3D object. 3D objects can be classified in many ways: convex/concave, polyhedral/non-polyhedral, rigid/non-rigid, textured/flat, etc. A convex object can be defined as follows: any line segment from any point of the object to any point of the object is also part of the object. This is also defined as a convex hull. A concave object can be best defined from the definition of a convex object, that is: a concave object is not convex. A polyhedral object is an object which can be entirely defined using planar surfaces; it doesn't have any curved surfaces. A non-polyhedral object has curved surfaces. A textured object is defined as an object having a non-uniform colored surface, as for a flat object would be of uniform color. A rigid object refers to an object which doesn't change it's 3D structure over time (is not deformed), and a non-rigid object can be deformed over time.

At the beginning of this project, the aim was to be able to deal with any kind of rigid, textured 3D object. But in the end, due to unsuccessful results, the type of object was further restricted to polyhedral, convex, rigid, textured 3D object.

### 1.3 Rendering: using existing software

Once we have the point files (2D and 3D), the correspondence information (ordering of the points in the files) and the images, the next step is displaying the rendered object. To simplify this task, an already existing software was used for rendering. Geomview, a 3D viewer developed by the Geometry Center[4] seemed, at first, to be a very good choice. The file format used to represent the 3D world was very simple and very easy to generate. But a serious problem quickly appeared, the rendering was unstable, especially when part of the object was outside of the viewing window, the texture from two adjacent triangles would no longer fit and became distorted. Two alternatives were available, either use another rendering program, or build one from scratch. The alternative chosen was to use VRML[1], which is still quite simple to generate and has quite a few viewers available on the Internet.

## 2 Generating a triangular mesh on the input points

The files generated by the preceding stage (tracking & 3D reconstruction) contains two types of information for the same points, 3D reconstruction and their 2D projection. Simple point clouds like these are very unattractive for displaying the structure of an object. The aim was to obtain a more intuitive display of the rendered object, using the textures from one of the images acquired in the previous step. But in order to map the textures from the image on the 3D reconstruction, more than 3D points are required. For that reason, a new 3D representation must be generated. The kind of representation usually used in texture mapping applications is the triangular mesh. Such a representation is very easy to generate, and very well understood and also very well documented. The software used to generate the tessellation was once again taken from the Geometry Center[4][5].

When the tessellation of a 3D point cloud is required, the point cloud is first projected to a plane and the tessellation is computed on this 2D projection instead of the 3D point cloud. The reason for this is that generating a tessellated convex hull in 3D is far more complex than in 2D and generally does not give good results. Since the input to this step is composed of corresponding 3D and 2D point clouds, the tessellation was made directly on the 2D data, and then transferred

[1]Virtual Reality Mark-up Language

to the 3D points. That is, the same point order was used on both the 2D and 3D data. Figure 1 is an example of an object's image and the 2D points used for reconstruction and tessellation. Figure 2 is an example of tessellation.

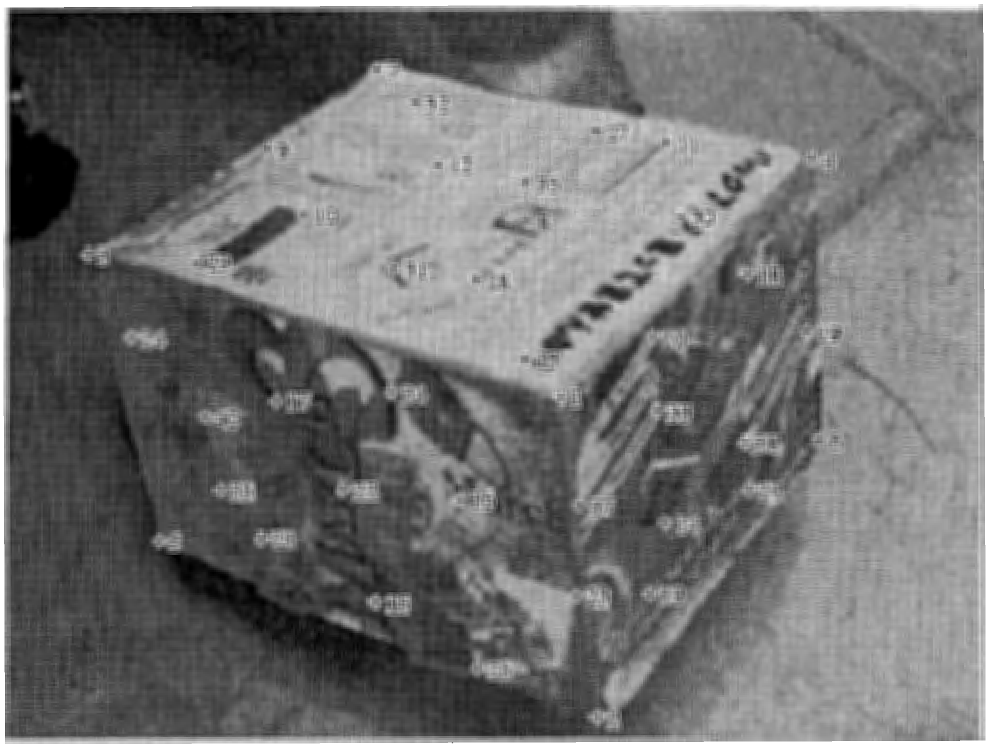

Figure 1: Image example, with 2D points.

## 3 Visualizing the textured 3D object

Generating a preliminary, crude output is very easy and straightforward, given the input file available. VRML output can be generated very easily; only two steps are required to obtain the data in a format suitable for VRML output. First step, generating the triangular mesh, as described in Section 2. Second step, normalizing the 2D data points to be texture coordinates. This is simply done by dividing the coordinates of the 2D points by the size of the images. The VRML output can then be easily viewed in most Web browsers equipped with the appropriate plug-in (see for example, [6]).

But as can be seen in Figure 3 and especially in Figure 4, the output is distorted and does not really look like the original box (Figure 1). The main reason for these results arise from the way the tessellation is computed. It is computed on the 2D points, which contain no information on the 3D structure of the object. Therefore, 2D triangles whose image contain parts of two surfaces of the object can appear in the tessellation. This problem, coupled to the fact that the number of tracked points is rather small leads to the generation of big triangles, that tend not to respect the structure of the object when transferred to the 3D reconstruction. In our example (Figures 1 to 4), there are a few triangles whose texture is going through the edges of the box and as such, distort[2] the reconstruction. For example, the triangles composed of the points number 34-31-28 or 20-18-30 are such triangles. Those triangles will be referred later in this document as *bad* triangles,

[2]It should be noted that the other distortion of the box, namely the stretching, is caused by another problem, present in the reconstruction of the 3D structure. For more details on this, please refer to [1].

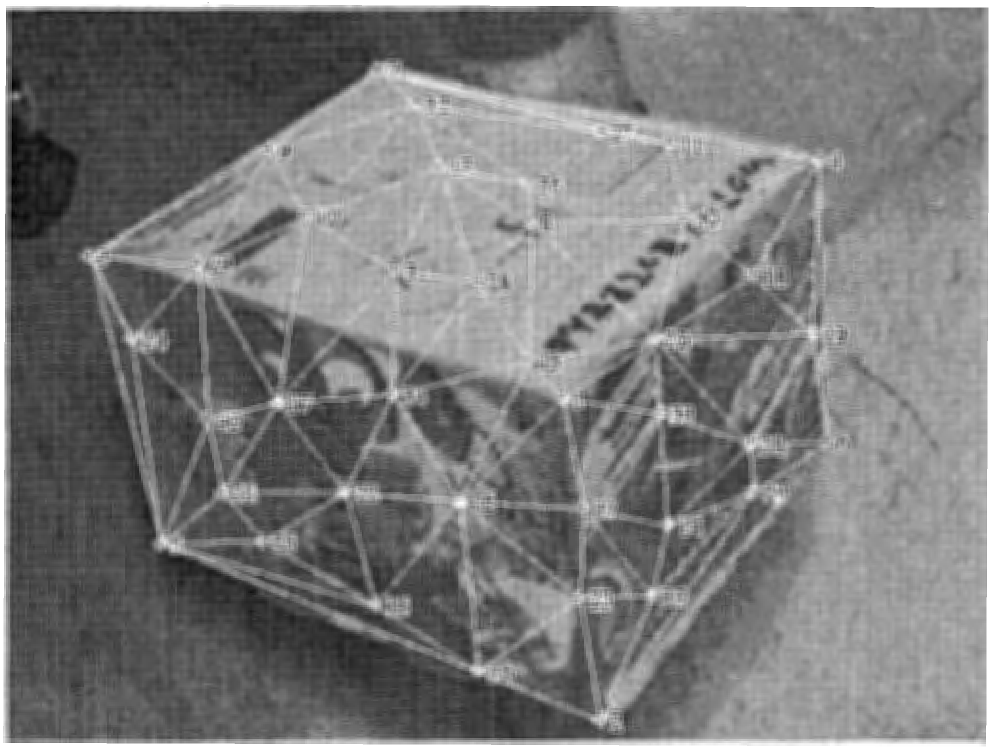

Figure 2: A tessellation (triangulation).

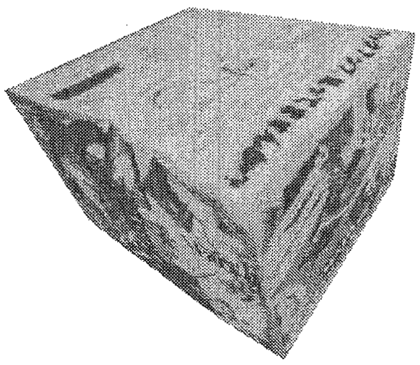

Figure 3: The rendered object.

while the triangles such as the one composed of the points 22-16-34 or 18-19-20 which contain only the image of a 3D planar surface will be referred as *good* triangles.

## 4 Improving the quality of the display

Simply displaying the rendered object has already improved a lot the visualization of the results. But there is still room for improvement. Simply eliminating the triangles that are clearly wrong (triangles that are crossing through an edge of the box) would already be a great improvement. Replacing those triangles with *correctly* generated triangles would be even better.

To achieve such a goal, we tried a few different techniques, mostly based on the texture content of each triangle. From one image to the next, the viewpoint of the camera, relative to the object, changes. Therefore, any triangle representing a planar surface (so-called *good* triangles)

Figure 4: Another view of the same object.

will contain strictly the same information from one view to another; although it will be sheared or rotated. This derives directly from an invariant found in projective geometry which states that the *cross-ratio* of distances is invariant to change in view points, or more precisely, that the *cross-ratio* of four distinct points on a line is invariant under perspective projection. Let $A$, $B$, $C$, $D$ be four points on a straight line $u$. Let them be projected into points $a$, $b$, $c$, $d$ upon another straight line $u_1$. Then the *cross-ratio* of the points on $u_1$ is equal to the *cross-ratio* of the points on $u$ for any projection.

$$\frac{(C-A)\cdot(B-D)}{(B-A)\cdot(C-D)} = \frac{(c-a)\cdot(b-d)}{(b-a)\cdot(c-d)} \tag{1}$$

For example, see Figure 5. But if a triangle is not representing a planar surface (i.e. crossing between two surfaces of the object), then that cross-ratio is no longer invariant and part of the triangle will be compressed or stretched, as can be seen in Figure 6. The first few techniques used to eliminate those *bad* triangles all try to find this skew in the triangle texture between many triangles (see Section 5).

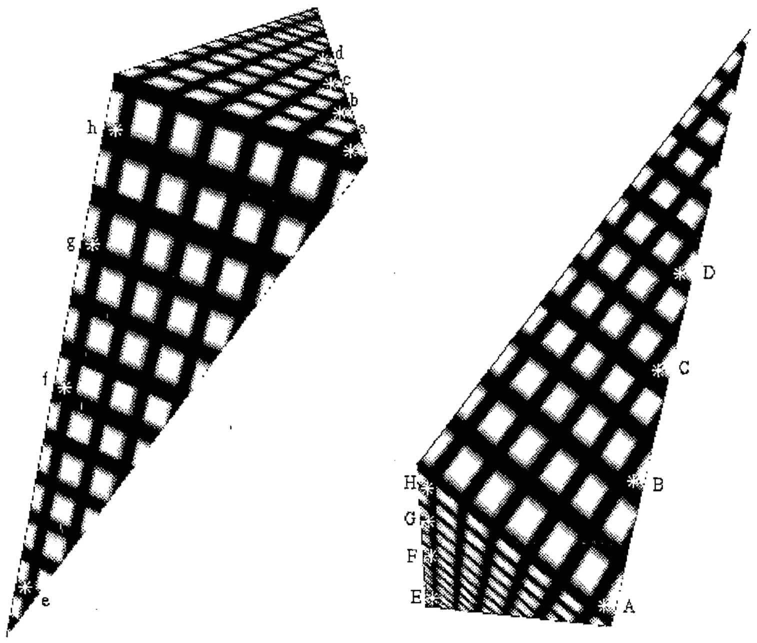

Figure 5: The same patch, corrected to have two triangles. Notice the points remain on the same line from the left image to the right image.

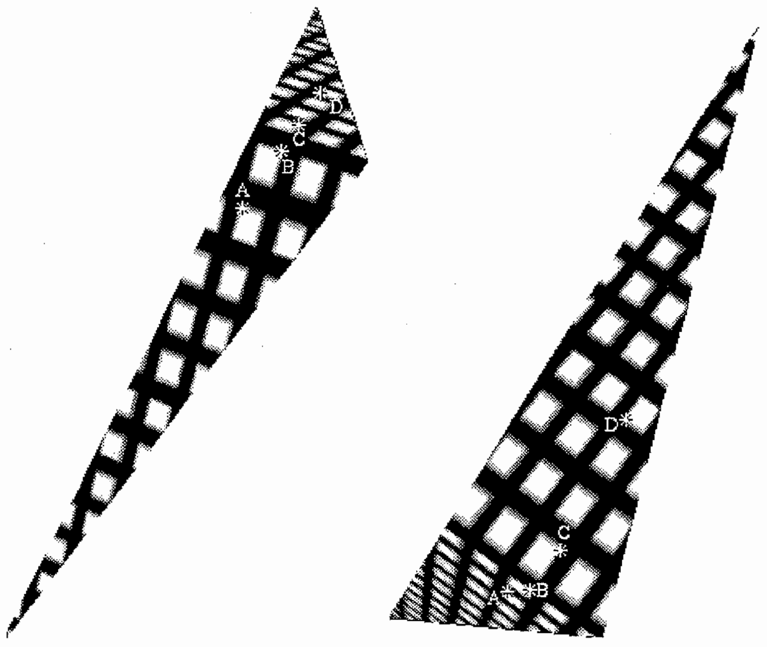

Figure 6: Example of distortion observed in a triangle, from two different views, 4 points on a line in the left image are no longer on the same line in the right image.

Once those *bad* triangles are eliminated, or at least identified, they have to be replaced by *better* triangles. In this case, a *better* triangle would be defined as a triangle that matches more closely the true surface of the object.

## 5 Comparing triangles

Since there is a substantial difference between two *bad* triangles' texture, when viewed from a different angle; the idea of comparing the content of each triangle from one view to another therefore makes sense. The only thing needed is an estimator which will quantify the difference between two images. In the absence of image noise, the difference between two views of a *good* triangle should be very small, and caused mostly by the digitizing error. On the other hand, when comparing two *bad* triangles (two triangles going through an edge or not representing a real planar surface), the error should be much higher. Since it is assumed that the digitizing error (pixel noise) is lower than 1 pixel, it should not make a big difference in the comparison of the triangles.

In order to compare the two views of the same triangle, it is required to be able to find corresponding pixel positions in two different views of the same triangle. To achieve this, the homography. Relating the two triangles is computed. A homography $H$ relates 2 triangles if and only if these triangles are projections of the same triangle in space.

### 5.1 Using the root mean squared error of the pixels

A very intuitive and easy estimator to use to quantify the difference between two sets of data is the Root Mean Square of errors $RMS_{error} = \sqrt{\sum \left( err^2_{pixel_{i,j}} \right)}$ where $err_{pixel_{i,j}} = Img1_{i,j} - Img2_{i,j}$ is the difference between the pixels from each triangle texture. The main advantages of this approach are the low computational cost and simplicity.

## 5.2 Why the RMS error isn't a good estimator

There are important drawbacks to the RMS error estimator. First, this technique requires a threshold to differentiate between good and bad triangles. This threshold doesn't have any *physical* meaning and therefore has to be set empirically. Second drawback, this technique is not robust to changes in brightness and contrast (offset and scaling of the gray value) that may occur in the data. This last drawback is a major one, since the input images do not come from a controlled environment, but instead, from bad or changing lighting conditions. These conditions result in a great variation in the RMS error, making it impossible to differentiate between *good* and *bad* triangles as can be observed in Figure 7.

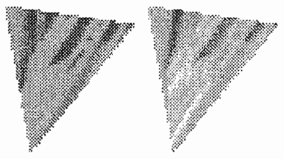

Figure 7: Contrast and brightness change for two views of the same triangle. In this case, the left triangle has a mean pixel value of 71 (the maximum being 255) and a standard deviation of 77. The right triangle has a mean pixel value of 91 and a standard deviation of 99.

## 5.3 The variance of errors, a more robust estimator

Using a very similar approach to the RMS error, it is possible to become robust to changes in brightness very easily. The idea is that if the two triangles have a different content (because they are distorted) then the pixel error will vary a lot. But if the only difference between the two triangles is in brightness, then the error will be approximately constant. So all one needs to verify is the variance of errors, described by the following equation:

$$Var_{error} = \frac{\sum \left( err^2_{pixel_{i,j}} \right) - n \cdot \overline{err_{pixel}}^2}{n-1} \tag{2}$$

Where $n$ is the number of pixels and $\overline{err_{pixel}}$ is the average gray-value error. This is another very simple way to differentiate the *bad* triangles from the *good* ones. Unfortunately, this estimator is not robust to contrast changes. It is also not more robust to pixel error than the RMS error estimator; and still requires the use of a threshold, which must be set empirically.

## 5.4 Increasing robustness in the presence of point noise, the neighbor best-search

Even using the variance of the errors, the difference between *good* and *bad* triangles is still not clear. The fact that this estimator, $Var_{error}$ is not robust to changes in contrast might be to blame, but another reason starts to emerge. The tracking algorithm seems to be making a bigger error than previously assumed. That results in a pixel error closer to 1 pixel instead than much lower than one pixel. In order to gain in robustness to pixel noise, a simple best search can be performed on the immediate neighbors of the *target* pixel. The 3x3 neighborhood of each pixel of the second triangle is being searched to find the one that minimizes the error. In the hope to compensate for any error of 1 pixel.

$$pixel_{err_{i,j}} = \min \left( p1_{i,j} - \begin{bmatrix} p2_{i-1,j+1} & p2_{i,j+1} & p2_{i+1,j+1} \\ p2_{i-1,j} & p2_{i,j} & p2_{i+1,j} \\ p2_{i-1,j-1} & p2_{i,j-1} & p2_{i+1,j-1} \end{bmatrix} \right) \tag{3}$$

Although the results were improved by the best-search algorithm, it was still impossible to make a clear difference between *good* and *bad* triangles.

## 5.5 Re-estimating the triangle points to reduce the pixel error

So to verify the error on the position of the points of the triangle was low, another technique was tried. It consisted in testing the whole neighborhood of each of the three triangle points with the whole neighborhood of each other of three triangle points, resulting in 729 (*NeighborhoodSize*$^3$ where *NeighborhoodSize* = *3 x 3)* comparisons of the texture content of the whole triangle (Figure 8). The best position was then used as the position of the point. The results of the neighborhood search revealed that it was indeed required, since the best position (which minimized the error) was the original one only in about 10% of the cases. But because of the shear volume of data that was required to analyze in order to obtain those results, this technique could hardly be used.

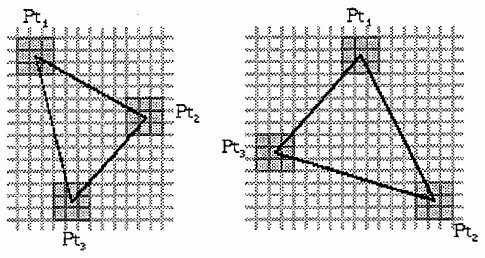

Figure 8: Re-estimation of the triangles' points using Neighborhood best-search between 2 corresponding triangles.

## 5.6 Equalizing the image using histograms

In order to increase the robustness to contrast changes, the image was equalized using the histogram equalization method. The technique is usually used to enhance the contrast of an image, but it has a very nice side-effect. It makes the distribution of pixels per gray-level flat. So if the texture content of the triangle should be the same, equalizing it with this technique will then remove the contrast difference between the two images. Another side-effect of this technique is that it re-centers the histogram, effectively eliminating the offset (or brightness difference).

This technique uses bins to do its processing. The number of bins is usually the total number of values a pixel can have (256 in this case) and the value for each bin of the histogram is the number of pixels having that gray-level value in the image. Each bin value is the number of pixel having that gray-level in the image. The histogram generated this way can be viewed as an estimate of the probability of occurrence of each gray-level. So what we want to do is to equalize this probability, which can be done by applying the equation

$$s_k = \sum_{j=0}^{k} \frac{n_j}{n} \tag{4}$$

to the histogram and re-mapping the values for all pixels in the image to these new values; where $s_k$ is the new value for the $k^{th}$ gray-level, $n_j$ is the number of pixels having the gray-level $j$ and $n$ is the total number of pixels. Essentially, what we are doing is taking the bins and equalizing the probability that a particular bin will fall at a particular gray-level (Figure 10).

## 5.7 Equalizing the triangles using histograms

The histogram equalization, used for the whole image did not, as expected, have any good effect on the results. The reason for that being that the differences in brightness and in contrast happen on

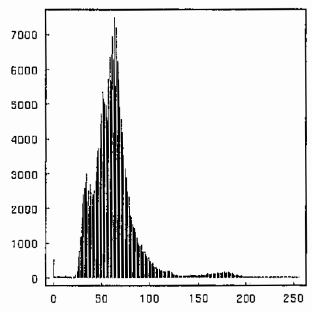

Figure 9: Histogram of an image before equalization.

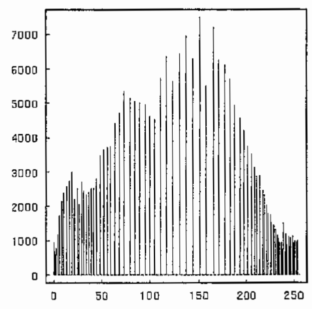

Figure 10: Same Histogram after equalization.

the different faces of the object. So equalizing the whole image will only make it worst, especially since the background, which is not part of the object, will still be used for equalization. Where this algorithm could be better used though is in equalizing the triangles' texture content before comparing two triangles.

As expected, this approach successfully reduced the error for *good* triangles. But it was still impossible to determine a threshold to differentiate the *good* and *bad* triangles.

## 5.8 Using correlation to compare triangles

Since the RMS error and the variance of errors can not be used successfully to separate *good* and *bad* triangles, regardless of any improvements, another basic method seems to be required. So we tried to use the correlation of corresponding triangles' texture from different view as an estimator for the error. The correlation of images can be represented by the equation:

$$x_{corr} = \frac{\sum_{i=1}^{n} ((u_i - \overline{u}) \cdot (v_i - \overline{v}))}{(n-1) \cdot \sqrt{\sigma_u^2} \cdot \sqrt{\sigma_v^2}} \tag{5}$$

Where $u_i$ and $v_i$ represent the $i^{th}$ pixel of the texture inside the triangles to compare, $\overline{u}$ and $\overline{v}$ are the mean pixel value of both triangles' texture, $\sigma_u$ and $\sigma_v$ are the standard deviation of the pixels' values.

## 5.9 Why correlation is no better than the RMS error or the variance of errors

Perhaps a little late in the project, the reason for the repeated failure of the different approaches became more obvious: all the methods were pixel-based. That is, the noise in the triangles' points, coupled to the error caused by interpolating to find the proper pixel value, was simply making this approach inappropriate. We needed to use another approach if we were going to use the content of the image. The pixels themselves do not hold any information on the structure of the image. That is, if there is a stretch in the image, then there might still be enough pixels having a high correlation score, or constant error to generate bad results. A good way to cope with these problems could be to use a method for extracting and matching features inside the triangle.

### 5.10 Extracting and matching features within the triangles

The logic behind this idea is that, using very simple geometry, it is possible to define normalized two-dimensional coordinates systems based on two views of the same triangle. Therefore, extracting and matching features from the two triangles would be very easy. In the presence of *good* triangles, these features would be at exactly the same coordinates in the normalized spaces. But if the triangle was a *bad* one, this positions would be different. But due to a lack of time, this algorithm was not tested nor implemented.

## 6 Adding some knowledge about the object, restricting to polyhedral objects

Since the previous pixel-based methods were unsuccessful, a different approach was used. We made the hypothesis that the object was also convex and polyhedral; i.e. it is assumed the object is formed only of planar surfaces and does not contain any hole. Therefore, the objective was to regroup the triangles into planes. Those planes can then be extended to their intersections so the *bad* triangles are replaced by *good* triangles.

### 6.1 Finding planes in the object

There are a few ways to extract planar information from a 3D tessellated point cloud. We could have used a mesh decimation algorithm, which would have *grown* planar surfaces[8][9]. It iteratively merges the triangles that are the most planar to form bigger planar surface. The problem with such and algorithm is that it does not identify *bad* triangles but simply merge good ones. An other way would be to regroup the triangles roughly on the same plane together to define only a few planes, and eliminate the triangles that do not fit one of those planes. This is the approach that was used, and that will be described in the next sub-sections.

#### 6.1.1 Representing the triangles for clustering

A very simple way to define the orientation of a plane in 3D is through it's normal. In order to classify the normals, to determine the planar surfaces of the polyhedral object, the normals were represented in spherical coordinates (Figure 11)$\vec{n} = \begin{pmatrix} \theta & \rho & \phi \end{pmatrix}^T$. Since the normals were normalized $\|\vec{n}\| = \rho = 1$, the norm $\rho$ coordinate was not required and the normals could then be represented in 2D space; using the zenith and the azimuth $\begin{pmatrix} \phi & \theta \end{pmatrix}$. These coordinates where then used for clustering.

#### 6.1.2 Clustering the normals

The clustering algorithm used was the Minimal Spanning Tree. As described in [2], it works in the following way: considering all the objects (normals in this case) as clusters, find the two clusters having a minimal distance and join them in a new cluster, positioned at the center of mass of the two merged clusters (Figure 12). Let $X_i$ and $X_j$ be two separate clusters, then:

$$d_{mean}(X_i, X_j) = \|\overline{X_i} - \overline{X_j}\| \tag{6}$$

This is repeated until only one cluster remains. In this case, since we don't want to obtain a complete Minimal Spanning Tree, but a certain number of clusters, the iteration is stopped earlier, when the distance between the two closest clusters is greater than a certain threshold, in this case, 10°. Also, the clusters containing less than $x$ normals are discarded, where $x$ is set to 3 in this implementation.

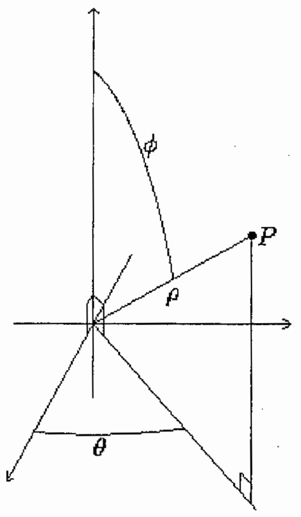

Figure 11: Spherical coordinates.

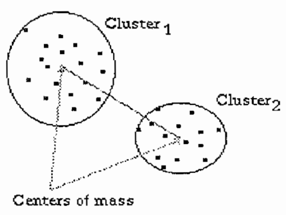

Figure 12: Clustering using the center of mass of the clusters.

### 6.1.3 Other clustering philosophy

There are other ways to generate clusters using the normals from the triangles. Although the general algorithm remains the same as stated above, the measure used to determine the distance between clusters can be changed. For instance, instead of using the center of mass of the cluster, one could use the closest sub-cluster distance (Figure 13):

$$d_{min}\left(X_i, X_j\right) = \min_{x \in X_i, y \in X_j} \|x - y\| \tag{7}$$

One could also use the farthest sub-cluster distance (Figure 14):

$$d_{max}\left(X_i, X_j\right) = \max_{x \in X_i, y \in X_j} \|x - y\| \tag{8}$$

The problem with both of these measures is that like all techniques involving minima or maxima, they tend to be very sensitive to *outliers*. The use of the center of mass is an easy way to gain robustness. Especially since in our case we know there will be outliers, since we know there is more than 1 cluster and probably *bad* triangles. These two other measures are therefore obvious bad choices.

### 6.1.4 Computing best-fit planes

Once the triangles' normals have been clustered in different planes, in order to reduce the noise in the 3D rendered image, all the points are projected on their respective planes. But, in order to achieve this, the plane that best-fit all the points of all the triangles of each cluster must be

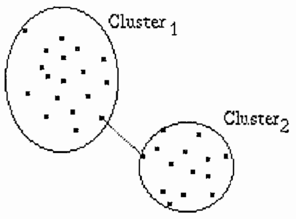

Figure 13: Clustering using the minimum distance between clusters.

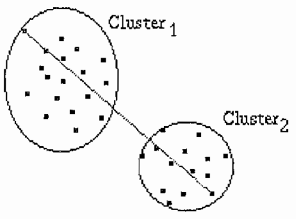

Figure 14: Clustering using the maximum distance between clusters.

computed. In order to compute the best-fit plane for each cluster, orthogonal regression was used. Orthogonal regression computes the best-fit plane, by minimizing the distance of each point to the plane. The source code for computing the orthogonal regression was taken from the Internet at [10].

### 6.1.5 Projecting the 3D points on their respective planes

Once the best-fit planes are computed for each cluster, each 3D point is then projected on their respective clusters. Points that are on more than one plane are projected to the intersection of the different planes they are part of (line or a single point).

### 6.1.6 Interpolating points on the intersections of the planes

Once the different planes have been computed and the points projected, the *bad* triangles must be replaced. Usually, each *bad* triangle has points on more than one plane. In order to replace the triangles, new points are created on the intersection of the planes. Since the point being projected on the intersection already exists on one of the two planes, it's projection is included in the other plane (see Figure 15).

### 6.1.7 Obtaining the 2D description for interpolated points

Once the new 3D points have been interpolated, their 2D projection on their new respective planes is computed. Then, the homography[3] between the 2D descriptions of the each of the 3D plane and the corresponding original 2D image points is computed. This approach is required in order to obtain the relation between the newly 3D created points and the 2D images. It also allows the program to function even if the projection matrix relating each image to the 3D points is unknown. Each newly created point is then projected using this new projection matrix, to obtain their positions in the original 2D image. Each plane's 2D points are then re-triangulated as described in Section 2. This last step is required to minimize the remaining holes in the final rendering (see Figure 16).

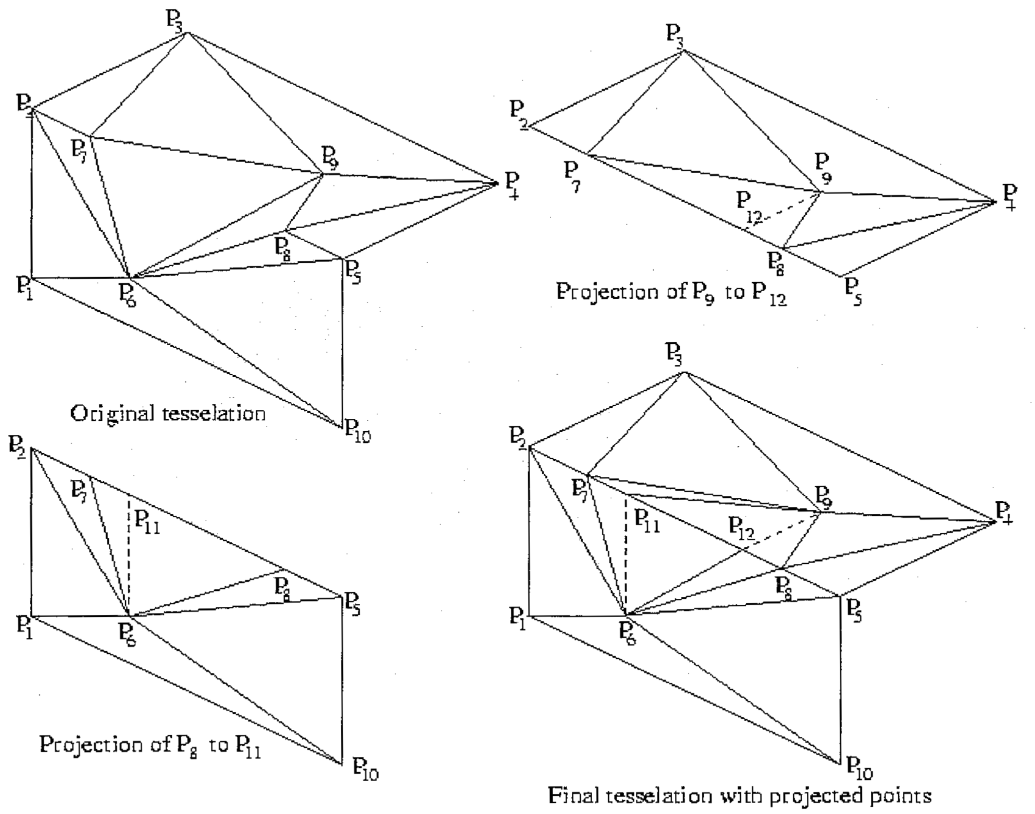
Original tesselation
Projection of $P_9$ to $P_{12}$
Projection of $P_8$ to $P_{11}$
Final tesselation with projected points
$P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$ $P_8$ $P_9$ $P_{10}$ $P_{11}$ $P_{12}$

Figure 15: Projection of points from a plane to another.

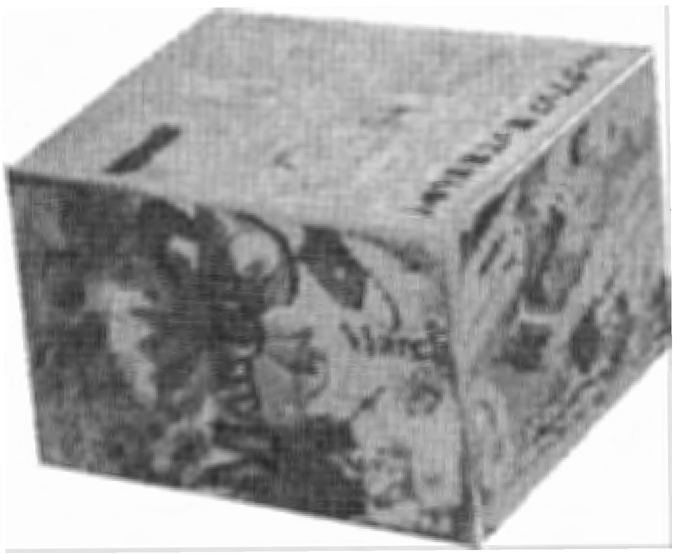

Figure 16: The final rendering.

## 7 Conclusion

In this report, we have presented a few approaches to generate the rendering of a reconstructed three-dimensional point cloud which represents a convex-rigid polyhedral object. These approaches were based on both the content of the texture of the images from which the three-dimensional point cloud was reconstructed and the structure of the triangulated point clouds. We also give an explanation why the pixel-based approaches were unsuccessful in this endeavor.

This implementation has the advantage that it does not require any additional information other than the three-dimensional and two-dimensional point clouds and the image from which the two-dimensional point cloud was extracted from. The use of a homography to relate the planar surfaces of the three-dimensional object to the camera image removes the need to know the projection matrix. On the other hand, we fell short of our goal, which was to be able to deal with any rigid-textured object, since this implementation is restricted to convex, polyhedral, rigid textured objects.

Further work could be done in order to lift the restriction that the object must be polyhedral, by using the texture information as outlined in Section 5.10, and to enable the processing of objects with occlusion, by processing different parts of the object separately and merging the surface information.

**Acknowledgments** I would like to thank my department head, Shigeru Akamatsu, who kindly gave me this formidable opportunity to work in his department at ATR. I would also like to thank all the people from the Co-op Japan Program for putting together such a great program, which allowed me to obtain this work in Japan.

## References

[1] M. Tonko and K. Kinoshita, On the Integration of Point Feature Acquisition, Tracking and Uncalibrated Metric Reconstruction. HIP lab. ATR, Internal Report # TR-H-261, January 1999.

[2] R. Duda and P. Hart, Pattern Classification and Scene Analysis. John Wiley & Sons, 1973.

[3] K. Kanatani and N. Ohta, Accuracy bounds and optimal computation of homography for image mosaicing applications. Proceedings of the 7th International Conference on Computer Vision, 20-27 September 1999, Kerkyra, Greece.

[4] http://www.geom.umn.edu/

[5] C.B. Barber, D.P. Dobkin, H. Huhdanpaa, The Quickhull Algorithm for Convex Hulls, TMS(22), No. 4, December 1996, pp. 469-483.

[6] http://www.intervista.com/

[7] C. Coelho, A. Heller, J. L. Mundy, D. A. Forsyth, A. Zisserman, An Experimental Evaluation of Projective Invariants, Geometric invariance in computer vision, 1992, pp. 87-104.

[8] W. Schroeder, Polygon Reduction Technique, 1995. IEEE Visualization '95. Advanced Techniques for Scientific Visualization, Section 1.

[9] W. Schroeder, J Zarge, W. Lorensen. Decimation of Triangle Meshes. In Edwin E.Catmull, editor, Computer Graphics (*SIG-GRAPH '92 Proceedings*), volume 26 pp. 65-70, July 1992.

[10] http://www.magic-software.com/