Rendering Images using Objects as Primitives

Course project - CS 638, Spring 2009

Lorenzo De Carli

Contents

(a) Giuseppe Arcimboldo, *Vertumnus*

(b) Octavio Ocampo, *Family of birds*

Figure 1: Early examples of object-based rendering

1 Introduction

The research for stunning visual effects has been a trademark of many artists throughout the centuries. One of the most interesting among such effects is the rendering of a well-known (to human beings) physical object using wisely-combined representations of other objects. The effectiveness of this approach depends on the fact that the observer is "tricked" into believing that she/he is facing a familiar entity, while realizing - after a short time - that he is really observing a collection of completely different objects.

The surrealistic connotation of this device may lead to believe that it was invented in modern times, but the idea has been known at least since the renaissance. The XVI-century Italian painter Giuseppe Arcimboldo ([1]) was probably the first to use it; more recently, the concept has been used by modern artists such as Octavio Ocampo ([2]). Figure 1 depicts some examples. As the reader can infer from the examples, the painter must have significant skills to be able to "fool" human perception and produce convincing result. This is probably the reason why, as far as we know, no attempts has been made to create an algorithmic, computer-based version of this technique. The aim of this project is to create an algorithm implementing object-based rendering in a (partly) automated way.

Figure 2: Giuseppe Arcimboldo, *The librarian*

1.1 Main issues

In general, creating a convincing results requires significant knowledge of both the object that is represented, and the objects that are used as *rendering primitives*. This is particularly clear in the painting in fig. 2 (also by Arcimboldo). In this example, a human being is "rendered" using images of books. It is interesting to see how, for the different parts of the human body, the author uses many instantiations of the concept of "book", with many different orientations, sizes and shapes. Directly simulating the creative process is likely to be very difficult, if possible at all. Consider, for example, the way the hair are hinted using an open book. While the association between these two entities can be conceived by a talented artist, there is no way it can be established by an automatic program (and there are probably many other cases like this). Another limitation of any automated technique is that the number of objects that can be used as rendering primitives is necessarily limited. Conversely, a human being can easily imagine any number of objects in different sizes, shapes and conditions.

From what has been said, it may seem that generate results similar to the examples in this section with an algorithm would be infeasible. However, despite reaching the quality of man-made results appears unlikely, the literature presents a series of techniques that can be leveraged to approach the problem in a simple way. Such techniques are described in the next section.

2 Related Work

Despite no implementation of object-based rendering can be found in literature, a variety of related works exist. In particular, both *texture transfer* and *photomosaics* involve the reproduction of a source image using a set of other images.

2.1 Texture Transfer

Texture transfer ([3]) is a simple extension to the Image Quilting algorithm by Efros and Freeman. With Image Quilting, a destination texture is grown picking tiles from a source texture image. At every step, a new tile is added to the generated texture. In the basic version of the algorithm, new tiles are chosen trying to minimize the *overlapping error* between them and the rest of the image. This approach aims at limiting the visual artifacts that are usually visible around the borders between contiguous tiles.

Texture transfer allows to "guide" the texture growth process in order to reproduce a source image. If $diff_{overlap}$ is the SSD between a tile and the areas of the target image it is going to overlap, and $diff_{source}$ is the SSD between the same tile and the correspondent region of the source image, at every step the algorithm picks the tile that minimizes:

$$w * diff_{overlap} + (1 - w) * diff_{source}$$

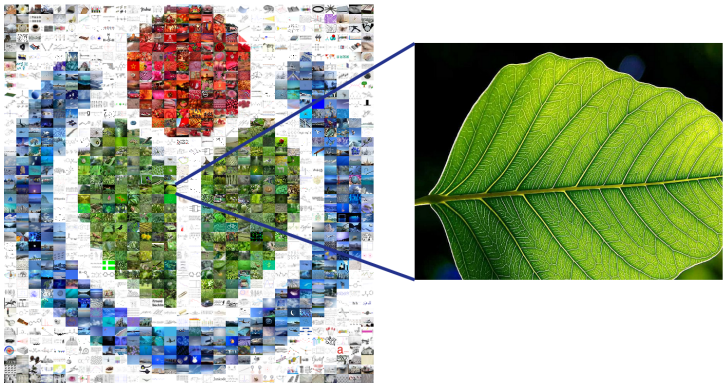
The parameter w allows to give different relevances to the two terms. Texture transfer produce results in which the synthesized texture resembles the source image. Figure 3(a) presents an example.

2.2 Photomosaics

Photomosaics are images in which a source picture is recreated using a large number of other images, scaled to a very small size. If the resulting image is not observed closely, it looks similar to the source picture. However, a closer examinations reveals its real nature of juxtaposition of hundreds or thousands small images. In this context, such images have the same roles of the tiles in texture transfer. Figure 3(b) is a photomosaic representing the logo of the Wikimedia project ([4]) - one of the pictures composing the image has been magnified to illustrate the technique.

Usually, photomosaics are created by dividing the source images in many rectangular areas. Every area is then replaced by a miniature of a picture picked from a library. The simpler way of performing the matching between areas and library images is to compare the average colors. However, several applications use more advanced approaches - for example, Metapixel ([5]) claims to use a "wavelet metric" (which is probably a comparison in the frequency domain).

(a) Texture transfer



(b) Photomosaic

Figure 3: Texture transfer and Photomosaics

2.3 Considerations

Texture transfer and Photomosaics can be considered as forms of objet-based rendering, since they use images as rendering primitives to "recreate" another image. Their main limitations are the constraints they pose on the **size**, **shape** and **location** of the tiles they use. In particular, all the tiles are usually of fixed size and have a rectangular or square shape. Also, the source image is divided in regions, and every region is mapped to exactly one tile. Little or no overlap is allowed between tiles, and the set of positions where a tile can be "pasted" in the destination image is limited.

Starting from these considerations, the following questions can be posed: *It is possible to emulate object-based rendering (as performed by humans) by extending techniques such as texture transfer and photomosaics?*. This approach is precisely the one which has been used in this course project. The proposed algorithm is described in sections 3 and 4, while section 5 presents preliminary experimental results.

3 Building object libraries

A preliminary condition for any object-based rendering technique is the availability of a suitable object library. Building such a library by hand is likely to be a cumbersome process - tools that can simplify this task are highly desirable. For the purpose of this project, we leveraged the LabelMe dataset from the MIT CSAIL lab ([6]). LabelMe it is a large online image collection, with several desirable properties:

1. Most images in the collection have been manually segmented. Segmentation information can be used to extract specific objects from pictures.

2. Objects depicted in the images are associated with words describing them. This enables users to search the library for specific contents.

3. A custom toolbox is provided to access the library from within Matlab. The toolbox exports functions for searching the database, downloading images and performing basic processing on them.

3.1 Extracting Objects from Images

Images in the LabelMe collection typically contain several objects; to make them available to an object-based rendering algorithm, such objects must be extracted and saved separately. To achieve this result, the LabelMe Matlab toolbox has been extended with a new function, *tune_img*, whose prototype is:

function tune_img*(D, objectname, src_filename, obj_index, dst_name)*

where D is the index of the LabelMe image database, *objectname* is a string that describes a specific object (e.g. "fruit"), *src_filename* is the image containing the object, *obj_index* is the index of the object (in case more than one object of the same type is present), and *dst_name* is the file where the extracted object will be saved.

The function extract *objectname* from *src_filename*, and create a new image containing only the selected object. This resulting image is then rescaled in three different sizes[1]; for each of the three new pictures, the function also generates a logical mask which defines which pixels belong to the object, and which pixels belong to the background. Figure 4 depicts the output of *tune_img* given a sample input.

3.2 Allowing Tiles of Different Sizes and Shapes

The extraction process aims at overcoming of some limitations of existing related techniques (see section 2.3). By resizing source objects in different ways, tiles of varying sizes are made

[1] Resizing is performed so that the sizes (along the principal dimension) of the three generated images are respectively 32, 64 and 128 pixels.

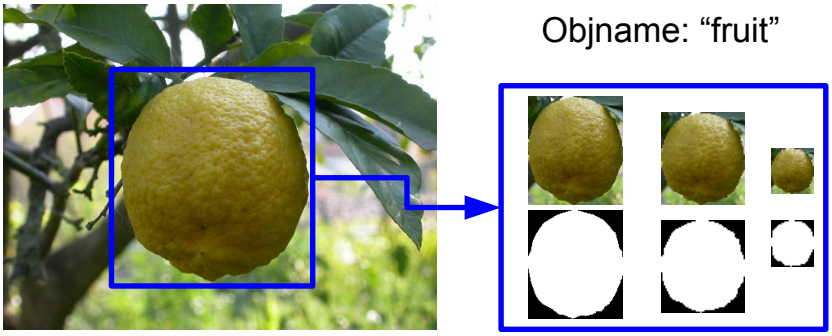


Figure 4: Operations performed by *tune_img*

available to the rendering algorithm. This approach is somewhat rough, as the set of possible tile sizes is quite limited. However, it is worth remembering that the aim of this work is just to create a proof-of-concept object-based rendering technique. The logical mask that accompanies each generated image enables the use of tiles of irregular shape. In fact, when the rendering algorithm (described in section 4) pastes the tile in the destination image, only the pixels marked by the logical mask are copied. In this way, the background of each tile is discarded.

3.3 Implementation details

The tune_img function described previously is implemented in the file *tune_img.m*. The function performs the following operations:

1. The image containing the desired object is loaded into Matlab by using the LabelMe function *Lmread* (line 7 in *tune_img.m*).

2. The region of the image containing the object is determined by calling *Lmobjectboundingbox* (line 8).

3. The object is extracted using the bounding box computed during the previous step (line 9).

4. The image of the object is rescaled to three different sizes (lines 10-34). During this step, logical masks for the objects are also created using the function *Lmobjectmask*. The original library function has an issue: if multiple objects with the same label were present in the image, it created a logical mask covering all of them. Therefore, the function has been extended, and can now receive an object index as an additional parameter. If for example, there are 4 objects labeled “fruit” within a single image, and the index value is 2, the function will only create a mask for the second object.

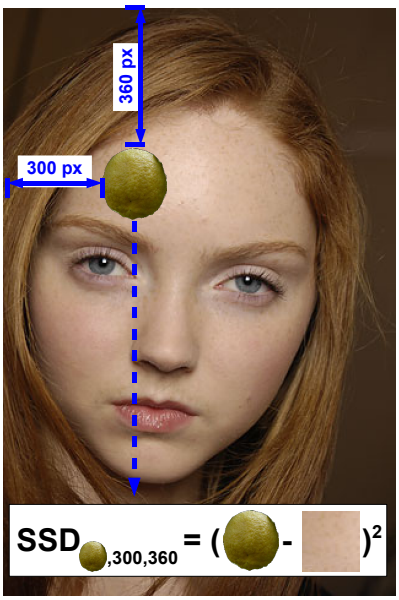


Figure 5: Matching objects with the source image

4 Object-based Rendering

The approach used to implement object-based rendering consists of a 2-step algorithm. In the first step, every picture in the object library is matched with the source image at different row and column offsets[2]. In the second step, the best matching solutions are used to render the source image.

4.1 Step 1 - Matching

Matching consists in computing the SSD between an object and a corresponding region in the source image. In the example in fig 5, an object - a lemon - is being matched with a source image - a human face, using 300 as column offset and 360 as row offset. Note that the matching process is guided by the object mask: the pixels on the background of the lemon are not considered in the SSD computation. The matching process starts from the upper left corner and "move" the object to the left (by increasing the column offset) until it has covered the entire width of the source image. Then, it increase the row offset, and perform the process again. The sequence is iterated until the whole source image has been covered.

The choice of the row and column steps involves a tradeoff between the computing time and the precision of the rendering algorithm. For testing purposes, the row step has been

[2]We make the assumption that the source image is large in comparison with the library objects, where "large" means that the ratio between the dimensions of the source image and the dimensions of the library objects is > 10.

set to $0.25 * tile_{v\_size}$, and the column step to $0.25 * tile_{h\_size}$, where v_size and h_size are respectively the vertical and horizontal dimension of the objectt that is being considered.

4.1.1 Combining the Solutions

Once the algorithm has computed the SSD values for a given object, it saves them in a *.mat* file. When all the objects in the library has been matched with the source image, the contents of the resulting .mat files are loaded in a single vector. Every element of the vector is a Matlab *struct* containing the following fields:

1. **row:** Row offset at which the SSD has been computed.
2. **col:** Column offset.
3. **sol_id:** Solution identifier: index of the object image within the library.
4. **class_id:** Class identifier: index of the object within the library. Images of the same object with different sizes have different *sol_ids*, but the same *class_id*.
5. **size_id:** Size of the image, ranging from 1 (smaller) to 3 (larger).
6. **wasd:** Sum of absolute differences. Despite the algorithm has been designed to use the sum of square differences, the implementation uses absolute differences. The reason is that absolute differences give similar results and are quicker to compute.

After the merging phase, the resulting vector is sorted according to the *wasd* field. At the end of the sorting process, the best solution (i.e. the one with the lowest value) is at the beginning, and the worst solution at the end.

4.1.2 Implementation Details

The function *preprocess_img* (file *preprocess_img.m*) performs the matching of an object with the source image. The prototype is:

function preprocess_img*(tile_img, tile_mask, source_img, sol_id, class_id, size_id)*

Where *tile_img* is the object, *tile_mask* is the object mask, and *source_img* is the source image; the last three fields are used to initialize the corresponding fields of the entries in the solutions vector. First, the function loads the tile and the source image (lines 3-9) and computes the row and column steps (lines 11-15). Lines 17-28 are a simple adjustment to the source image size to ensure that the object will always fit within the image. In lines 30-51, the function cycles computing the sum of absolute differences between the tile and the source image at different offsets. Lines 53-54 sort the solution vector, which is then saved to a Matlab file (lines 56-57).

4.2 Step 2 - Rendering

The rendering step is the core part of the algorithm. The solutions vector created during the previous step is visited, picking solutions and pasting them into the output image. The algorithm runs until a target coverage of the source image has been reached, or it gets to the end of the solutions vector. Since the vector is ordered, solutions are considered starting from the best one and proceeding towards the worst one - this ensures that a solution will be considered only after all the solutions with a better (smaller) SAD value.

4.2.1 Filtering the Solution Vector

Unconditionally picking all the solutions found in the vector would cause two significant issues:

1. **Excessive overlapping:** Solutions computed using the same object in adjacent regions of the source image tends to cluster. In other words, if the $i-th$ entry in the solution vector is relative to object O at offset (r_O, c_O), it is likely that solutions $i+1...i+k$ will be relative to the same object O, with similar offsets. k can vary - empirically, it is in general around tenths of solutions. The consequence is that several instantiations of the same object will appear almost in the same place, overlapping with each other and creating an unpleasant visual effect.

2. **Predominance of some objects over others:** Unavoidably, some objects in the library will match better than others with the source image. For example, if the source image is a human face, objects with a color close to the human skin will lead - on average - to better solutions. The consequence is the rendering process will tend to show a strong bias towards some objects, almost completely ignoring others.

Both issues can be tackled by performing simple filtering on the solution vector:

Avoiding overlapping/solution clustering: Before a new solution is picked, it is tentatively added to the result, and its overlapping with other already placed solutions is computed. If this overlapping is above a configurable threshold (*max_overlap*), the solution is discarded. The overlapping is defined as:

$$Overlap = \frac{Reassigned\_pixels}{tilesize_{Width} * tilesize_{Height}}$$

Where the numerator is the number of pixels already defined in the result[3] that will be covered by the new tile, and the denominator is the tile area.

Ensuring fair distribution of object types: During the rendering process, the algorithm keeps track of how many times every object has been used. This is accomplished by keeping a counter for every *class_id* in the library (see 4.1.1). Suppose an entry is being considered for

[3] Initially the result image is empty.

inclusion in the result. If i is the *class_id* of the object, N is the number of possible *class_id* values (i.e. the number of objects in the library), and C_i is the number of times object i has already been picked, then the new entry is picked only if the following inequality is satisfied:

$C_i \leq class\_scale\_factor * \frac{1}{N} * \sum_{j \neq i} C_j$

4.2.2 Masking the Source Image

In general, the user may want to render only a portion of the source image. Consider, for example, the picture of a man: rendering both the man and the background will most likely lead to a confused result. Therefore the algorithm receives as input, together with the source image, a mask specifying which regions in the image should be rendered.

4.2.3 Rendering Algorithm Pseudocode

The following is a high-level pseudocode version of the rendering algorithm:

```
1  Input: solutions, source_mask, target_coverage, max_overlap, min_inclusion,
2         class_scale_factor
3  Output: result
4
5  coverage = 0;
6  i = 0;
7  result = {empty image};
8  solutions_set = {empty set};
9  for ( h = 0; h < #objects_in_the_library; h++ )
10   solutions_count[h] = 0;
11
12 while ( coverage < target_coverage && i < size(solutions)) {
13   current_solution = solutions(i);
14   ovl = compute_overlap(result, current_solution);
15   solution_usage = solutions_count[current_solution.class_id] + 1;
16   inclusion = compute_inclusion(source_mask, current_solution);
17
18   if ( ovl <= max_overlap &&
19        solution_usage <= class_scale_factor * avg(solutions_count) &&
20        inclusion >= min_inclusion ) {
21         add_solution(solutions_set, current_solution);
22         solutions_count[current_solution.class_id]++;
23   }
24   i++;
25 }
26
27 for ( h =  size(solutions_set; h >= 0; h-- )
28   paste_solution(solutions_set[h], result);
29
30 return result;
```

The algorithm receives as input the vector of solutions, the mask delimiting the region of interest, the desired coverage of the source image, the maximum overlap between tiles, the

minimum overlap between each solution and the region of interest in the source image, and the class scale factor (to avoid excessive repetitions of the same object in the result). The result is initially set to an empty image, as the set of picked solutions. The counters for all the *class_ids* are initially set to 0 (all the initializations are performed in lines 5-10). In lines 12-25, the solution vector is traversed. The overlap between the new solution and the rest of the image is computed in line 14. The overlap between the new solution and the region-of-interest in the source image is computed in line 16 (this second overlap must be above the *min_inclusion* threshold for the solution to be accepted). Lines 18-20 verify if the solution is acceptable, according to the criteria described in section 4.2.1. If the solution is accepted, it is added to the "chosen solutions" set (*solutions_set*). Once enough solutions have been picked, they are added to the result image, starting to the worst solution until the best (lines 27-28). This inverse traversal ensures that a solution will never be occluded by a worse one.

The rendering algorithm has the advantage of being simple to implement; the running time is also limited (few minutes to generate a result starting from a 1200 * 700 source image). However, it consists of a greedy approach, which does not ensure the optimal result. Also, it does not guarantee that the target coverage of the source image will be reached (although in all the experiments adequate coverage was always achieved).

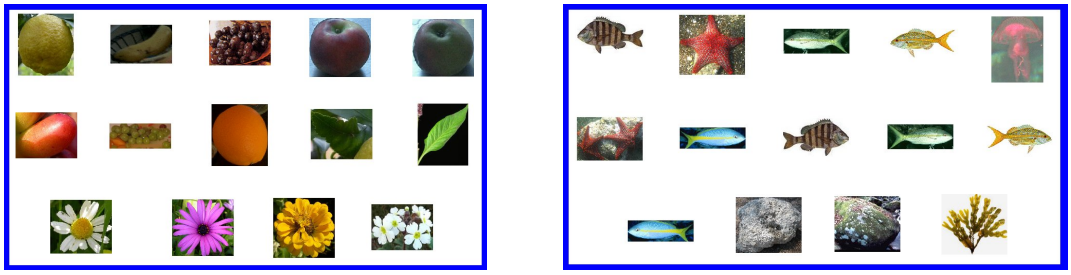
4.2.4 Implementation details

The rendering algorithm is implemented in the function *fill_image* (file *fill_image.m*), whose prototype is:

function [new_img, new_mask] = fill_image*(source_img, source_mask, sols, images, allowed_classes, allowed_sizes)*

source_img is the source image, *source_mask* delimits the region of interest, *sols* is the solution vector, *images* is the set of library objects, *allowed_classes* is the set of objects, among those in the library, that will be used in the rendering process, and *allowed_sizes* is the set of allowed tile sizes. The last two parameters can be used to guide the rendering process, restricting it to a specific set of tiles and sizes. To avoid cluttering the function with too many parameters, *target_coverage*, *max_overlap*, *min_inclusion* and *class_scale_factor* described in the pseudocode (4.2.3) are hardcoded in the script - the user can change their default values by editing the M-file. The implementation is not described in detail, as it contains a significant amount of Matlab-specific code, and a pseudocode version has been already presented. Briefly, lines 12-25 in the pseudocode (solution vector traversal) map to line 24-63 in the M-file, and lines 27-28 in the pseudocode (result generation) map to lines 65-81 in the M-file.

The function returns two objects: the result image and a mask describing which pixels of the result have been assigned. The second parameter is needed if the user wants to separately render different regions of the source image, and then merge the results. As explained in the next section, this approach significantly improves the quality of the generated image.



(a) "Fruit" library (b) "Fish" library

Figure 6: Sample object libraries

5 Experimental Evaluation

To evaluate the algorithm, two libraries have been created. Each library includes 14 objects in 3 different sizes, leading to a total of 84 images. A summary of the objects in the libraries is depicted in fig. 6

5.1 Full Image Rendering

Fig. 7 depicts the result of the algorithm when run on the whole region of interest. Fig. 7(a) is the source image, fig. 7(b) is the image mask delimiting the region of interest, and fig. 7(c) is the rendered image. The result highlights several issues:

1. The accuracy of the method in rendering the source image is not enough to get a clear and recognizable result.
2. While the algorithm fairly distributes the choice of tiles among all the object in the library, randomly mixing the objects across all the image contributes to the confusion in the result. Using different objects for different areas of the image would probably lead to a better result.
3. The juxtaposition of many tiles with different shapes and colors creates high-frequency spatial noise in the image, which is especially noticeable in areas that, in the source image, do not have a lot of details (e.g. the forehead).

The rest of this section explains how it is possible to tackle the first two issues by allowing user intervention in the algorithm.

5.2 Rendering Different Regions Separately

The quality of the results can be improved by allowing the user to segment the image in multiple regions, and to render them separately. The user also has the responsibility of determining which tiles and tile sizes should be used when filling each region. This approach

(a) Source (b) Mask (c) Result

Figure 7: Results obtained by rendering a full image in a single step

allows to "blend" high-level knowledge about the image, provided by a human, into the algorithm. For example, if the source image is a human face, a user may want to separate the facial features (eyes, nose, mouth) from the rest, and render them separately using only small tiles (to allow for finer matching between the source image and the tiles). In the same way, it may define a region enclosing the hair, and render it with large tiles, since hair do not have a strong structure. Fig. 8 presents an example of image segmentation.

Fig. 9 presents result of images that have been divided in multiple regions, which have been rendered separately. The individual results for each region have then been merged to obtain a single result. For images 9(a) and 9(b) five regions were defined, covering face contour, eyes, nose, mouth and hair. An additional region, covering the body, was created for image 9(c). Region masks are not depicted for brevity's sake.

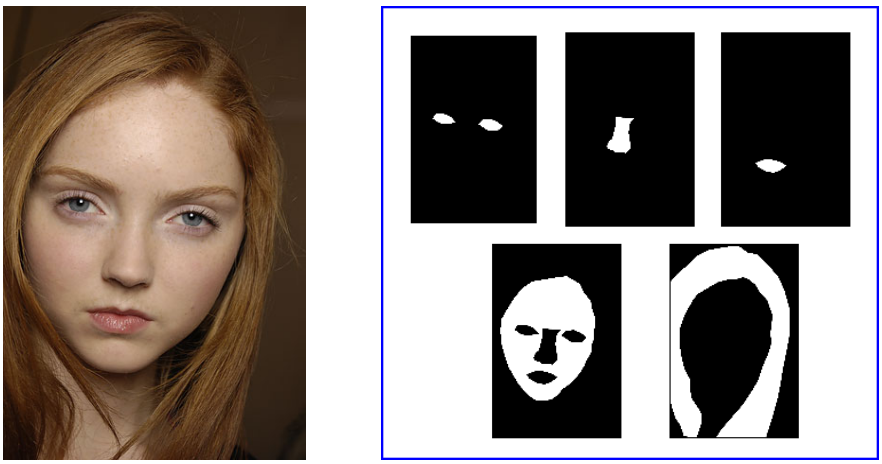
5.3 Implementation details

5.3.1 Rendering the whole image in a single step:

To render the whole image in a single step (as in fig. 7) the user must call *fill_image* (see 4.2.4) and pass a region covering the entire image to the function.

5.3.2 Rendering different regions separately

To render different regions separately (as in fig. 9), the user must create the region masks and then call *fill_image* multiple times, one for every region. The simplest way to create regions is to use the Matlab *roipoly* function. Every time it runs on a region, *fill_image* produces a result where only the specified region has been rendered, and a new mask. This mask specifies which pixels have been filled, and can be used to merge renderings of different regions into



(a) Source image (b) Segmented regions

Figure 8: Segmentation example

(a) Segmented rendering - 1 (b) Segmented rendering - 2 (c) Segmented rendering - 3

Figure 9: Results obtained by rendering different region separately

the final result. The merging process is done using *merge_masks* (file *merge_masks.m*):

function final_img = *merge_masks(source_set)*

source_set is a vector of structs, where every element must include a rendered region in a field called *img*, and the mask generated for that region by *fill_image* in a field called *msk*. Using the masks as a guidance, the function assembles all the rendered regions into a final result. Note that the user has to create *source_set* by hand from the results of the various runs of *tune_img*.

6 Conclusions

This report presented a proof-of-concept technique for object-based rendering. The technique gets inspirations from the work of several artists, in particular the XVI-century Italian painter Giuseppe Arcimboldo, and consists in rendering a source image using a collection of images of various object - *the rendering primitives*. The proposed technique works by computing the matching of all the rendering primitives with the source image, and then picking the best results to perform the actual rendering. Experimental evaluation showed that running the algorithm in a completely automated way leads to somewhat confusing results, but the quality improves significantly if the process is guided by human interaction.

Future work may include reducing the amount of required user interaction, as currently the user has to manually segment the source image. The process could be simplified by using a segmentation algorithm. The issue with this approach is that even the best automatic segmentation solutions in literature ([7], [8]) do not always lead to proper segmentation. Another possible direction for improvement is to detect the level of details in the source image, to process different areas in different ways. For example, areas with a low level of detail could be blurred in the rendered image, filtering out the high-frequency spatial noise often present in results (see section 5).

Finally, it is important to point out that the technique is not aimed at producing a photorealistic rendering of the source image. Quite on the contrary, the goal is to produce a highly abstracted result - in this result, the source image is merely used as a guide to assemble a collection of objects in a meaningful way. In this context, the results are promising, and the technique can be already used as a "building block" in a manual rendering process.

References

[1] Wikipedia page on Giuseppe Arcimboldo, <*http://en.wikipedia.org/wiki/Giuseppe_Arcimboldo*>.

[2] Wikipedia page on Octavio Ocampo, <*http://en.wikipedia.org/wiki/Octavio_Ocampo*>.

[3] A. Efros, W. Freeman, *Image Quilting for Texture Synthesis and Transfer*, SIGGRAPH 2001.

[4] Wikimedia project home page, <*http://wikimediafoundation.org*>.

[5] Metapixel home page, <*http://www.complang.tuwien.ac.at/schani/metapixel/*>.

[6] LabelMe image library, *http://labelme.csail.mit.edu/*.

[7] D. Comaniciu, P. Meer, *Mean Shift: A Robust Approach toward Feature Space Analysis*, IEEE Trans. Pattern Analysis Machine Intell. 24(5), No. 5, 2002.

[8] J. Shi and J. Malik, *Normalized Cuts and Image Segmentation*, IEEE Trans. Pattern Analysis Machine Intell. 22(8), 1997.