Abstract

FOAM, Found Object Assemblage Mosaic, is a new kind of mosaic where three dimensional objects are used to compose a given image/photo instead of the conventional notion of tiles. The end result is a three dimensional landscape of objects joined, stacked, and essentially piled together to compose the original image, while at the same time preserving its feature edges, level of detail, and perceivable colors. Unlike other kinds of mosaic, FOAM brings a three dimensional component to the mosaic, giving a relief-like effect to the resultant image.

Venturing into a three dimensional representation of mosaics, FOAM presents a novel method to circumvent the tile packing problem faced by conventional mosaic algorithms. FOAM also addresses the implicit level of detail problem in other mosaic algorithms when the size of the image tiles is not sufficiently small to represent the smaller details of the given image.

Subject Descriptors:

I.4.0  [Image Processing and Computer Vision] General
I.4.3  [Image Processing and Computer Vision] Enhancement
I.4.6  [Image Processing and Computer Vision] Segmentation
J.5    [Arts and Humanities] Fine Arts

Keywords:

Mosaic, Non-Photorealistic Rendering, Image processing and enhancement, Image segmentation
FOAM: Found Object Assemblage Mosaic

By

Jesse Prabawa Gozali

Department of Computer Science
School of Computing
National University of Singapore
2005/2006
FOAM: Found Object Assemblage Mosaic

By

Jesse Prabawa Gozali

Department of Computer Science
School of Computing
National University of Singapore
2005/2006

Project No: U066030
Advisor: A/P Tan Tiow Seng

Deliverables:
Report : 1 Volume
Source Code : 1 CD-ROM
Abstract

FOAM, Found Object Assemblage Mosaic, is a new kind of mosaic where three dimensional objects are used to compose a given image/photo instead of the conventional notion of tiles. The end result is a three dimensional landscape of objects joined, stacked, and essentially piled together to compose the original image, while at the same time preserving its feature edges, level of detail, and perceivable colors. Unlike other kinds of mosaic, FOAM brings a three dimensional component to the mosaic, giving a relief-like effect to the resultant image.

Venturing into a three dimensional representation of mosaics, FOAM presents a novel method to circumvent the tile packing problem faced by conventional mosaic algorithms. FOAM also addresses the implicit level of detail problem in other mosaic algorithms when the size of the image tiles is not sufficiently small to represent the smaller details of the given image.

Subject Descriptors:

I.4.0 [Image Processing and Computer Vision] General
I.4.3 [Image Processing and Computer Vision] Enhancement
I.4.6 [Image Processing and Computer Vision] Segmentation
J.5 [Arts and Humanities] Fine Arts

Keywords:

Mosaic, Non-Photorealistic Rendering, Image processing and enhancement, Image segmentation

Implementation Software and Hardware:

Intel Pentium 4, NVIDIA GeForce 6600, MS-Windows, C++, OpenGL, GLUT, Advanced Visualization Lab GLM, NVIDIA Cg
Acknowledgements

First of all, I would like to extend my sincere gratitude to my supervisor, A/P Tan Tiow Seng, for his constant support and guidance throughout the project. I am grateful for the opportunity and privilege to be able to work on this Undergraduate Research Opportunity Project under his care.

I would like to thank my brother, Johan Prawira Gozali, for his kind advice and expertise in addressing my numerous queries. He has been a great inspiration for the completion of the project. I would also like to thank my closest dear friend, Sng Ai Lin, for her constant moral support and listening ear.

Finally, I would like to give my utmost gratitude for God Almighty for His countless blessings throughout the many milestones in the project. Without Him, the project and its entirety would not have been possible.
# Table of Contents

Title ................................................................................................................................. i  
Abstract .......................................................................................................................... ii 
Acknowledgement .......................................................................................................... iii

1. Introduction .................................................................................................................. 1  
   1.1. Mosaic .................................................................................................................. 1  
   1.2. Found Object Assemblage .................................................................................... 2  
   1.3. FOAM: Found Object Assemblage Mosaic .......................................................... 2

2. Related Work ............................................................................................................... 4  
   2.1. Tile Packing ......................................................................................................... 4  
   2.2. Database Searching ............................................................................................. 5  
   2.3. Level of Detail ...................................................................................................... 6  
   2.4. Mosaic on the GPU .............................................................................................. 6

3. FOAM Formalization ................................................................................................. 7  
   3.1. Problem Definition .............................................................................................. 7  
   3.2. Algorithm Description ......................................................................................... 7

4. Implementation Outline ............................................................................................. 8  
   4.1. NVIDIA Image Processing Framework .................................................................... 8  
   4.2. Framework Modifications .................................................................................... 9  
   4.3. FOAM as A Filter Graph ...................................................................................... 11  
      4.3.1. Input Preparation .......................................................................................... 12  
      4.3.2. Edge Detection ............................................................................................ 14  
      4.3.3. Image Segmentation .................................................................................... 17  
      4.3.4. Vermis Lines Creation .................................................................................. 29  
      4.3.5. Mosaic Composition ..................................................................................... 30  
   4.4. Implementation Caveats ....................................................................................... 36  
      4.4.1. AVL GLM Library Constraints ...................................................................... 37  
      4.4.2. Resource Consumption ................................................................................ 37  
      4.4.3. Database Creation ....................................................................................... 37

5. Results ....................................................................................................................... 38

6. Conclusion .................................................................................................................. 41  
   6.1. Summary .............................................................................................................. 41  
   6.2. Advantages of FOAM ......................................................................................... 41  
   6.3. Recommendations for Future Work ...................................................................... 42

References ..................................................................................................................... 43
1. Introduction

1.1. Mosaic

As a decorative art, Mosaic is an art-form that depicts a supposed discernable image and is made out of smaller pieces of material such as colored glass, stone, or in the modern sense, tiles (Wikipedia, 2005c). Unlike Found Object Assemblages which belong to the Renaissance Period onwards, Mosaic art-forms predate back to the 4th century BC in Macedonia and can be considered to be a traditional art-form. In the field of Non-Photorealistic Rendering (NPR), much work has been done to create algorithms to mimic these traditional Mosaics (Lansdown & Schofield, 1995). Two successful examples are the ones mimicking the Opus Vermiculatum and the Opus Musivum Mosaics (Battiato, Di Blasi, Farinella, & Gallo, 2006; Di Blasi & Gallo, 2004). The former ‘opus’, or work/composition, refers to Mosaics that outline the shape of the Mosaic motif (supposed discernable shapes in the composition) to create an aural effect, emphasizing the design. Vermis, the Latin word for worm/worm-like, refers to how the tiles tend to worm around the motif to provide the emphasis. The latter ‘opus’ extends the notion of Opus Vermiculatum and refers to Mosaics that extend the worming effect to fill the entire area instead of just around the motifs (Shelby Glass Studio, 2005).

![Figure 1.1: Opus Vermiculatum](http://www.liceus.com/cgi-bin/tcua/2100.asp)

![Figure 1.2: Opus Musivum](http://www.liceus.com/cgi-bin/tcua/2100.asp)

On the other hand, other forms of Mosaic have also emerged that detract from mimicking these traditional forms of Mosaic. In essence, these forms of Mosaic generalize the notion of what constitutes a Mosaic tile. Photomosaic is a Mosaic where the tiles are thumbnail images arranged in a rectangular grid (Di Blasi & Petralia, 2005a). Decorative Mosaic is one where colored tiles
are used to compose the larger image but may be arranged unlike that of traditional Mosaics (Hausner, 2001). Jigsaw Image Mosaic is one that composes the larger image from smaller images like that of Photomosaic, but generalizes the images to be of any arbitrary shape instead of mere rectangular ones (Kim & Pellacini, 2002).

1.2. Found Object Assemblage

In the arts, the term “Found Object” is given to an item/object that would not normally be considered to be art, for example, the everyday objects that we see around us: fruits, cans, furniture, kitchen appliances, etc (Wikipedia, 2005b). An Assemblage is an art-form used to describe a sculpture formed by joining smaller individual pieces together, including that of “Found Objects” (Wikipedia, 2005a). In today’s modern society, Found Object Assemblages are also commonly created to be and known as “trash art” due to their use of recycled materials instead of just any “Found Object”.


Figure 1.4: Found Object Assemblage of Burnt Blocks by Tony Sevil. From Uralla Gallery. Retrieved April 5, 2006, from http://www.urallagallery.com/assemblage.html

1.3. FOAM: Found Object Assemblage Mosaic

This project will introduce a new kind of Mosaic, FOAM, Found Object Assemblage Mosaic, where three dimensional objects are used to compose a larger image/photo. The end result is a three dimensional landscape of objects joined, stacked, and essentially piled together to compose the original image, while at the same time preserving its feature edges, level of detail, and perceivable colors. By factoring a three dimensional component to the Mosaic, FOAM brings a relief-like effect to the resultant composition.
Unlike other forms of Mosaic, FOAM does not try to bring yet a new meaning to the notion of Mosaic tiles. Instead, FOAM replaces this notion of two dimensional tiles with that of three dimensional objects. While this may stretch our very notion of a Mosaic composition, this paradigm shift enables FOAM to circumvent the packing problem that exist in virtually all Mosaic algorithms. FOAM also addresses the implicit level of detail problem in other Mosaic algorithms when the size of the image tiles is not sufficiently small to represent the smaller details of the given image.

Figure 1.5: FOAM
2. Related Work

Despite fundamental differences in their approach, various Mosaic implementations tend to address and revolve around the same set of problems, i.e. how to pack the tiles as tightly as possible. Previous approaches to the Tile Packing problem includes an Energy Minimization approach (Kim & Pellacini, 2002), use of Voronoi Diagrams (Dobashi, Haga, Johan, & Nishita, 2002), and even rigorous mathematical modeling of the problem (Elber & Wolberg, 2003). Most
Mosaic implementations also constraint themselves by the need to emphasize the feature edges of the original image. Dobashi et al (2002) emphasize feature edges by increasing the edge with, essentially outlining them in black. Elber & Wolberg (2003) use the precise placement of tiles at the off-sets of feature curves to emphasize them. Hausner (2001) placed the tiles according to the image gradient calculated from its feature edges in order to emphasize them. Di Blasi & Gallo (2004) also used the image gradient of the image to help orient the tiles and emphasize on the feature edges. Different from Hausner’s method, however, their method allows for some deformation of the tiles, allowing better packing and subsequently better emphasis on the feature edges. For Jigsaw Image Mosaic, Kim & Pellacini (2002) segmented the input image and increased the spacing between them to emphasize the feature edges between segments. On the other hand, Haeberli (1990) did not attempt to emphasize the feature edges at all and merely tessellates the image with variable tile shapes and sizes. Implementations of Photomosaic merely assume the best packing arrangement, i.e. when all their tiles (image thumbnails) are placed regularly in a rectangular grid (Di Blasi & Petralia, 2005a).

While FOAM detracts from the need to pack any notion of a tile, FOAM still addresses the need to emphasize feature edges of the original image. For this purpose, FOAM adopts a method using the image gradient of the image similar to that of Di Blasi & Gallo (2004), essentially mimicking how the traditional Opus Musivum arranges its tiles, as this method yields more visually pleasing results than that of the other methods we observe.

2.2. Database Searching
Along with Photomosaic, some Mosaic implementations also address the problem of searching a database of tiles to find one(s) that resemble the color to be represented as closely as possible. This entails the use of a clever searching algorithm as well as an efficient data structure. Photomosaic searches a database of image thumbnails. Often, the chosen image thumbnail still has to be color-shifted to better represent the hue/saturation/intensity of the required color. Jigsaw Image Mosaic searches a database of image shapes, each with an associated color for one that minimizes an energy minimization function representing the quality of the Mosaic composition (Kim & Pellacini, 2002). Puzzle Image Mosaic searches a database of image shapes for one that best matches the shape of the required Voronoi Diagram cell as well as the color of the original image. With the chosen image shape, a color shift is explicitly performed to match the color to be represented. Unfortunately, this often detracts the viewer from recognizing the original color composition of the constituent image shapes (Di Blasi & Petralia, 2005b). In our implementation, FOAM does not address the problem of database searching. FOAM assumes a complete database
of 1331 objects (11 different objects per RGB channel from 0.0 to 1.0 with increments of 0.1) to circumvent any large overhead required to search for a matching object.

### 2.3. Level of Detail

In virtually all Mosaic implementations, the level of detail problem is only implicitly addressed without any revelation to how the algorithm is able to distinguish between areas of the image that require smaller sized tiles and areas that do not or how the tile size actually varies with the level of detail required. An observation of experimental results from various Mosaic algorithms concludes in two general methods to address the problem. The first more straightforward method is to merely assign the tile size to be smaller than or equal to the smallest detail to be represented. The second method merely assigns some arbitrary values for tile sizes for different segments of the image. Assuming the image can be segmented in some way, a user-defined set of parameters can be easily programmed into the User Interface for this purpose. For the former method, there is no need for any notion of Image segments and only one tile size value is needed for the Mosaic composition. With this in mind, FOAM addresses the level of detail problem by having the object sizes explicitly dependant on image segment size. For this, the sizes of all image segments and a scale parameter, to control the rate of change of object size with segment image size, are needed.

### 2.4. Mosaic on the GPU

Some Mosaic implementations make use of the Graphics Processing Unit (GPU) to help efficiently generate Voronoi Diagrams used as tile placement guidelines or as tiles themselves in their approaches (Hoff, Keyser, Lin, Manocha, & Culver, 1999). Other than to generate Voronoi Diagrams, the GPU is not commonly used by any other Mosaic implementation to help address the problem. In this project, FOAM will utilize the GPU to help in the final step of the composition, i.e. to merge object of different layers of depth into one. In the initial feature extraction stages of the algorithm, the role of the GPU will also be factored in as much as possible to balance the computational load with the CPU.

![Figure 2.9: Voronoi Diagram. From Precision Modeling Laboratory, Inc. Retrieved April 5, 2006, from http://www.pml.co.jp/ghl/2D-fig/voronoi1.jpeg](http://www.pml.co.jp/ghl/2D-fig/voronoi1.jpeg)
3. FOAM Formalization

3.1. Problem Definition

We can formally define the problem as follows:

**Problem (Found Object Assemblage Mosaic):** Given an image $I_a$ and a set of arbitrarily shaped 3D objects $\{F_i\}$, find the image $I_b$ as a set of shapes $\{S_j\}$ such that:

- the union over $\{S_j\}$, as $I_b$, for all $j$, resembles $I_a$ as closely as possible where:
  - the color at point $(x, y)$ in $I_a$ should be as close as possible to the color at $(x, y)$ in $I_b$, and
  - the feature edges in $I_a$ should correspond as close as possible to some shape boundary of $\{S_j\}$; and
- each $S_j$ is a 2D projection on the image plane of a translated and rotated copy of any $F_i$ possibly overlapped by some other $F_i$ and/or incorporating a small deformation.

3.2. Algorithm Description

The algorithm to solve the above FOAM problem is proposed as follows:

For a given input image $I_a$ and a set of three dimensional objects $\{F_i\}$:

1. From $I_a$, using a Canny Edge Detection algorithm, compute an edge image $I_1$.
2. From $I_a$, compute image $I_2$ as a union over $\{X_k\}$, a set of image segments of $I_a$.
3. From $I_1$ and $I_2$, using an Image Gradient approximation algorithm, compute image $I_3$ as a union over $\{L_k\}$, a set of *Vermis* lines surrounding feature edges as object placement guidelines, where for all $L_k$, the perpendicular distance between all $(x, y)$ in $L_k$ to some other closest $L_k$ in the same segment, associated with some $X_k$ of $(x, y)$ in $I_2$, is directly proportional to its segment size by some user-defined set of parameters. Let the set of perpendicular distances between a pair of mutually closest $L_k$ in the same segment be $\{D_k\}$.
4. From $I_a$, $I_3$, and $\{F_i\}$, compute image $I_4$ as a union over a set of shapes $\{S_j\}$:
   
   For all $(x, y)$ of all $L_k$ in $I_3$, find $F_i$ and its associated projection on $I_a$, $S_j$, such that:
   
   a. the center of $S_j$ coincides with some $L_k$ in $I_3$,
   b. $S_j$ intersects as little as possible with all other $S_j$ in $I_a$,
   c. the orientation of $S_j$ follows the Image Gradient of $I_a$,
   d. the dimension of $S_j$ perpendicular to the gradient of its corresponding $L_k$ at its center is equal to the $D_k$ of that $L_k$, and
e. the mean color of \( S_j \) is as close as possible to the color at its center in \( I_a \).

5. For all \((x, y)\) in \( I_a \), compute image \( I_b \) with the following iterative scheme, starting with \( n = 4 \):

   a. For all \((x, y)\) in \( I_b \) not corresponding to any \( S_j \) in \( I_a \), compute \( I_{n+1} \) as a union of shapes \( \{S_j\} \) such that:

      - the center of \( S_j \) coincides with some \((x, y)\),
      - \( S_j \) intersects as little as possible with all other \( S_k \) in \( I_{n+1} \),
      - the orientation of \( S_j \) follows the Image Gradient of \( I_a \),
      - the dimension of \( S_j \) parallel to the image gradient at its center is equal to the \( D_k \) in that segment, and
      - the mean color of \( S_j \) is as close as possible to the color at its center in \( I_a \).

   b. From \( I_n \) and \( I_{n+1} \), compute image \( I_{n+2} \) as a union over \( \{S_j\} \) and \( \{S_k\} \).

   c. If \( n(\{S_j\}) > 5 \), repeat the iteration with \( n = n + 2 \).

4. Implementation Outline

4.1. NVIDIA Image Processing Framework

As the GPU is inherently a 3D graphics hardware, implementing a 2D image processing application on the existing APIs would be quite an awkward task (Jargstorff, 2004; Segal & Akeley, 1994). This is not to say that the existing APIs, such as OpenGL, do not provide for such mechanisms or that the GPU does not provide or allow for such image processing features (NVIDIA Corporation, 2004a; NVIDIA, 2004b; NVIDIA 2005a). However, building such applications on an image-centered framework, instead of directly on the available low-level APIs, would be more advantageous. As such, FOAM will be implemented on the NVIDIA Image Processing Framework (Jargstorff, 2004).

In summary, the framework has the following characteristics and advantages:

a. It is written in C++ as a Model-View-Controller design and is based on the OpenGL 2.0 API, and the GLUT library.

b. It supports Non-Power-Of-Two and up to 16bit RGBA images.

c. Images are ultimately represented as Pixel buffers on the GPU allowing the actual image processing task to be completely done on and by the GPU without unnecessary copying or movement between system memory and GPU memory.

d. Image processing algorithms are implemented in the Cg programming language.

e. It utilizes the following OpenGL extensions:
a. WGL_ARB_pBuffer – to use the GPU Pixel buffers to store intermediate and/or final results of Image Filters
b. WGL_ARB_render_texture – to allow the usage of the buffers as textures
c. WGL_ARB_pixel_format – to allow choosing of a specific pixel type for the framework
d. NV_texture_rectangle – to allow the usage of textures of Non-Power-Of-Two sizes in OpenGL
f. Image Filters are implemented as Filter Graphs with a data-pull mechanism where data is pulled from the previous nodes in the graph, which in turn pull their data from their input nodes up the graph.
g. A high-level interface to the image data allows any OpenGL draw command to draw directly into an Image and also allows the retrieval of OpenGL texture handles to textures containing the Image data.

4.2. Framework Modifications

While the existing framework brings great ease and control over the design of general Image Processing Filters, the framework has not effectively addressed the following issues:

a. The data pull mechanism dictates the flushing of the entire pipeline whenever there is a change in any node in the pipeline, i.e. even when the changes should only affect the inputs of a small number of its nodes.
b. Image filters are designed as a filter graph, but user-defined parameters are ultimately controlled through the final node of the graph.
c. Only the final image data is kept on the GPU while intermediate results are discarded as they flow through the filter graph.
d. The framework only provides high level interfaces for GPU-bound Image data and none for those bound for the CPU.

As such, we have made the following modifications to the framework in order to better implement FOAM as a filter graph in the framework:

a. Input Images of the filter nodes are stored as private members of the nodes. Updates to the input images are only necessary if there are any changes reported by the input nodes. This avoids any unnecessary flushing of the entire pipeline and provides an input cache-like mechanism. This trades-off space on the GPU memory for faster refresh rates of the filter-graphs.
For computationally intensive filter nodes at the ends of some filter graph, their output images are also stored as private members of the nodes. This is especially important for the last most computationally intensive node of the FOAM filter graph.

Some Image Filters are implemented as networks of filter references instead of having each filter with its own private filter node members. The end nodes of filter graphs are then responsible for rearranging the connection of globally available filter nodes, i.e. they are global to other filter graphs. This allows us to discretize user-control over various stages of the filter graph.

With discretized user-control, the framework now provides a mechanism to flush the entire pipeline on demand. For example, for the filter graph $[A \rightarrow B \rightarrow C \rightarrow D]$, if after D has produced its output, changes are made in node B, a pipeline flush is required so that D can force C to update its input and C subsequently forcing B. Unfortunately, there is no efficient way to avoid this overhead of updating B’s input of A. Without the pipeline flush, with the current data pull mechanism, C would not be able to automatically detect the changes made in B. Consequentially, D will not detect changes in C and thus not reflect changes made in B.

With no high-level support for CPU-bound Image data, low-level OpenGL commands are used to transfer data to and from the GPU. Image data is saved on the CPU following the pixel format of the framework. The Image data is only updated as necessary to avoid unnecessary GPU-to-CPU data transfer overhead.
4.3. FOAM as A Filter Graph

FOAM is implemented on the modified framework as a filter graph as follows.

![Figure 4.1: FOAM as a Filter Graph](image-url)
While in the previous sections, we have used the following notions of an image interchangeably, here we will emphasize the subtle differences and follow the following naming conventions to avoid confusion:

- **Given Image**: The original image provided as input to the FOAM filter graph
- **Input Image**: The image to be processed by some filter node or filter graph

### 4.3.1. Input Preparation

This stage consists of the following filter node:

<table>
<thead>
<tr>
<th>Filter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HistogramEqFilter</strong></td>
<td>This Filter performs a Histogram Equalization on the given image.</td>
</tr>
</tbody>
</table>

With the given image, we are posed with the issue of how to better idealize the given image so that the following filter nodes may yield better results. For this purpose, FOAM employs a Histogram Equalization filter on the given image to improve on its contrast. Subsequently, the output of this filter may be used to obtain better results in the Edge Detection or Image Segmentation filters.

From the formal proof by Gonzales & Woods (2002), we obtain the following algorithm to compute the Histogram Equalization of a grayscale image:

a. For all pixels in the Input Image, assume a 16bit grayscale image and create the Image Histogram, i.e. the frequency of occurrence for each grayscale value.

b. From the Image Histogram, compute the Normalized Cumulative Histogram to obtain a Probability Density Function of grayscale value occurrences in the Given Image.

c. The Normalized Cumulative Histogram thus gives the Look Up Table (LUT) to obtain the new gray level values, i.e. the new gray scale value for $x$ is the value corresponding to $x$ in the LUT.

There is, however, no equally direct and efficient way to compute the Histogram Equalization of color images. We can, however, compute a rough approximation by performing the above algorithm separately on each RGB channel and subsequently combine them to obtain the resultant Histogram Equalized color image (Matthew, 2004). We have employed such a method in FOAM and have obtained the following satisfactory results:
Figure 4.2: RGB channels before Histogram Equalization

Figure 4.3: RGB channels after Histogram Equalization

Figure 4.4: Image before and after Histogram Equalization.
As the above results have shown, the caveat in employing such an approximation scheme is that new colors may be introduced in the resultant image (See Figure 4.4 (right)). Despite this difference in color composition, however, we can see that the contrast of the image has indeed been improved.

With the linear nature of the algorithm in computing the Image Histogram and the Normalized Cumulative Histogram, the filter is implemented partially on the CPU and only the LUT step is performed in parallel for all pixels on the GPU. Here, since the size of the LUT corresponds to the entire range of $2^{16}$ grayscale values, the LUT is efficiently made available to the GPU as a 2D texture of size $2^8 \times 2^8$, i.e. as opposed to a 1D texture image (Lefohn, Kniss, & Owens, 2005).

### 4.3.2. Edge Detection

This stage consists of the following filter nodes:

<table>
<thead>
<tr>
<th>Filter Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreyScaleFilter</td>
<td>This filter performs a grayscale conversion from the input image.</td>
</tr>
<tr>
<td>GaussFilter</td>
<td>This filter performs a simple Gaussian blur on the input image. It is a part of the original NVIDIA framework (Jargstorff, 2004).</td>
</tr>
<tr>
<td>SobelFilter</td>
<td>This filter calculates the image gradient of the input image with a Sobel operator.</td>
</tr>
<tr>
<td>UnCannyEdgeFilter</td>
<td>This filter performs the last 2 steps of the Canny Edge Detection algorithm on the input image, i.e. it performs Non-Maximum Suppression and Hysteresis Testing.</td>
</tr>
<tr>
<td>EdgeDetectionFilter</td>
<td>This filter performs a Canny Edge Detection algorithm on the input image and comprises of GreyScaleFilter, GaussFilter, SobelFilter, and the UnCannyEdgeFilter.</td>
</tr>
</tbody>
</table>

Before we are able to create the *Vermis* lines necessary to mimic the tile placements in an *Opus Musivum* Mosaic, we will first need to detect the image features and their corresponding edges from the given image. For that purpose, FOAM employs a Canny Edge Detection algorithm (Green, 2002) and implements it as a filter graph of four filter nodes. In this modular design, the Edge Detection Algorithm can be easily modified to accommodate future optimizations, e.g. employing a better Image Gradient approximation algorithm than the Sobel operator. The additional overhead is also reasonable because throughout these filter nodes, the image data need not leave the GPU memory (Jargstorff, 2004).
The first node, the GreyScaleFilter, performs a grayscale conversion of the input image as a required input for the Edge Detection algorithm. The result is, however, passed on to the second filter node, the GaussFilter, first. Here, a simple Gaussian blur is performed on the grayscale image. Since the Edge Detection algorithm essentially detects changes in the color gradient, the blurring will allow the algorithm to differentiate between the changes that correspond to actual feature edges and the ones that merely correspond to textural attributes. For example, the edges surrounding a piece of clothing should be detected, but the thread linings corresponding to its material makeup should not be detected.

The next step in the algorithm is handled by the SobelFilter node to compute the image gradient with a Sobel operator in order to approximate the edge strength of the image. The Sobel operator employs two 3 x 3 convolution masks to estimate the gradient in the x and y direction correspondingly.

\[
\begin{array}{ccc}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{array}
\]

\[
\begin{array}{ccc}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{array}
\]

The magnitude of the Edge Strength can then be approximated using the following formula (Green, 2002):

\[ |G| = |G_x| + |G_y| \]

In the last node of this stage, the UnCannyEdgeFilter completes the Canny Edge Detection algorithm by performing the last three steps, i.e. with the computed edge strengths, the corresponding edge directions can then be approximated accordingly.
Essentially, edge directions are rounded up or down to directions of either: 0°, 45°, 90° or 135°. Negative angles are ignored, i.e. converted into the allowable range (Green, 2002).

With the edge direction known, Non-Maximum Suppression is then performed in order to obtain thin edge lines, i.e. pixels with edge strengths lower than any of its neighboring pixels in the perpendicular direction of the edge is set to zero. Finally, Hysteresis Testing is done to eliminate any streaking (discontinuous) edges. Essentially, Hysteresis Testing is performed by specifying two threshold values, $T_{low}$ and $T_{high}$ (Green, 2002).

1. If(strength $\geq T_{high}$)
2. strength = 1
3. else if(strength < $T_{low}$)
4. strength = 0
5. else if(any neighbor’s strength $\geq T_{high}$)
6. strength = 1
7. else
8. strength = 0

Like in all other filter nodes, the input images in the filter nodes of this stage are cached as a private member and only updated when any changes are reported by the input nodes. Also, with the mere coordinator role of the final node, the EdgeDetectionFilter, and the linear nature of the filter nodes, the actual filter nodes are implemented as private members of the coordinator node, the EdgeDetectionFilter. The output image of the EdgeDetectionFilter is also cached due to the nature of its input caching and its mere role as a coordinator.

The output of the Histogram Equalization filter is used as input to this stage with the rationale that images with better contrast will have better emphasized edges. For the same set of user parameters (threshold, etc), we obtain the following Edge Images for with and without Histogram Equalization.
There is, however, one caveat of using the Histogram Equalized image as input to this stage, i.e. it may introduce additional edges that need not necessarily correspond to any actual feature edges in the image. However, for our purpose, since the yielding of non-existent edges is outweighed by the yielding of actual edges of a higher contrast version of the given image, this stage’s input will be directly connected to the Histogram Equalization filter in the FOAM filter graph. Another rationale for this is that the edges detected by the algorithm will only affect the final Mosaic as much as the user will allow it to. The final Edge Image detected by the algorithm will ultimately be determined by controlling the parameters in the User Interface. As such, any unwanted edges, if any, can easily be efficiently removed by the user.

### 4.3.3. Image Segmentation

This stage consists of the following filter nodes:

<table>
<thead>
<tr>
<th>Filter Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TwoPassGaussFilter</strong></td>
<td>This filter performs a simple Gauss filter on the input image as two passes of a 1D Gauss filter. It is a part of the original NVIDIA framework (Jargstorff, 2004).</td>
</tr>
<tr>
<td><strong>ColourBitsFilter</strong></td>
<td>This filter performs an Image color depth change from the original 8/16 bits per RGBA channel into 2 bits per channel on the input image.</td>
</tr>
<tr>
<td><strong>FillGapsFilter</strong></td>
<td>This filter approximates an image artifacts reduction algorithm on the input image.</td>
</tr>
<tr>
<td><strong>MultiFillGapsFilter</strong></td>
<td>This filter performs an iterative image artifacts reduction algorithm based on the FillGapsFilter on the input image.</td>
</tr>
</tbody>
</table>
In this stage, we would like to segment the given image into areas of approximately the same color, or actually into areas corresponding to perceivable image features, and then approximate the area/size of each segment. While Image Segmentation deserves a much more rigorous analysis as a deeply investigated Computer Vision problem, here we have employed a more straightforward and efficient method to approach the problem. While such a rough approximation may yield incorrect segmentations (i.e. yields segments within actual segments), the method is again justified by the fact that image segments will only affect the final Mosaic as much as the user will allow it to as its effects are limited by the user-defined parameters associated with it. Another reason, as also mentioned by Di Blasi & Gallo (2004) for justifying an approximate Edge Detection algorithm, is that while a more rigorous framework may benefit the actual image segmentation, it will still be limited by how much it can actually improve on the aesthetics of the final Mosaic composition. The trade-off between its added value for aesthetics and its higher computational complexity is the issue here. For this purpose, we have adopted a purely image space approach.

To segment the image, we first apply a simple Gaussian blur filter to the input image with reasons similar to the Edge Detection algorithm, i.e. in order to eliminate/reduce textural artifacts which will lead to unnecessary segmentation. The next filter node, the ColourBitsFilter, will perform the actual image segmentation. In this node, the input image is simply reduced from its original color depth of 8- or 16-bits per RGB channel into a 2-bit RGB image. This “rounding-off” of color values will automatically cluster like-colors into the same segment. While a better approximation to how colors are physically perceived by the human eye would be to use a color space like that of the CIE L*u*v or CIE L*a*b color spaces (Wikipedia, 2005d) instead of the RGB color space, the actual color representation in hardware as RGBA channels limits us from concluding with an equally direct method. Also, the current scheme already allows us to obtain the following satisfactory results:
On the other hand, a closer look at the resultant image reveals the existence of artifacts within any particular segment and that some of the segments are quite insignificantly small sized. While the Gaussian blur filter has played a large role in reducing much of the artifacts if the ColourBitsFilter were to be used without it, the degree of blurring has an optimal range in which too much would cause the blurring of not only textural artifacts, but also feature edges. As such, we should employ a mathematical morphology operator such as closing or opening in order to compensate for such artifacts (Fisher, Perkins, Walker, & Wolfart, 2003).

An opening operator is defined as erosion followed by a dilation operator, while a closing operator refers to a dilation followed by an erosion operator. Both erosion and dilation operators essentially consist of (typically) a 3 x 3 structuring element of 1s and a notion of background and foreground pixels. The erosion operator on a pixel is thus as follows: if all the neighbors of the pixel correspond to a foreground pixel, it is left as it is. Otherwise, replace it with a background pixel. For the dilation operator: if all the neighboring pixels correspond to a background pixel, it is left as it is. Otherwise, replace it with a foreground pixel (Fisher et al, 2003).
The closing operator is very useful and has the overall effect of enlarging the boundaries of foreground pixels and at the same time, shrinking background colored artifacts within foreground pixel areas. The opening operator, on the other hand, achieves an effect of removing boundary foreground pixels that do not conform to the shape of the structuring element. Opening and closing operators are typically used to remove salt noises / image artifacts depending on whether they correspond to foreground or background pixels (Fisher et al, 2003).

On the other hand, since we are dealing with images with no definite background color for all segments of the image, the use of such operators would not suffice for our purpose (Fisher et al, 2003).
To address this problem, we propose the use of a different operator in the FillGapsFilter to simulate the opening/closing operators for color images. In this operator, we also use a $3 \times 3$ structuring element like typical opening/closing operators. However, the algorithm differs and is as follows:

1. Compute the mode of the neighboring pixels’ and the original pixel’s colors.
2. Set the color of the pixel to be equal to the mode color value.
3. If the original color is also the mode color value, favor the original color over other color modes.

Essentially, the FillGaps operator abandons the notion of background and foreground pixels that have constrained the use of the opening and closing operators. An observation of its effects over images shows how it seems to simulate both the closing and opening operators, i.e. on one hand, it tends to remove “background” pixels within areas of “foreground” pixels (Figure 4.13 bottom), but at the same time, it also removes boundary “foreground” pixels (Figure 4.13 top). The overall effect is that, the FillGaps operator tends to cluster like-color pixels together by either removing oddly positioned pixels or introducing new pixels to fill up gaps.
Figure 4.14: ColourBitsFilter without Gaussian blurred input

Figure 4.15: Five iterations of FillGapsFilter without a Gaussian blurred input
However, just like the problems associated with dilation/erosion, the FillGaps operator will affect all areas of the image indiscriminately, whether it needs further refinement or otherwise (Fisher et al, 2003). As such, the result over some number of iterations does not converge well to an image free of artifacts. While its effects in removing pixels are appreciated, the image’s feature shapes and edges may be harmed by excessive use of the operator.

On the other hand, since here we use the FillGaps operator to merely compensate for any lacking left unaddressed by the Gaussian blur filter, there is no need to perform the FillGaps operator to such a harmful degree. Experimentation has shown that five number of iterations achieves a good balance between artifact reduction and degree of image distortion.

![Figure 4.16: ColourBits with Gaussian blurred input before (left) and after five iterations of FillGapsFilter (right)](image)

With the fixed number of iterations and the linear nature of their data flow, the FillGaps nodes are implemented as an array of private members of the MultiFillGaps filter node. One other thing to note here is that due to the limitations of the Cg programming language (the latest fp30/fp40 profile was used), finding an efficient algorithm to find the mode of a set of values on the GPU is non-trivial. The bulk of the problem lies with the following two restrictions:

1. For-loop structures are essentially unrolled into a linear set of instructions on the GPU. As such, arrays may not be accessed with the index variable of the for-loop.
2. If statement conditionals must be scalar. The compiler does not provide an overloaded version to handle vector inequalities. And with no direct method to translate the vector inequalities into scalar ones, further analysis needs to be done in order to find one that yields the least number of assembly instructions.
The proposed Cg program is thus as follows:

```c
1. void
2. main ( in float2 vTextureUV: TEXCOORD0,
3.     out half4 vOutputColor: COLOR,
4.     const uniform samplerRECT oImage
5. )
6. {
7.     half4 g[9];
8.     half sum[9], nMax;
9.     g[0] = texRECT(oImage, vTextureUV+float2(-1,-1));
10.    g[1] = texRECT(oImage, vTextureUV+float2(0,-1));
11.    g[2] = texRECT(oImage, vTextureUV+float2(1,-1));
12.    g[3] = texRECT(oImage, vTextureUV+float2(-1,0));
13.    g[4] = texRECT(oImage, vTextureUV);
14.    g[5] = texRECT(oImage, vTextureUV+float2(1,0));
15.    g[6] = texRECT(oImage, vTextureUV+float2(-1,1));
16.    g[7] = texRECT(oImage, vTextureUV+float2(0,1));
17.    g[8] = texRECT(oImage, vTextureUV+float2(1,1));
18.    int i, j;
19.    for(i = 0; i < 9; ++i) sum[i] = 0;
20.    for(i = 0; i < 9; ++i) {
21.        // Do for all cells
22.        for(j = 0; j < 9; ++j)
23.            if(dot(g[i]-g[j], half4(1,1,1,1)) == 0)
24.                ++sum[j];
25.    }
26.    nMax = max(sum[0],sum[1]);
27.    for(i = 2; i < 9; ++i)
28.        nMax = max(nMax, sum[i]);
29.    for(i = 0; i <9; ++i)
30.        if(nMax == sum[i]) vOutputColor = g[i];
31.    if(nMax == sum[4]) vOutputColor = g[4];
32.    vOutputColor.w = 1.0f;
33. }
34.}
```

Here, we use multiple for loops to circumvent the restriction for the array index. The `dot()` function is also used to translate the vector inequalities into scalar. The above program yields 360 assembly instructions. As a comparison, using the `length()`, `any()`, and `any()` with `cross()` functions yield 506, 508, and 612 instructions respectively.
With the resultant segmented image, we now calculate the segment area/size of each segment and write, for all pixels of a segment, its count value in its Alpha channel. The algorithm for calculating the Segment Area is a mixture of a region growing algorithm, which is recursive by nature, and a scan line algorithm which can efficiently be iterative implemented. We present the full algorithm here and subsequently show how the role of the recursive region growing part of the algorithm is essential to ensure correctness.

```c
#define unvisited 1

for(all pixels, px) {
    if(px.A < unvisited) continue;

    fRand = getRandomFloat(); // range is [0,1)
    px.A = fRand;
    count = 1;

    top = bottom = px.Y;
    left = px.X - 1;
    right = px.X + 1;

    do {
        noMorePixels = true;

        for(all pixels, px_, from
top left to top right,
top right to bottom right,
bottom right to bottom left,
bottom left to top left) {
            if(same color && px_.A == unvisited) {
                if(no neighbor with same color) continue;
                px.A = fRand;
                ++count;
                noMorePixels = false;

                if(at top) { gotTop = true;
                    count += countR(to left) + countR(to down);
                }
                if(at right) { gotRight = true;
                    count += countR(to up) + countR(to left);
                }
                if(at bottom) { gotBottom = true;
                    count += countR(to right) + countR(to up);
                }
                if(at left) { gotLeft = true;
                    count += countR(to down) + countR(to right);
                }
            }
        }
    } // end do
}
```

// end for
The recursive part of the algorithm corresponds to the following procedure:

```
1. proc countR(direction) {
2.     if(same color && px.A == unvisited) {
3.         px.A = setVisited();
4.         top = countR(to top);
5.         right = countR(to right);
6.         down = countR(to down);
7.         left = countR(to left);
8.         return 1 + top + right + down + left;
9.     } // end if
10. } // end proc
```
Without the recursive part of the algorithm, we can be presented with the following situation:

![Figure 4.17: SegmentAreaFilter Caveat](image)

In the illustration above, we can see that as the algorithm first encounters the area pointed to by the arrow, it ignores the area because it has no neighboring pixels that has already been counted. The algorithm will only be able to realize that it has left out the area when it reaches a point where the two areas meet as shown on the picture on the right. The recursive algorithm is thus necessary to compensate for any areas previously left out of the counting in the iterative algorithm. Correspondingly, these areas must be left out by the iterative algorithm as it should be able to distinguish between, for example, the following 2 scenarios. Of course, here we are assuming that the two segments in the picture on the right have exactly the same color.

![Figure 4.18: Segment Distinction](image)

With the scan line nature of the algorithm, the SegmentAreaFilter node is implemented on the CPU. This hybrid algorithm is an optimization over using a purely recursive algorithm to approach the problem. The main issue against using such a purely recursive approach is the depth of recursion and subsequently, the size of the stack required to run the algorithm. With the hybrid approach, the depth of recursion incurred will depend on the shape of the segment and not on its size. On top of that, with the clustering of segments done by the MultiFillGaps filter, the stack size should no longer be an issue for the algorithm.
Coming back to the Image Segmentation algorithm, while arguably, a similar algorithm should be employed to directly segment the image and at the same time also calculate its area, such an algorithm would not allow us to optimize the resultant segmented image before its areas are to be calculated. On the other hand, if we segregate the step to segment the image and the one to calculate the areas, we now have an additional pass over the entire image as an added complexity. This is in contrast to simply having the ColourBits and MultiFillGaps filters that run in parallel for the image pixels on the GPU.

One other important note for this stage is its corresponding input. While previously we have reasoned for using the Histogram Equalized image as input for the Edge Detection stage, here, we are torn between the two options as the optimal result can be obtained from either one depending on the input image.

![Figure 4.19: With Histogram Equalization (right) the image segmentation is better](image1)

![Figure 4.20: With Histogram Equalization (right) the image segmentation introduces unnecessary segments](image2)

In support for using the Histogram Equalized image as input, as the contrast of the input image is improved, the saturation of the various perceivable image segments is also improved. As a result of this improved saturation, the ColourBitsFilter will yield a better clustering of
colors. On the other hand, in support for using the original input, improving on the image contrast may actually introduce additional segments within actual image segments. For example, as the contrast between an object’s color and its color under lighting (specular highlights) is improved, the ColourBitsFilter may cluster the highlighted sections of the object as a separate segment instead. With this reasoning, the input to the Image Segmentation stage is made user-dependent and can be easily toggled through the User Interface.

4.3.4. **Vermis Lines Creation**

This stage consists of the following filter nodes:

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTFilter</td>
<td>This filter performs a Distance Transform operation over the input image.</td>
</tr>
<tr>
<td>LvLineFilter</td>
<td>This filter computes the Level Line Matrix from the input image (Di Blasi &amp; Gallo, 2004).</td>
</tr>
</tbody>
</table>

In order to create the guidelines for object placement, we employ a similar algorithm used by Di Blasi & Gallo (2004) to mimic the tile placements of an *Opus Musivum* Mosaic. A Distance Transform is computed over the Edge Image from the previous stage. The algorithm employed generalizes into a Distance Transform algorithm for Sampled Functions as described in full by Felzenszwalb & Huttenlocher (2004). Essentially, a Distance Transform of an Edge Image will fill in the non-edge pixels with values corresponding to their distance towards the closest edge pixel. This problem is modeled after finding the minimum convolution of a sampled function with a parabola in terms of the squared Euclidean distance (Felzenszwalb & Huttenlocher, 2004). The resulting algorithm correctly computes the Distance Transform of the input image in linear time.

![Figure 4.21: Distance Transform of an Image with the red pixels corresponding to its feature edges](image)
The algorithm for calculating the distance transform algorithm for the squared Euclidean distance in one-dimension is as follows (Felzenszwalb & Huttenlocher, 2004):

1. \( k \leftarrow 0 \) // Index of rightmost parabola in lower envelope
2. \( v[0] \leftarrow 0 \) // Locations of parabolas in lower envelope
3. \( z[0] \leftarrow -\infty \) // Locations of boundaries between parabolas
4. \( z[1] \leftarrow +\infty \)
5. for \( q = 1 \) to \( n - 1 \) // Compute lower envelope
6. \[ s \leftarrow ((f(q) + q^2) - (f(v[k]) + v[k]^2))/(2q - 2v[k]) \]
7. if \( s \leq z[k] \)
8. then \( k \leftarrow k - 1 \)
9. goto 6
10. else \( k \leftarrow k + 1 \)
11. \( v[k] \leftarrow q \)
12. \( z[k] \leftarrow s \)
13. \( z[k + 1] \leftarrow +\infty \)
14. end if
15. end for
16. \( k \leftarrow 0 \)
17. for \( q = 0 \) to \( n - 1 \) // Fill in values of distance transform
18. \[ D_f(q) \leftarrow (q - v[k])^2 + f(v[k]) \]
19. end for

In order to accommodate the accuracy of the Distance Transform values from the CPU for the GPU, the values are encoded across the RGBA values of the image, i.e. each RGBA channel holds 2 significant figures totaling up to an accuracy of 8 decimal places altogether.

With the Distance Transform of the image obtained, we can compute a Level Line Matrix of the input image (Di Blasi & Gallo, 2004), i.e.

\[
LLM(x,y) = \begin{cases} 
1 & \text{if } DT(x,y) \% width = 0 \\
0 & \text{elsewhere}
\end{cases}
\]

Essentially, the Level Line Matrix corresponds to an image whereby the detected feature edges of the image are uniformly reproduced, i.e. in an equally spaced manner, outwards and inwards, but taking into account the presence of other nearby feature edges as well. This corresponds directly with our notion of Vermis lines.

### 4.3.5. Mosaic Composition

This stage consists of the following filter nodes:
FirstPassFilter | This filter performs the first pass rendering of 3D objects.
--- | ---
SecondPassFilter | This filter performs a secondary pass rendering of 3D objects over some existing pass’s rendering.
JoinPassFilter | This filter merges two consecutive pass renderings into one image.
CollageFilter | This filter manages the inputs and outputs of the SecondPassFilter and JoinPassFilter to compose the final Mosaic result.

The actual Mosaic composition is done at this stage and is essentially realized by the SecondPass and the JoinPass filters. These two filters make up the bulk of the iterative algorithm used to compose the Mosaic as previously defined in Section 3.2. The CollageFilter plays the role of a coordinator to loop the output of the JoinPassFilter back to the SecondPassFilter as well as the JoinPassFilter. This processing of input image by the SecondPassFilter and then the input image and the output of the SecondPassFilter by the JoinPassFilter is referred to as one iteration. For the first iteration, the input of the JoinPass and SecondPass filters come from the FirstPassFilter. At the end of each iteration, the CollageFilter is also responsible for testing if the result so far has converged as required.

The algorithm for the FirstPassFilter and SecondPassFilter are as follows:

```
1. proc FirstPassFilter() {
2.     for(all pixels) {
3.         if(notVisited() && isGreen()) {
4.             if(isViable(width)) {
5.                 drawObject();
6.                 markVisited();
7.             } // end if
8.         } // end if
9.     } // end for
10. }
11. proc isViable() {
12.     center = findGreenPixelAtDistance(0.375*width);
13.     object = findObject(center.Color);
14.     resizeObject(object, width);
15.     rotateObject(object, center.gradient);
16.     checkIntersection(object.dimensions);
17. }
18. proc markVisited() {
19.     for(all pixels in a rectangle of dimensions equal to dimensions of object projection) {
20.         mark(pixel.rotateBy(object.gradient));
21.     }
22. }
23. }
```
As we can see in the FirstPass algorithm, the color of the object is actually an approximation and does not actually correspond to the color of the pixel at its center. The reason for this rises from the following dilemma: The dimensions of the object and hence the ultimate position from which we should sample our color depends on the object chosen. On the other hand, before our chosen object depends on the required color. As such, to approximate with an acceptable error range, we have employed the following method:

As we can see, optimistically, the maximum error corresponding to the above approximation scheme only amounts to a mere 1/8 of the object width on a best case average.

Despite their similarities, the FirstPassFilter and the SecondPassFilter were designed with different restrictions in mind. For the FirstPassFilter, as it corresponds to the top most layer of the Mosaic, directly visible to the viewer, it is extremely important for the object projections to intersect with each other as minimal as possible, and at the same time to conform to the...
computed *Vermis* lines. Since the object is always oriented with the longest side parallel to the image gradient, we have the maximum area of the intersection equal to

\[
\left( \frac{x}{2} \tan(22.5^\circ) \right) \left( \frac{x \sqrt{2}}{2} - \frac{x}{2} \right) = 0.04289x,
\]

where \( x \) is the lane width.

**Figure 4.23: Maximum Intersection between lanes**

With no *Vermis* lines to follow, the intersection calculation for the SecondPassFilter is made to be between any object. An alternative approach is to have the objects placed along some form of guideline as well. For this purpose, we can either follow the *Vermis* lines or the feature edges. However, if we follow either of these guidelines per se, the final Mosaic may not be guaranteed to have all its pixels occupied by some object projection. For this purpose and due to the simplicity of the approach, we place no restriction over the object placement for the SecondPassFilter and simply place the objects at any background pixel found in the previous iteration.

In making the connection between object-space and image-space coordinates, we have obtained the following formulae, i.e. they are dependent on the Field Of View parameter, \( \theta \). From experimentation, we found that the fish eye effect of the `gluPerspective()` can be reduced to an acceptable amount while still maintaining the three dimensional sense of the projection when \( \theta \) corresponds to 15 degrees. We also model the equation by considering the ratio of the object-space to the image-space units so that the value can be easily modified in the implementation.

\[
\tan(0.5 \times \text{FOV}) = \frac{(h')/2}{d} = \frac{(R \times h)}{(2 \times d)}
\]

So,

\[
d = \frac{(R \times h)}{(2 \times \tan(7.5))}
\]

where:

- \( h' \) = height of image in object space
- \( h \) = height of image in pixels (image space)
- \( R \) = ratio of object space units to image space pixels, \( R = h'/h \)
- \( d \) = distance between the eye and the drawing plane in object space
While in the algorithm, we associate the intersection testing with a rectangle with dimensions equal to the dimensions of the object’s projection, in the actual implementation, we have reduced the area considered to merely 36%, i.e. only 60% from each dimension is considered. This value is obtained by experimentation and is necessary because the actual image gradient values obtained for the pixels are not entirely accurate. This lack in accuracy may be attributed to the nature of the employed Distance Transform algorithm. Thus for this reason as well, we are limited from using the GPU to automatically generate the exact positions to draw the objects, e.g. by considering the intersections between the Vermis lines and lines corresponding to certain values of the image gradient. The following picture shows how the gradient of the image corresponding to some angle actually relates to an area instead of a straight line along the actual gradient.

![Figure 4.24: Pixels corresponding to gradients between 90° and 180°](image)

The color space used for finding a matching object is the RGB color space, however, we can also approximate the color distance to the CIE L*u*v color space to obtain results closer to how we normally perceive color. The following algorithm approximates the color distance in the CIE L*u*v color space (Riemersma, 2006):

\[
\begin{align*}
\bar{r} &= \frac{C_{1,R} + C_{2,R}}{2} \\
\Delta R &= C_{1,R} - C_{2,R} \\
\Delta G &= C_{1,G} - C_{2,G} \\
\Delta B &= C_{1,B} - C_{2,B} \\
\Delta C &= \sqrt{\left(2 + \frac{\bar{r}}{256}\right) \times \Delta R^2 + 4 \times \Delta G^2 + \left(2 + \frac{255 - \bar{r}}{256}\right) \times \Delta B^2}
\end{align*}
\]

Subsequently, the algorithm for the JoinPassFilter is as follows:
While there may be some caveats in using an image-space approach to the problem, i.e. by detecting and replacing background pixels at each iteration, the probability of encountering an infinite loop is fairly low, i.e. only if the color of the object’s projection is equal to the background color. As the background color assumes an 8bit RGB value, the probability amounts to merely $2^{-24}$ or 0.000000059604644775390625. Also, in order to actually achieve an infinite loop, the object’s projection must be able to yield just the background color and nothing else. On the other hand, circumventing the problem is trivial as we can simply randomize the background color in the SecondPassFilter at every iteration. For our purpose, however, such a refinement was felt to be unnecessary.

While an alternative approach would be to somehow further utilize the available Z-buffer of the GPU, such an algorithm would dictate the necessity to maintain all drawn objects as 3D objects and not as 2D projections like in the proposed algorithm. While such an approach is ideal because, assuming we need not recalculate the positions of all the objects, we can then adjust the lighting on all the drawn objects and be able to see the reflected changes in near real-time (depending solely on the lighting computation time). In the proposed algorithm, the entire process from iteration one until it converges would have to be repeated if any lighting parameters are to be changed. Having said this, the question of whether employing an object-space approach to the problem, instead of the proposed image-space one, can significantly increase the results’ aesthetics and compensate for the added computational cost, begs to be answered.

In this implementation, we have chosen to orientate the objects so that its projection on the image is maximal. By maximizing the projection area of each drawn object, we can reduce the number of iterations required to converge towards the Mosaic. Consequentially, the performance of the FirstPassFilter will affect the number of iterations required. While the FirstPassFilter only operates on pixels corresponding to *Vermis* lines, the SecondPassFilter operates on any background pixel and thus depends greatly on how much area has already been covered by object projections. Thus, with this maximal projection constraint, we only have 4 different ways to orientate our object, ignoring the rotations about the z axis, i.e. corresponding to the full 180 degree rotation combinations about the x and y axis.

```plaintext
1. proc JoinPassFilter() {
2.   for(all pixels of Iter(n)) {
3.     if(background color)
4.        Iter(n).pixel.color = Iter(n+1).pixel.color
5.   } // end for
6. }
7.
```
One other thing to note here is that the calculation of the image gradient is done on the CPU. While this can easily be done on the GPU like that for computing the Edge Image, it is important to consider the degree of accuracy that can be obtained. Since the RGB channels correspond to 16-bit values, i.e. the half data type on the GPU, the number of represented decimal places is merely 3. While the same encoding on all 4 RGBA channels like that of the DTFilter can be employed, the question of whether all image gradient values for all pixels need to be calculated begs to be answered. As the image gradient only needs to be calculated for pixels corresponding to object drawing positions, we justify the choice of having the image gradient calculation on the CPU instead. Also, there is the issue of the arc tangent function, used to calculate the image gradient, actually being translated rather inefficiently to 13 to 15 assembly instructions on the GPU,

With the scan line nature of the algorithm, the filter is implemented on the CPU. This is however, inevitable regardless of the nature of the algorithm because the algorithm will ultimately need to make the necessary OpenGL calls on the CPU in order to draw the objects.

In order to exploit the possibility that the same object may be drawn multiple times for the Mosaic, display lists are created for the drawn objects as a form of batch processing (Wloka, 2003) as opposed to merely use the Immediate mode in OpenGL. The actual loading of objects is done via the Advanced Visualization Lab GLM library (Indiana University, 2006; Robins, 2000), originally written by Nate Robins who is also known for porting the GLUT library into the MS Windows 32-bit platform. However, in view of a modular and more refined means to manage the objects, a separate class, FObject, is created to handle object management. Essentially, the FObject class acts as a middle person between FOAM and the AVL GLM library. It handles tasks such as: finding an object of some mean color from its database, setting and accessing of object dimensions. The class is also the one that ultimately makes the OpenGL calls to draw the object, and hence changes to how the objects are oriented at the drawing position can be easily modified in this class. The class plays a major role in terms of bridging the image space and object space unit conversions as well. Most importantly, however, the class is also responsible for caching the object descriptions as well as their corresponding display lists. The cache is implemented as a circular array and is initially allowed to increase dynamically from 32 to 256 entries. After reaching its maximum size, the cache employs a replacement policy that discards the oldest entry. Other cache replacement policies can also be easily implemented in the FObject class.

4.4. Implementation Caveats
4.4.1. AVL GLM Library Constraints
As FOAM essentially makes a Mosaic of three dimensional objects, the final result can only be as good as the object and its eventual representation on a 2D plane. As such, FOAM depends greatly on the AVL GLM library (Indiana University, 2006; Robins, 2000). In our experimentation, we have found several inconvenient limitations of the library that have hindered FOAM from reaching its fullest potential.

1. Only PPM textures are supported
2. Object transparencies and texture masking are not fully supported

4.4.2. Resource Consumption:
In terms of computation, FOAM detracts from completely saturating the CPU and only utilizes 40% of its Pentium 4 3.2GHz test bed. This may be attributed to the caching system of the FObject class. We have also briefly discussed memory issues of FOAM in section 4.3 in discussing the Stack Size requirement for running the SegmentAreaFilter algorithm, the maximum size of the FObject cache, as well as the issue of caching intermediate images in the GPU to ensure near real-time user interaction. Currently, FOAM peaks at about 110 to 120MB of memory to compose a Mosaic. The current setting of the cache is actually set to limit the total peak memory of FOAM at this amount. Increasing the maximum cache size will subsequently increase the peak memory consumption. While we have not been able to obtain figures to analyze the amount of GPU memory consumed, FOAM runs well in both 128MB and 256MB GPU memory environments, which are both standard specifications for commercial grade high performance Graphic Cards. On the other hand, if memory consumption on the GPU becomes an issue, the buffering of intermediate results can simply be reduced to a minimum. Another possible approach is to employ a more sophisticated memory management for the Image data in the framework. Currently, only a simple reference counting is done.

4.4.3. Database Creation
Unlike other Mosaics where either a database is not needed or the database normally involves a collection of images, a database of OBJ files, especially ones that can conform to the restrictions of the AVL GLM library as well as be visually pleasant to “tile” can be hard to come by. Currently, the database for FOAM is built using data from a single OBJ file and its corresponding material library and texture images. The texture images are converted into PPM using an Adobe Photoshop macro that allows batch processing of images. The actual generation of object files and their corresponding material libraries is done via a bash script that also allows for a more uniform naming convention, i.e. to ascertain which object corresponds to which mean color. The final database is simply hard coded into the program as
a static 3D array of object names. A bash script was also employed to generate the header file that contains the static array. To realistically compare with the performance of other Mosaic algorithms, we generated a total of 1331 objects of distinct color (from the same OBJ file) corresponding to 11 per RGB channel ranging from 0 to 1 by 0.1 increments. We could have generated the objects following a different color space instead but the current method serves as a more realistic approximation to how an ideal object library should be like. The actual conversion between color spaces is implemented in the FObject class instead.

5. Results

FOAM was tested on an Intel® Pentium® 4 Prescott 3.2GHz system with 1GB DDR2 RAM and a PCIe x16 NVIDIA GeForce 6600 256MB DDR. The following reflects the results obtained:

![Figure 5.1: FirstPassFilter and Vermis Lines for Lena image](image)

The Vermis Lines were calculated within 2 seconds from the given input. The computation of the Vermis lines alone is sufficient for real-time user interaction.
As we can see from the results above, FOAM correctly addresses the problem definition as described in Section 3, i.e. the color at pixels of the Mosaic matches as perceivably close as possible to the color of those pixels in the original image. Secondly, the object placement accentuates the feature edges from the original image. Lastly, all the pixels of the image have indeed been replaced by some projection of an object. A much closer look at the results shows a rather graveled look of the whole Mosaic, much like a relief sculpture of some degree. One can imagine an actual physical realization of the Mosaic: If we were to stack the objects used for the Mosaic and arrange them in exactly the same way in a sandbox, one would obtain the same visual treat looking at the composition from above downwards.
The following contrasts the results obtained from using the approximated CIE L*u*v color distance as opposed to the RGB color distance:

![Figure 5.4: Color Distance in RGB (left) and approximated CIE L*u*v (right) color spaces](image)

The following table lists the timings associated with various images used in this paper:

<table>
<thead>
<tr>
<th>Image</th>
<th>Dimensions</th>
<th>FirstPassFilter Timing</th>
<th>Mosaic Composition Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena</td>
<td>512x512</td>
<td>22s</td>
<td>1m05s</td>
</tr>
<tr>
<td>Strawberry</td>
<td>800x669</td>
<td>42s</td>
<td>1m08s</td>
</tr>
<tr>
<td>Flower</td>
<td>585x582</td>
<td>27s</td>
<td>1m58s</td>
</tr>
</tbody>
</table>

**Table 5.1 Mosaic Timings**

One important note here is that since the object sizes used for different images are different, the timings are not directly comparable with each other. A closer look at the algorithms suggests that the running time for the FirstPassFilter depends on the number of *Vermis* Lines and the sizes of the objects. For the SecondPassFilter, it depends on how well the FirstPassFilter performs and also on the sizes of the objects, in particular, the size of the smallest object allowable.
6. Conclusion

6.1. Summary
In this paper we have shown how FOAM deals with the Tile Packing and Level of Details problems. More specifically, we have shown how by shifting the paradigm into a three dimensional space, the Tile Packing problem commonly faced by Mosaic algorithms can be circumvented. FOAM also proposes a solution to the Level of Detail problem, often left implicit or unraveled in most Mosaic algorithms.

In its implementation, FOAM has emphasized various trade-offs between computation complexity and the achievable aesthetics, especially in terms of Image Segmentation as well as the final Mosaic creation stages. By implementing Image filters in both CPU and GPU, FOAM has illustrated some limitations to the GPU programming paradigm as compared to solving problems on the CPU. In particular, we have shown how limitations in the GPU language constructs tend to impede its level of programmability. Secondly, we have also seen how some problems can be naturally sequential and thus not suitable for the GPU.

In conclusion, while most Mosaic algorithms are normally considered under the domain of Non-Photorealistic Renderings, FOAM has shown how a paradigm shift into the three dimensional domain can bridge the NPR and PR domain, i.e. while FOAM emulates a new artistic rendering style and thus belongs to the NPR domain, further optimizations and modifications can be made for FOAM to ensure better Photorealistic renderings of the drawn three dimensional objects.

6.2. Advantages of FOAM
FOAM thus presents us with the following advantages:

1. Implemented as a filter graph, FOAM becomes very modular and changes can be easily factored into the filter graph.
2. Saturating both the CPU and GPU to approach the problem, FOAM was implemented more efficiently than a strictly CPU-bound implementation.
3. Using three dimensional objects instead of mere two dimensional images, FOAM deals better with the traditional Mosaic’s notion of concave tiles. In fact, FOAM has no restriction over the shape of object used for the Mosaic.
4. With its domain in the three dimensional space, FOAM makes room for works from the PR domain to be factored in to achieve photorealistic visually pleasing results.
5. By mimicking the tile arrangement method of the *Opus Musivum* Mosaic and compensating for gaps by use of the Z-buffer, FOAM has addressed the problem of defining edges rather aesthetically as well as efficiently for the given problem definition.

6. With its three dimensional domain, FOAM is able to circumvent the Tile Packing problem.

7. FOAM has also proposed a reasonable approximation scheme to automatically addresses Level of Detail problem to allow uncompromising detail from the final Mosaic.

8. Drawing its strength from extracting image features, FOAM is an application of various solutions to the Computer Vision problems such as Edge Detection, Image Segmentation. FOAM makes room for more sophisticated image feature extraction algorithms to be used (See Section 6.3).

9. FOAM illustrates the use of the GPU, natively a specialized 3D SIMD processor, for Image processing, more specifically in the NPR domain.

### 6.3. Recommendations for Future Work

As mentioned before, FOAM makes room for much future work to be done.

1. Implemented essentially using Pixel Buffers, FOAM can be improved by using instead the new Frame Buffer Object (FBO) specification from NVIDIA (NVIDIA 2005b).

2. FOAM can benefit from a better framework for User Interface. One such candidate is to use the Microsoft Foundation Class Library. Due to the Model-View-Controller design of the current framework and its tight utilization of the GPU library, the porting may easily be non-trivial. Another alternative is to integrate FOAM as an Image Processing plugin, e.g. with the Adobe Photoshop CS2 SDK.

3. As the bulk of the computation falls in the actual Mosaic composition stage, the use of the DirectX 9 Instancing API instead of OpenGL’s Display Lists may yield better performance.

4. As FOAM has left the problem of Database Searching left unaddressed, FOAM can benefit greatly from employing an efficient data structure to represent the database. One such candidate is to use the Antipole Clustering Strategy which has been efficiently implemented for Photomosaic (Di Blasi & Petralia, 2005a).

5. Lastly and most importantly, FOAM can utilize more sophisticated image feature extraction algorithms to achieve even more pleasing visual results. One such candidate is to apply a solution to the Shape-From-Shading problem in order to extract height values associated with various objects in the image so as to approximate its shape. The resulting Mosaic can be imagined to approximate to that of a Relief sculpture (Prados, Faugeras, & Camilli, 2004).
References


44