TEXTURE SYNTHESIS AND PHOTO-REALISTIC RE-RENDERING OF ROOM SCENE IMAGES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Kyle J. Ziga

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT ix

1 Introduction 1

2 Textures 4

- 2.1 What is a texture? 4
- 2.2 Texture Synthesis 5
  - 2.2.1 Previous Work 7
  - 2.2.2 Our method 9
- 2.3 Results 17
- 2.4 Future Work 19

3 Image Re-rendering 22

- 3.1 Perception 22
- 3.2 Scene Reconstruction 22
  - 3.2.1 Depth 24
  - 3.2.2 Lighting 28
  - 3.2.3 Reflectance 37

4 Implementation 40

- 4.1 Depth Data to Object File 40
- 4.2 Texturing the Objects 41
- 4.3 Lighting 43
- 4.4 Material Properties 45
- 4.5 Post-processing 47

5 Results 50

Page

6 Future Work 53

REFERENCES 54

LIST OF TABLES

Table Page

2.1 Initial parameters by texture class 16

LIST OF FIGURES

Figure Page

1.1 Example output of the DzineSteps tool. On the left is the original image. On the right is the modified image. 2

1.2 Example product image that a user may want to replace a surface with. . . 3

2.1 Example texture types including random (left), irregular (center) and repeating (right) 5

2.2 Visual representation of texture being mapped to a mesh and then rendered to an image [5]. 6

2.3 Example of a successful texture synthesis result. Left is the input texture, right is the synthesized output. 7

2.4 Illustration of parameter effect on output using [9]. 9

2.5 Example texture classes. Left: Stochastic, Center: Irregular, Right: Regular.11

2.6 Left: Example 3x3 pixel neighborhood, Center: Comparison of neighbors to center pixel, Right: Local binary pattern codeword. 13

2.7 Example of a local binary pattern encoded image. 14

2.8 Illustration of patch size progression through user feedback. Left: Each box represents the patch size for each iteration of texture synthesis, Right: Green box shows appropriate patch size on the order of the fundamental repeating structure. 17

2.9 Experimental results: Examples of successful synthesis results. The smaller image is the input sample and the larger image is the synthesized output. Output synthesized at twice the input size. 20

2.10 Experimental results: Examples of failed synthesis attempts. The smaller image is the input sample and the larger image is the synthesized output. Output synthesized at twice the input size. The red boxes highlight errors in the synthesized output. 21

2.11 Illustration of the poor results common for wood textures. 21

3.1 High level system diagram. 24

3.2 Flow chart of depth estimation process. 26

Figure Page

3.3 Example of estimated depth map (right) from original image (left). 29

3.4 Example of an environment map captured with a 360 degree panoramic camera, a similar image could generated using a mirrored ball. 30

3.5 Example of an environment map captured with a 360 degree panoramic camera, a similar image could generated using a mirrored ball. 30

3.6 Flow chart for in-view light source estimation. 31

3.7 The 9 3x3 Laws masks used as part of the Make3D features. Vector combinations labeled above each image. 33

3.8 Filter masks for oriented edge filters spaced at 30 degrees apart. 34

3.9 Example output of the in-view light classifier. Left is the original image while right is the output with detected sources marked in red. 35

3.10 Example of objects rendered into panoramic image environments, material used is brushed aluminum. The top image is the rendered object, the bottom image is the environment map used during rendering. 36

3.11 Intrinsic image breakdown example. Top image is the original, bottom left is the shading component, bottom right is the reflectance component. . 38

4.1 Example of an image split into foreground and background. The foreground and background are then loaded as 3D models and textured with the original image. Top is the original image, bottom left is the background model, bottom right is the foreground model. 41

4.2 Example of a 3D object with and without texturing. On the left is a cube with a standard material, all settings default. On the right is the same cube with a standard material but a texture map was provided. 42

4.3 Texture used during the texture mapping for the cube in Figure 4.2. 42

4.4 Illustration of the different types of lights available in ThreeJS. 43

4.5 Examples of a rendered scene with varying illumination intensity from 4 watts to 60 watts. 44

4.6 Comparison of homography and our result. Left image is our result, right image is homography. 45

4.7 Illustration of diffuse vs. specular reflection. Arrows represent incident light rays. 46

4.8 Example renderings of different ThreeJS material types. (a) is the basic material, (b) is Lambert material, (c) is Phong material and (d) is the standard material. 48

Figure Page

5.1 Final output for a kitchen scene, backsplash is the replaced surface. 50

5.2 Final output for a kitchen scene, island counter is the replaced surface. . . 51

5.3 Final output for kitchen scene, wall is the replaced surface. 51

5.4 Final output for outdoor scene, brick pillars are the replaced surfaces. . . . 52

5.5 Final output for indoor living room scene, carpet is the replaced surface. . 52

ABSTRACT

Ziga, Kyle J. M.S, Purdue University, December 2018. Texture Synthesis and Photorealistic Re-rendering of Room Scene Images. Major Professor: Fengqing Zhu.

In this thesis, we investigate methods for texture synthesis and texture re-rendering of indoor room scene images. The goal is to create a photorealistic redesign of interior spaces by replacing surface finishes with a new product based on a single room scene image. Specifically, we focus on automating this process to reduce manual input while enabling high-quality and easy-to-use experience. The most common method of rendering textures into a scene is called texture mapping. Texture mapping involves mapping pixels in a texture sample to vertices in an object model. Typically, a large texture sample is required to perform texture mapping properly. Given a small texture sample, texture synthesis creates a large sized texture that appears to have been made by the same underlying process. In the first part of this thesis, we present a method of texture synthesis that automatically determines a set of parameters to produce satisfactory results based on the texture types. The next challenge is to create a photorealistic re-rendering of the synthesized texture in the room scene image. 3D scene information such as geometry, lighting and reflectance is crucial to making the re-rendered image realistic. These properties contribute to the image formation process and must be estimated to create a scene-consistent modification. Knowing these parameters allows effects like highlights, shadows and inter-object reflections to be maintained during the re-rendering process. We detail methods for estimating these parameters from a single indoor image. Finally, we will show a web-based implementation of these methods using the WebGL library ThreeJS.

1. INTRODUCTION

Realistically manipulating images and video has seen many applications with the rise of virtual and augmented reality in recent years. These applications span fields such as medicine, providing medical students a safe and controlled environment for practicing surgical techniques, military, heads-up displays provide real-time information to air and ground troops [1], the gaming industry, and construction or interior design, visualizing changes before committing them.

From a single image many people are able to grasp scene level properties such as illumination, relative positions of objects and material properties. An ordinary viewer can almost always tell when lighting is inconsistent, such as missing or out of place highlights and shadows, or when the perspective of an object or surface is not quite right. Even with the technological improvements made in the previous decades, it is still difficult for computers to robustly estimate these same properties which come relatively easily to human viewers. Changes to an image must be consistent with these scene properties to be realistic.

Many complex interactions take place in the image formation process. Light travels through a scene and is reflected, refracted, absorbed and diffracted by objects. The interactions are directed by the geometry of scene objects as well as the material the objects are made of. In photo-realistic computer graphics rendering these interactions are attempted to be quantized and simulated. For example, individual light rays are “traced” from the image plane back through the scene and their interactions with objects are determined by the properties of the synthetic surfaces. The crucial step of any project involving realistic re-rendering of images is explaining the interactions that took place during the image formation. This process of recovering intrinsic properties of a 3D scene from a 2D image is called inverse rendering.

In this thesis we will discuss applications of inverse rendering in the interior design space. We have worked with a company, DzineSteps, who provide a state-of-the-art virtual visualization platform for surface and color selection in interior design. They provide a software to help clients create a virtual gallery showcasing products such as flooring, tile and counter tops on appropriate surfaces. This can increase the emotional engagement of the buyer while eliminating guesswork and visually answering the question, “How will it look in my room?” An example output of the DzineSteps tool can be seen in Figure 1.1.

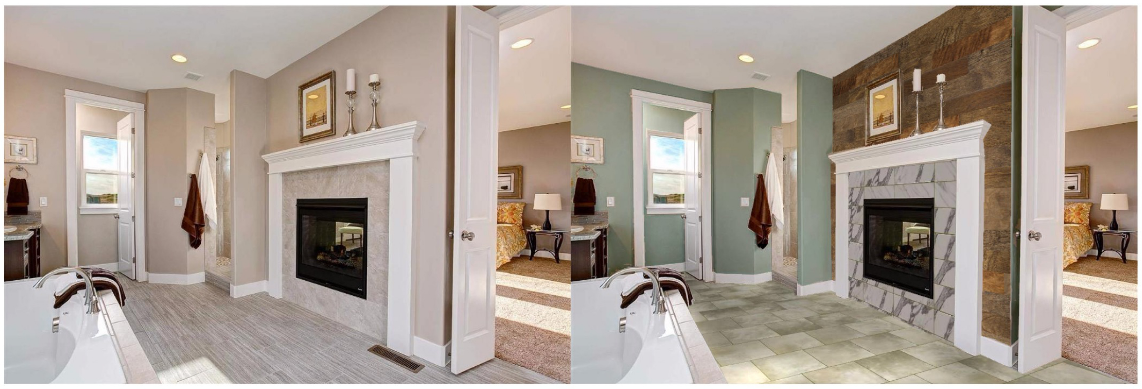

Fig. 1.1. Example output of the DzineSteps tool. On the left is the original image. On the right is the modified image.

The tool allows a user to select a surface of interest using a click-based segmentation [2]. The user can then upload or select a new product to apply to the surface of interest. An example of this new product image can be seen in Figure 1.2. From this information our tool will automatically generate foreground and background 3D models of the scene from an estimated depth map. In parallel with depth estimation scene illumination will be estimated including both sources visible in the image and global illumination which contributes to the scene from behind the camera’s field of view. Surface reflectance is estimated as well and applied to the 3D object mesh. Combined, these components are used to reconstruct the 3D scene from the image.

The surface of interest is then re-textured with the selected product, scene lighting is applied and an output image is rendered.

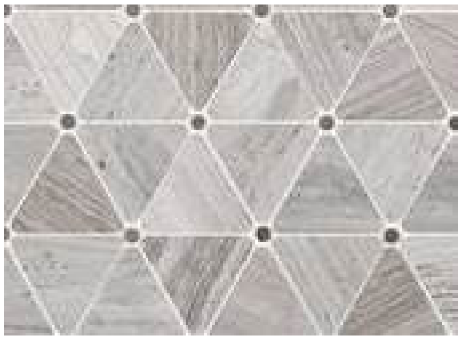

Fig. 1.2. Example product image that a user may want to replace a surface with.

To allow for rendering complex scene properties using off-the-shelf rendering software we represent our scene using 3D objects. For 3D objects texture mapping can be used for defining surface color, specular reflection, surface normals, transparency, diffuse reflection, shadows, or surface displacement [3]. The use of textures to provide surface color details is the most common. This is done by warping the texture image to adhere to the geometry of the surface, mapping texture pixels to object vertices. For this to be done properly a large enough texture sample needs to be provided. Most texture samples are small images, therefore texture synthesis must first be used to generate a large enough texture sample for texture mapping. This is described in section 2.2.

This thesis will first describe the work that has been done towards developing an easy-to-use texture synthesis algorithm. Then we will explain methods for estimating scene properties from a single image to reconstruct a 3D scene. Finally, a web based implementation will be described using the WebGL library ThreeJS.

2. TEXTURES

In this section we describe the importance of textures in computer rendered images. We then introduce a method for automatically creating large texture samples from small samples. These large texture samples are used during the rendering process for texture mapping, or applying texture information to 3D objects.

2.1 What is a texture?

Texture can describe a variety of natural phenomena with repetition; we are interested in those textures which represent visual appearance. One of the main goals of our project, and computer graphics in general, is to reproduce the realism of the physical world. Textures are ubiquitous in real images making them important for conveying a realism in computer rendered images by providing the detailed information of the appearance of a surface [4]. Examples of different types of textures can be seen in Figure 2.1. Some textures have a strong repeating pattern while others allow for more variation or imperfection. Still others seem to be completely random. We will see how these differences in texture types will factor into our algorithm.

The details of a surface can be created using a technique known as texture mapping. When computer generated scenes are rendered to an image the locations on the image where the object vertices should be is determined and the points are projected. When an object is textured, portions of the texture image are found which correspond to the projected points of the object [5]. An illustration of this can be seen in Figure 2.2. For the texture to maintain proper scale after the mapping a large enough texture sample must be used. The user uploaded texture samples will commonly be small. These will be similar to the sample images found at home improvement store websites for items such as flooring, tile or counter tops. These small samples will need

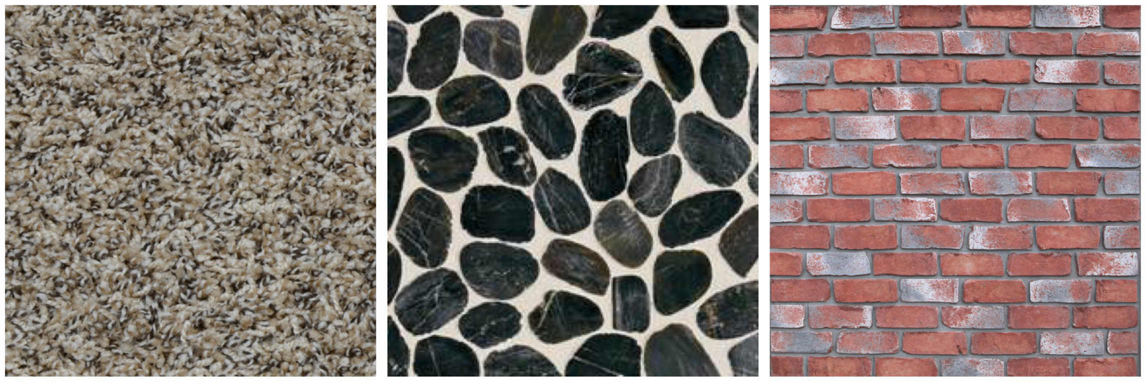

Fig. 2.1. Example texture types including random (left), irregular (center) and repeating (right)

to be grown to a large enough size so as to maintain the scale of the pattern during texture mapping. For this we will need to use a texture synthesis algorithm. Texture synthesis is the process of creating a large texture from a small input texture. This is described fully in Section 2.2.

2.2 Texture Synthesis

Texture synthesis is the procedure of "growing" a large texture image given a small sample texture. Texture synthesis has a variety of applications in areas such as computer vision, computer graphics and image processing. These applications include occlusion fill in by image inpainting, texture mapping, image compression and more. Also, with the growth of augmented and virtual reality applications, efficient texture synthesis algorithms have become more important. An example texture synthesis result can be seen in Figure 2.3. The synthesized texture should not be a repetition of the sample pattern. Instead it should look as though it was created by the same underlying process. In this way only small texture samples are needed to generate large sized texture data.

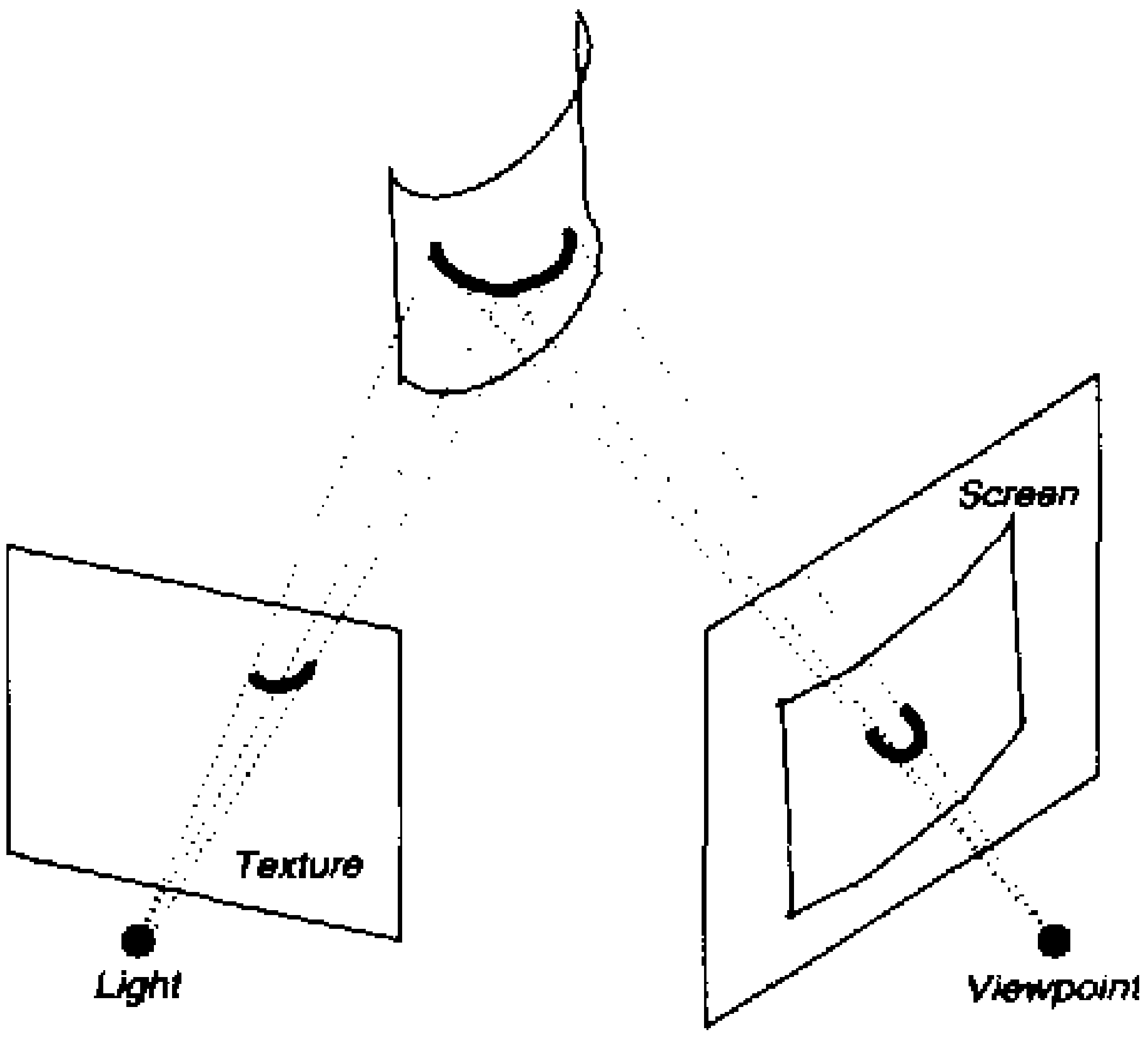

Fig. 2.2. Visual representation of texture being mapped to a mesh and then rendered to an image [5].

Current texture synthesis methods require a user to understand and carefully set parameters to produce satisfactory results. The DzineSteps tool wants to allow an untrained user to make image modifications, therefore requiring they set parameters is undesirable. We leverage differences in texture types to create a method which can automatically synthesize a wide range of textures. To try and account for errors we introduce a user feedback method of parameter correction. This rids the need for users to select actual parameter values while still utilizing the user as a means of validation for the synthesized result.

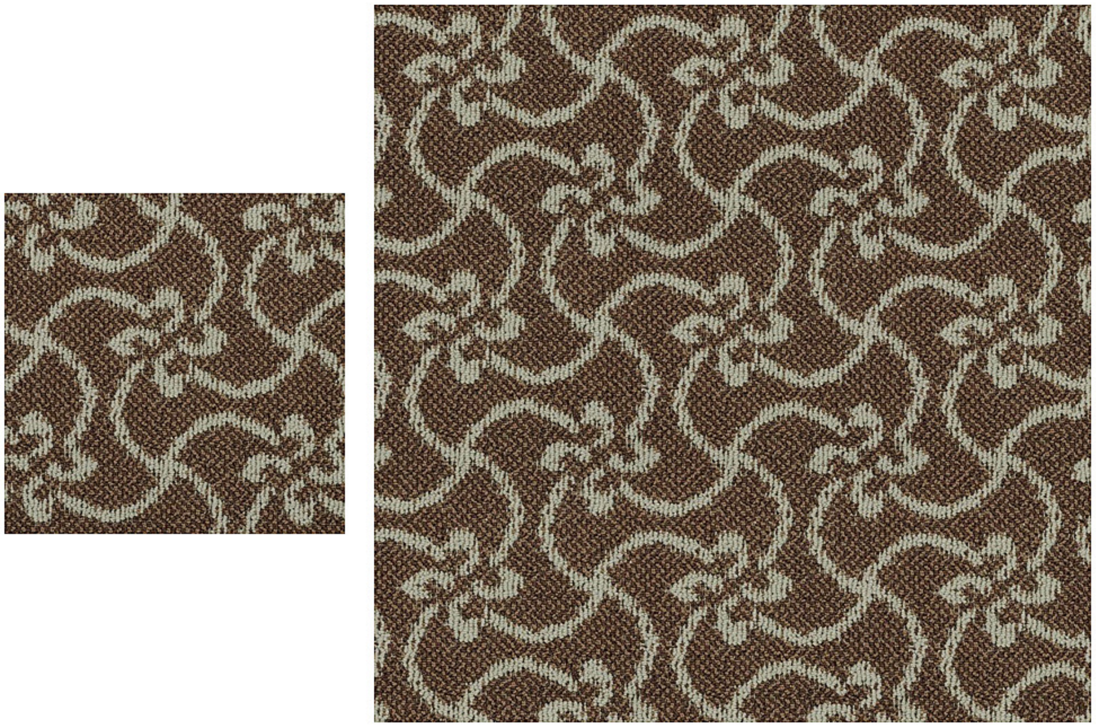

Fig. 2.3. Example of a successful texture synthesis result. Left is the input texture, right is the synthesized output.

2.2.1 Previous Work

Texture synthesis algorithms can be separated into two main categories: pixel-based methods and patch-based methods. Both methods copy pixels directly from the input texture sample to the output synthesized sample based on some defined condition. Efros and Leung [6] were the first to use a pixel-based technique. Their approach begins with a single seed pixel and grows the synthesized texture from that starting location. To synthesize a pixel their algorithm finds all neighborhoods from the sample texture which are similar, by some criteria, to the neighborhood in the synthesized texture with the pixel to be synthesized at the center. Once all candidate neighborhoods from the input texture are collected, one is chosen at random to prevent repetition in the output. The center pixel of the chosen neighborhood is then taken as the new synthesized pixel value in the output and the algorithm moves on to the next pixel location in a raster ordering. This process can be extremely slow, limiting the practical application of such methods. Wei and Levoy [7] address this issue of speed by extending the previous method by using tree-structured vector quantization to speed the process of searching for candidate neighborhoods. Their method reports output quality equal or better to previous techniques while running two orders of magnitude faster.

The second category, patch-based methods, include the most recent techniques developed. These methods find and copy an entire patch from the input image. The patches are placed into the output image and then the transition from one patch to another must be taken into account so as to hide the seams between patches. The way the patches are made to transition smoothly differs between approaches. In [8] the boundary artifacts are removed by blending the transition areas. They use feathering, or blurring, across the patch boundaries in order to create smooth transitions from one patch to the next. Efros and Freeman in [9] allow neighboring patches to slightly overlap, then compute a similarity metric for the overlap regions of two patches. Using this metric they construct a list of all patches which meet the criteria, then randomly select one patch to avoid repetition in the synthesized texture. They then perform a boundary cut which minimizes the error in the overlap regions of two patches, finding the best seam. Selected parameters for this method highly affect the output, as illustrated in Figure 2.4. The two previous methods both use regular and constant sized patches, generally square. In [10] the boundary cut is extended further. They use irregular patches without a constant patch size in order to find the optimal seam between patches. A graph cut approach is used to determine the optimal patch seam for any given region of the output texture.

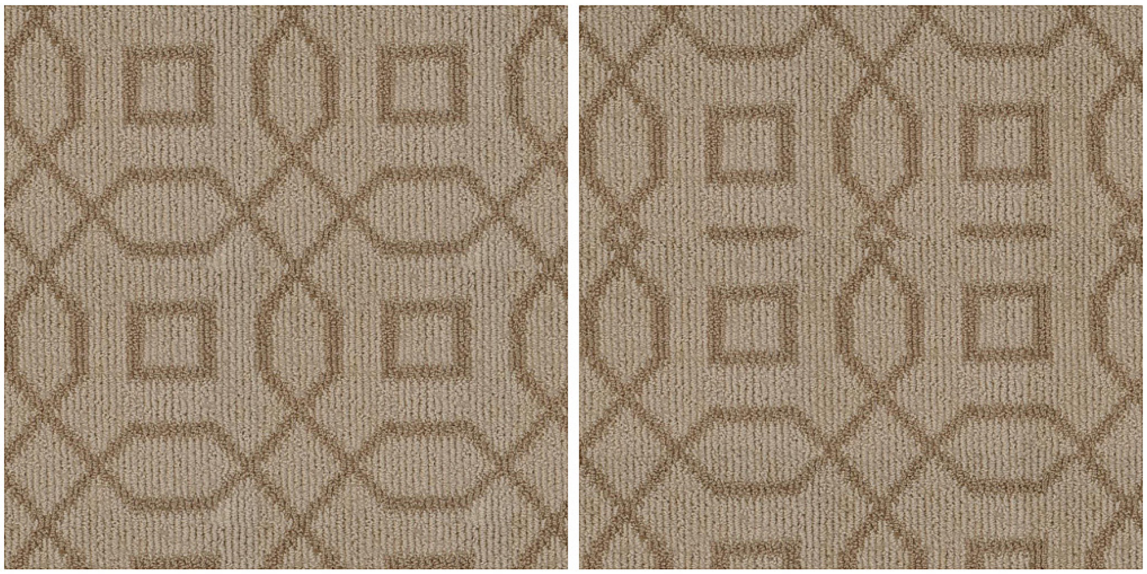

Fig. 2.4. Illustration of parameter effect on output using [9].

2.2.2 Our method

Some methods work better for certain types of textures while some textures are extremely difficult to synthesize regardless of the method used. However, these results come from a user working with the algorithms and understanding how parameters affect the output. Such a process is tedious, burdensome to untrained users, and difficult to adapt to real-life applications. With these considerations we chose to base our work on the image quilting method described in [9]. In this case the parameters include: size of the patch, overlap area for neighboring patches and error tolerance for the overlap regions. For someone familiar with the details of the texture synthesis algorithm being used parameter tuning is not a big issue, but for untrained users this can be a frustrating trial and error process. We propose a solution to this problem by first classifying the input texture into a set of predetermined texture types. Once the texture type is known, an initial estimate of parameters for the image quilting method can be set without any user input. This allows for texture synthesis to be carried out automatically in applications where a user does not have a technical understanding of the algorithm. We only require the user to provide feedback on whether or not the synthesized result is satisfactory. If necessary parameters are updated and another iteration of the texture synthesis algorithm may take place.

Texture Classification

We found that the texture synthesis parameters which produce the best results strongly depended on certain characteristics of the input texture. For example, if the input texture has a strong repeating pattern, the patch size must be on the order of the fundamental repeated element, visualized in the right image of Figure 2.8. If this is not done correctly, the structure of the pattern would not be maintained as seen in Figure 2.4. It was also noticed that in highly structured textures the error tolerance in the overlapping regions of patches needed to be very strict. Failure to do this would also result in the structure of the pattern not being kept. On the other end of the spectrum we found that if the input texture was very random and noise-like, the parameters needed to be more relaxed. The output for these types of textures depended on the error tolerance for the overlap regions being much lower. If this was not done the output would look repetitive and a clear difference from the input image would be noticeable. These observations led us to create a preprocessing step in the image quilting algorithm [9] in which we first identify the type of texture the input is. Once we know which type of texture the input is we can predict a good starting point for the image quilting parameters. Then, with user feedback, we can adjust the parameters accordingly until the synthesized output is satisfactory.

Texture Types

Textures can generally be classified into two types: stochastic and regular. However, most real-world textures fall somewhere between these two classes forming a spectrum of texture types [11]. In our preprocessing step we classify an input texture into one of three texture types: stochastic, irregular and regular. An example of each texture type is shown in Figure 2.5. These three classes were selected based on similar parameters needed during a trial and error process to produce the best texture synthesis results using image quilting. The application of our work is in the field of interior design, as such we can provide examples of each type of texture relating to interior design. Stochastic textures are random, noise-like textures, commonly found in carpets. Irregular textures fall between stochastic and regular texture types. Textures that fall under this category do not have a clear repeating structure, but also are not completely noise-like. Examples of this type of texture are marble, granite, or natural stones. Regular type textures have very strong repeating structures, like the square patterns shown in the right image of Figure 2.5. A single element of this repeating pattern, or the fundamental repeating pattern, can be identified and extracted. An example of the fundamental repeating pattern can be seen highlighted in green in the right image of Figure 2.8. Examples of this texture class are brick or tiling.

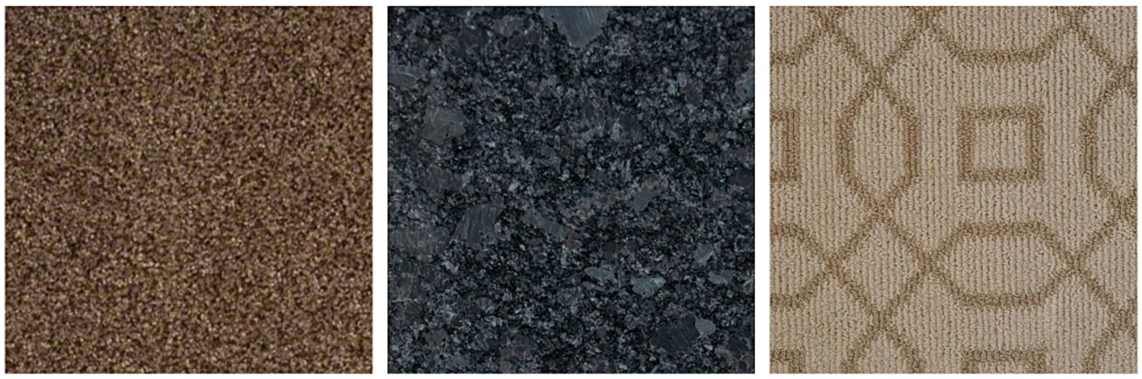

Fig. 2.5. Example texture classes. Left: Stochastic, Center: Irregular, Right: Regular.

Local Binary Patterns

To perform the texture classification we made use of local binary patterns [12]. A texture feature vector could be extracted from the input image by using the local binary pattern operator on neighborhoods within the input image, then constructing a histogram for the transformed image. This method was selected for computational simplicity. Generating a local binary pattern image is done by encoding each pixel based on the pixel's relation to neighboring pixels. For a simple explanation of local binary pattern encoding we will examine a 3x3 neighborhood from an image as shown in Figure 2.6. Each of the eight neighbors is compared to the center pixel. Every neighboring pixel is assigned to one bit in an 8-bit binary number. If the neighboring pixel is greater than the center pixel, that pixel position is assigned a value of one. If the neighboring pixel is less than the center pixel, a value of zero is assigned to that pixel position. Once all comparisons have been made the center pixel is assigned the integer value of the 8-bit codeword by concatenating the newly assigned binary values of the neighboring pixels. The order of the neighboring pixel values used when generating the codeword is arbitrary but must be kept consistent for all pixels. The neighborhood for performing the encoding can be modified in two ways. First, a circular neighborhood can be used where only a radius is defined. Since the points of this neighborhood will not lie on the rectangular grid of the image, pixel values for neighboring locations will be interpolated. Second, the number of neighbors does not need to be set to eight. A different amount of neighbors will change the number of possible encoded values in the output image.

Once this process is completed we have an image in which each pixel value is the local binary pattern encoding. An example of a local binary pattern encoded image can be seen in Figure 2.7. We now need to construct the texture feature vector from this image. The feature vector is constructed by concatenating histograms for non-overlapping windows within the local binary pattern image. For our 3x3 neighborhood example we selected a window size of 15x15 pixels following [12]. For each window

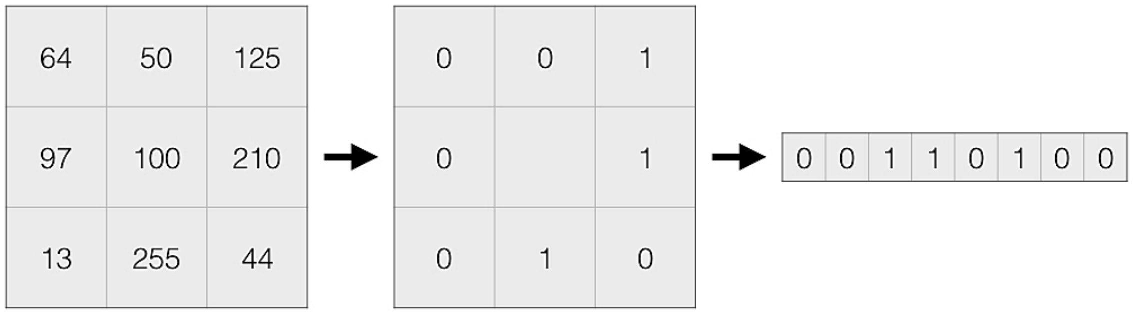

Fig. 2.6. Left: Example 3x3 pixel neighborhood, Center: Comparison of neighbors to center pixel, Right: Local binary pattern codeword.

we could construct a histogram of length 256. The histogram for each window would be concatenated to form the global feature vector for the texture image. This feature vector could then be used by any classification method.

Classifying

We use a straight-forward approach to classify textures as a proof of concept. In order to classify input texture images of unknown texture type, sample textures belonging to each class must first be collected. We used four samples from the stochastic class, ten samples from the irregular class and eight samples from the regular class. The local binary pattern feature vector was then computed for each training image. When the class of an unknown texture is desired, the same steps are followed to compute the feature vector of the unknown texture. Then the feature vector is compared with the training data to determine which class the new texture belongs to. In this paper we compute the chi-square distance between the feature vectors since chi-square distance can be used to measure distance between histograms. We then use a nearest neighbor approach and determine the class that the new texture belongs to as the one with the minimum average chi-square distance.

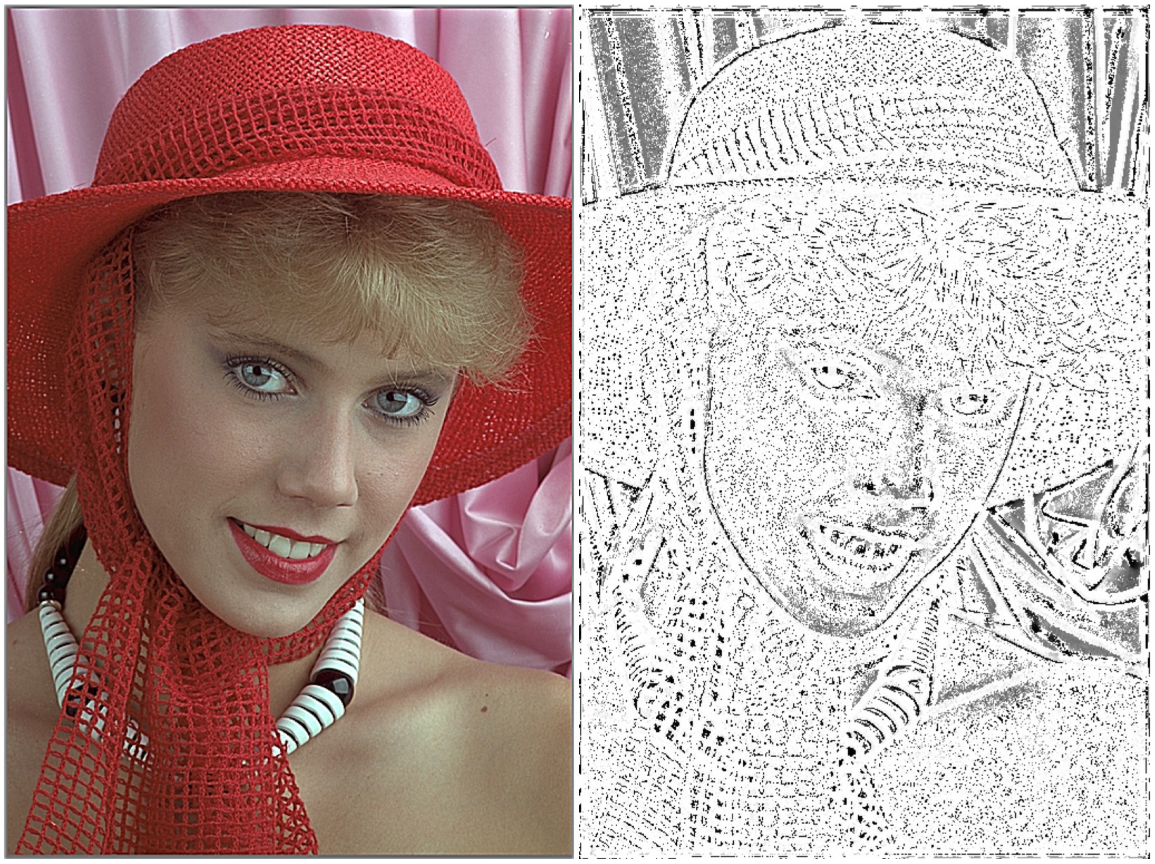

Fig. 2.7. Example of a local binary pattern encoded image.

User Feedback

The intent of this paper is provide a method which eliminates the need for the user to perform tedious parameter tuning while still producing high quality texture synthesis results. However, even the latest texture synthesis algorithms do not succeed in synthesizing every input texture, this issue is addressed further in Section 6. For this reason we still need to use some user information to steer our texture synthesis method towards a more appropriate set of parameters. In our method the user simply needs to respond to the question of a satisfactory output with a “yes” or “no” after the synthesis has taken place. If answered “yes”, the program will terminate, if “no”, the parameters from the previous run will be adjusted and synthesis will start over until the results satisfy the user.

Our method follows these steps to synthesize a texture:

1. Compute the local binary pattern image of the input texture.
2. Use the local binary pattern image to construct the texture feature vector for the input texture.
3. Classify the input texture as one of the three texture classes.
4. Set initial parameters for texture synthesis based on the texture class.
5. Run texture synthesis algorithm with defined parameters.
6. When synthesis is complete ask user if the result is satisfactory.
    (a) If yes, quit.
    (b) If no, update parameters and go to step 5.

Parameters

Based on the class of texture the input image has been identified, along with the user feedback, we were able to develop a strategy to set the parameters for the image quilting algorithm which produce the best results. Since there is variability in the best parameters even within a texture class, the user feedback will allow the parameters to be updated. Table 2.1 shows the initial set of parameters used once the texture class is identified. The patch size parameter is defined as a percentage of the input image size, the overlap is defined as a percentage of the patch size and the error tolerance is a raw difference in a selected error metric between patch overlap regions.

The user response to the output of the texture synthesis tells our method what to do next. If the user says that the output is not satisfactory the parameters need to be adjusted to try and correct the error. The overlap area still remains to be

Table 2.1.
Initial parameters by texture class

Class	**Patch Size**	**Overlap**	**Error Tol.**
Stochastic	0.5	0.05	0.1
Irregular	0.4	0.1	0.01
Regular	0.9	0.1	0.001

calculated as a percentage of the patch size. Error tolerance also does not need to be adjusted once the texture class is found. The only parameter that needs to be adjusted is the patch size. The adjustment is made as a percentage of the previous iteration's patch size. The new patch size is ninety percent of the previous patch size for each class. It was found that this parameter plays the most important role in the synthesis results. For regular textures the patch size needs to be on the order of the fundamental repeating structure or the pattern will not be kept. An error in a repeated regular pattern is easily perceived by humans. Generally, the patch size must be large enough to capture the structures within the input texture, but small enough that the output image does not look too similar to the input texture. The output must look like a different image generated by the same process. To achieve this our method starts with an over estimation of the patch size and slowly decreases the size until the output is satisfactory. Figure 2.8 demonstrates this procedure. Each box represents the patch size for one round of texture synthesis. When the patch size is set to the size of the green box it is on the order of the fundamental repeating structure, as demonstrated by the right image, and the synthesized texture is accepted. After each round the patch size is reduced to ninety percent of the previous size regardless of which class the input texture belongs to.

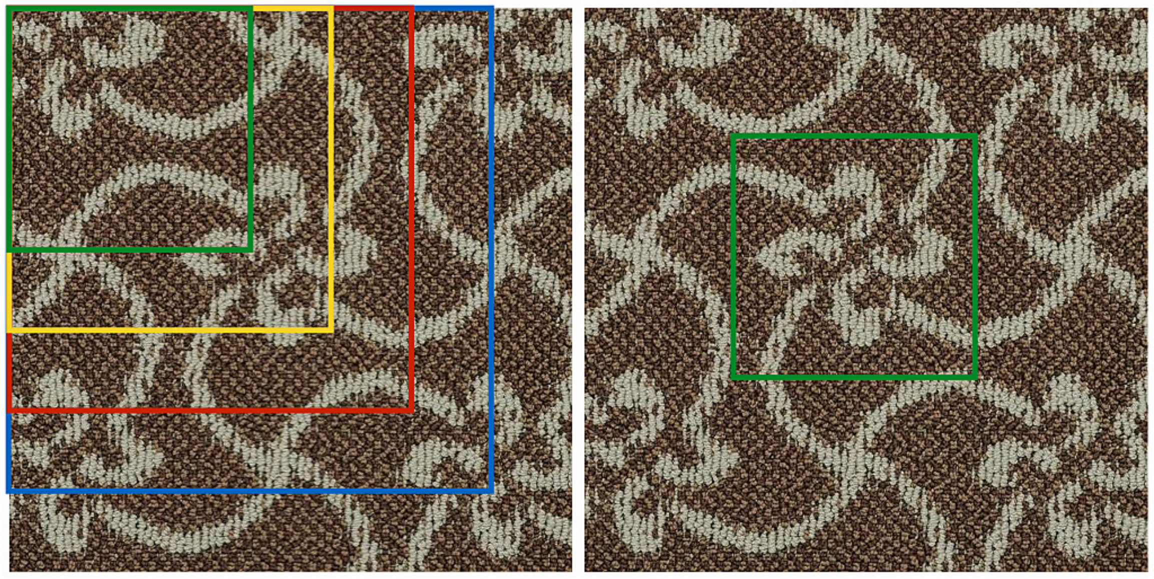

Fig. 2.8. Illustration of patch size progression through user feedback. Left: Each box represents the patch size for each iteration of texture synthesis, Right: Green box shows appropriate patch size on the order of the fundamental repeating structure.

2.3 Results

To test our method we used real world textures of varying texture types. The textures were selected to represent a range of those that were commonly found in the interior design industry such as flooring and counter tops. No parameters for the image quilting method were needed as input from the user. The only user interaction was the feedback described in previous sections.

Our classification results are satisfactory, though only tested on a small sample size. We used twenty-two training images and received eighty percent accuracy when testing on images containing similar characteristics to the training set. As the sample size increases, intra-class variation may cause an issue using a nearest neighbor approach as discussed in this paper. In that case another classification method may be investigated.

The results of our method are shown in Figure 2.9. For stochastic type textures, (Figure 2.9 (a), (b), (c), (d), (e) and (f)), our method works very well. This is because a wide range of parameters would yield pleasing results. Any perceptual differences in the synthesized output for different parameters are difficult to detect. Structural textures with a strong repeating pattern are also successfully synthesized using our method. Unlike stochastic textures, structural textures are very sensitive to the parameters. This causes our methods to require approximately 3-5 iterations of user feedback in order to correctly synthesize most structural textures. Results of structural textures are shown in the third image in Figure 2.9 (j), (k), and (l).

The texture class that is challenging for the proposed method is the irregular texture. There are two reasons why these texture types are difficult to properly synthesize. The first is that they do not have a fundamental repeating pattern but they do contain some structure which can be easily seen when synthesized incorrectly. Taking pixels or patches from the sample texture image limits the available data used to synthesize, creating the possibility of leaving out some structural characteristics which may be obvious to a human viewer. This can be seen in Figure 2.10 (a). The stones clearly have some structure, but not a well defined structure. In the synthesized image it is obvious that mistakes have been made when pebbles of different color and texture are attempted to be matched in an overlapping region. The second reason is that most real world irregular textures have multiple layers of textures within them. For example Figure 2.10 (b) is an irregular texture of stones, but within each stone there exists a stochastic type texture. Our method will try to synthesize the irregular structure as best it can, but in certain cases such as the one in the figure mistakes are obvious. This is not true in all cases. Figure 2.9 (g), (h), and (i) show cases where irregular textures are able to be correctly synthesized by our method. We believe that in order to overcome the challenge of irregular texture synthesis, methods other than patch based algorithms must be developed. The use of patches limits the total information available to that which exists in the input image. Model based methods have the possibility to produce better results for irregular textures. One would need to extract an accurate model for the generation of the texture given in the input image, then synthesize a new image according to the model for the output.

2.4 Future Work

Improvements can be made towards a more fully automatic method for texture synthesis. Our method uses a basic way of classifying textures into one of three texture types and then synthesizes directly from there. More sophisticated techniques could be used to improve upon the classification problem. This problem can also be regarded as a regression problem instead of a classification problem. In other words an attempt could be made to use features from the input texture sample to directly estimate the most likely parameter values to be used during synthesis rather than first classifying a texture type and then setting parameters. For both of these improvements recent methods using deep learning come to mind.

The issue of irregular texture types resulting in unsatisfactory synthesis results is also something which needs to be addressed. The most pressing texture type relating to interior design where this comes up is with wood patterns. Patch based methods fail drastically for these textures, as seen in Figure 2.11. This is because there is a clear global structure to the pattern and often a direction, however examples of how to continue this pattern are often missing from the input sample. Synthesis algorithms will do their best to match parts of the patterns based on local structure, but they have no idea how to maintain a global structure. For this reason another synthesis method would have to be used entirely.

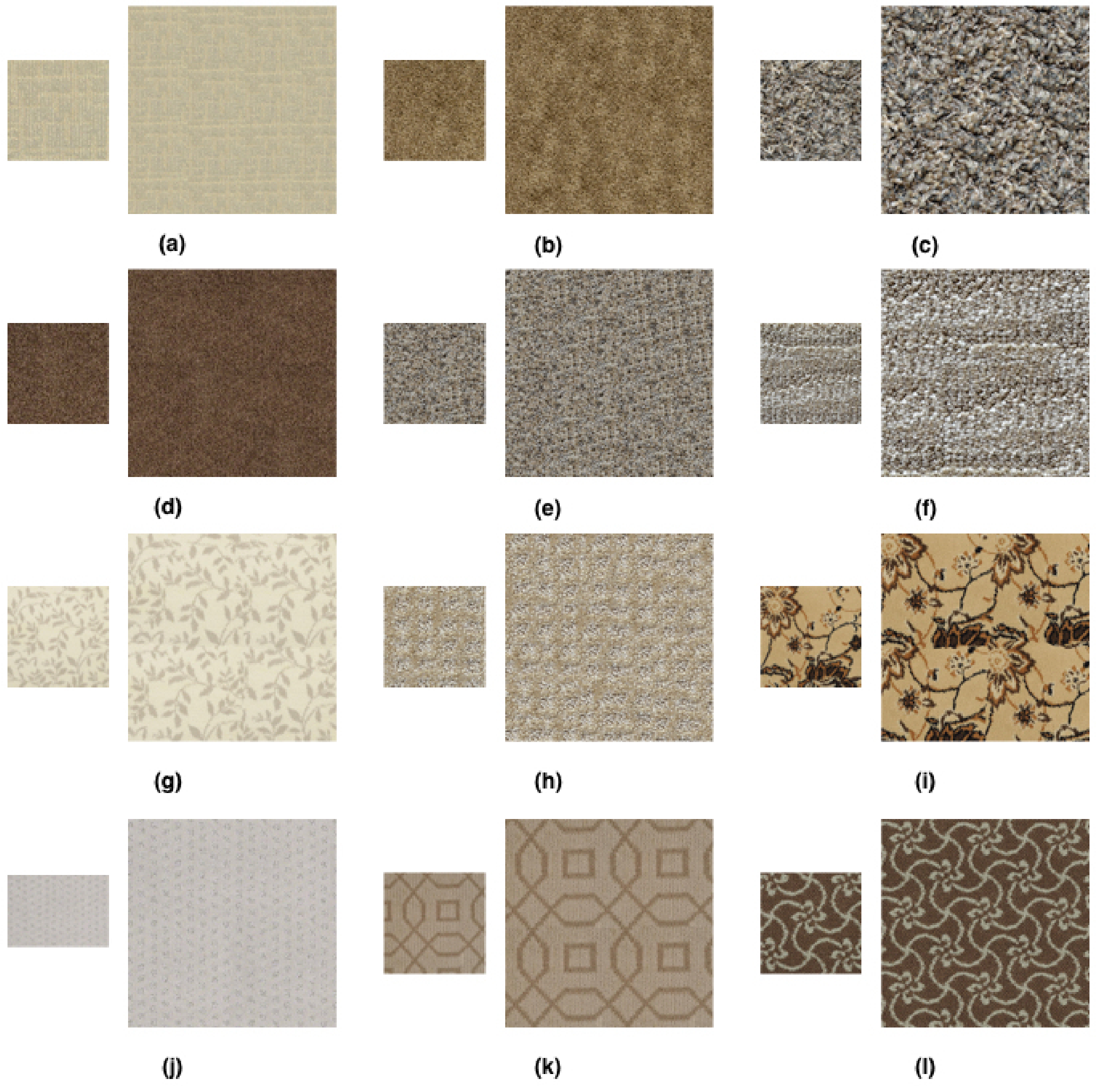

Fig. 2.9. Experimental results: Examples of successful synthesis results. The smaller image is the input sample and the larger image is the synthesized output. Output synthesized at twice the input size.

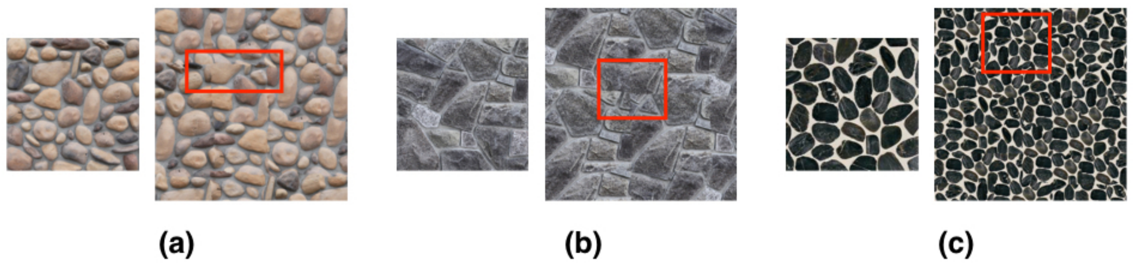

Fig. 2.10. Experimental results: Examples of failed synthesis attempts. The smaller image is the input sample and the larger image is the synthesized output. Output synthesized at twice the input size. The red boxes highlight errors in the synthesized output.

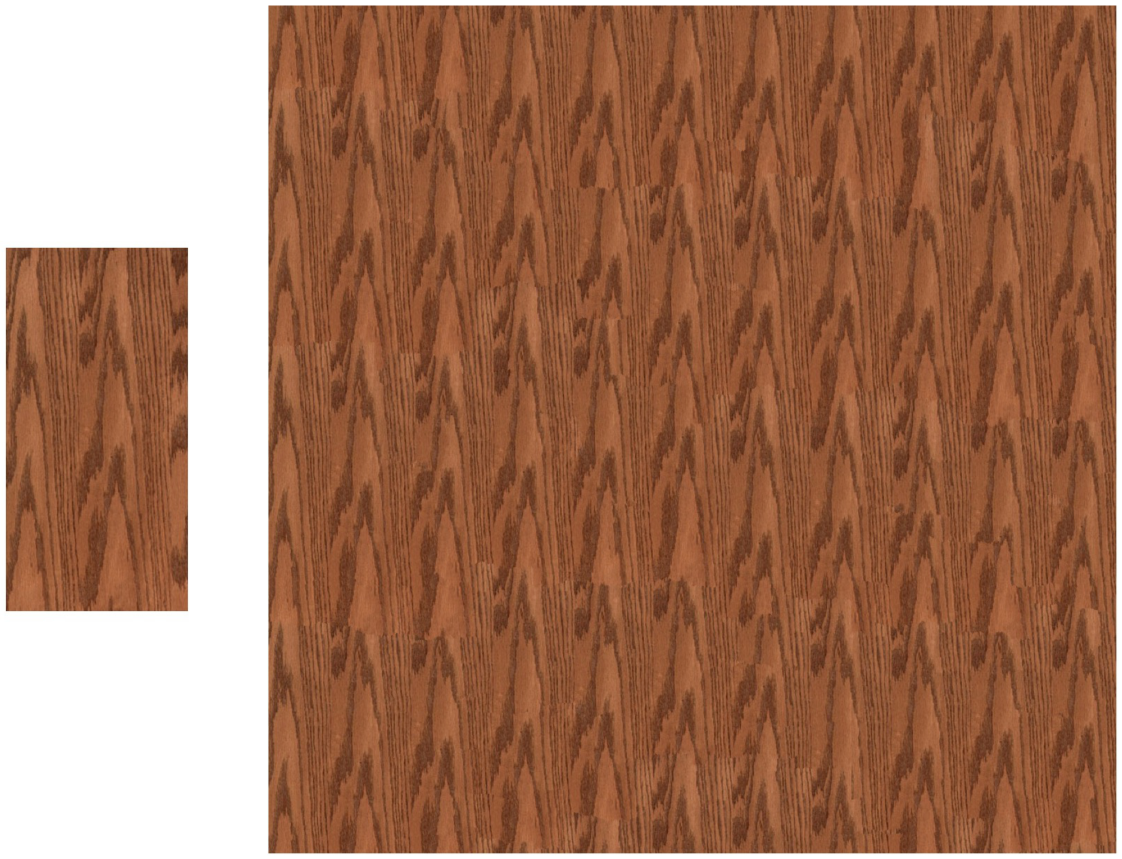

Fig. 2.11. Illustration of the poor results common for wood textures.

3. IMAGE RE-RENDERING

Now we must explain how to take the newly synthesized texture and insert it onto the desired surface in an image. The goal is to maintain realism so there are multiple factors which will contribute to a satisfactory result. Since our target application will allow non-technical users to manipulate images we only want to use low dynamic-range (LDR) images captured from typical cameras such as from a smart phone. These images have a limited exposure range resulting in a loss of detail in the highlight and shadow regions.

3.1 Perception

Exact scene parameter estimations are not needed to produce visually pleasing rendering results. Many image synthesis methods leverage the limitations of the human visual system in order to reduce computation while retaining image fidelity. These methods incorporate metrics to determine when an approximate solution for parameters such as illumination, geometry and reflectance will be visually indistinguishable from a reference solution [13]. It has also been shown that estimates for point light sources can still create visually similar renderings [14]. The coarse estimates which are described in the next sections will be sufficient for creating realistic renderings.

3.2 Scene Reconstruction

To make physically grounded edits to an image we must first infer the physical scene corresponding to a single low dynamic-range image. The physical scene consists of three main components which contribute to the image formation process: geometry, illumination and surface reflectance. Though the desired results can be made using tools such as Photoshop, it takes someone with artistry and expertise. To make this work accessible to the average user these properties should be obtained automatically with minimal user input. We assume the user have no understanding of the imaging process. We closely follow the work in [15] where they aim to insert a new 3D object into a scene. The difference between inserting an object and replacing a surface seems minimal, however the consequences are bigger than expected. This is explained in more detail in Section 4.

Creating realistic re-renderings of images with one or more surfaces replaced with new materials requires basic information about the 3D scene. This information includes: spatial layout or relative depth of objects in the scene, material properties such as reflectance and scene lighting. Why are these scene properties integral in creating realistic modifications? One perceptual cue for the human perception of depth is occlusion [16]. One object overlapping another implies relative positions. To create a sense of realism the re-rendered result must be consistent, with respect to occlusion, with the original image. Therefore relative depth of objects must be extracted. Another important factor in the perception of photo-realistic rendering is inter-reflection between objects. Reflections between objects must agree with perceived spatial layout of the scene. Incorrect, inconsistent or missing reflections provide immediate clues that the image was artificially generated. To properly maintain reflections in a scene reflectance properties must be found both for existing scene objects and new materials to be placed in the scene. Lastly, the scene lighting must be accounted for. The shadows seen in an image provide depth cues. These cues provide information about the volume of space the objects are in and relative positions of the object [17]. Again, incorrect, inconsistent or missing shadows or other lighting effects will cause obvious errors in the attempted realistic re-rendering. We automatically produce coarse estimates of this scene level information to reconstruc a 3D scene from a 2D image. A high level system diagram can be seen in Figure 3.1.

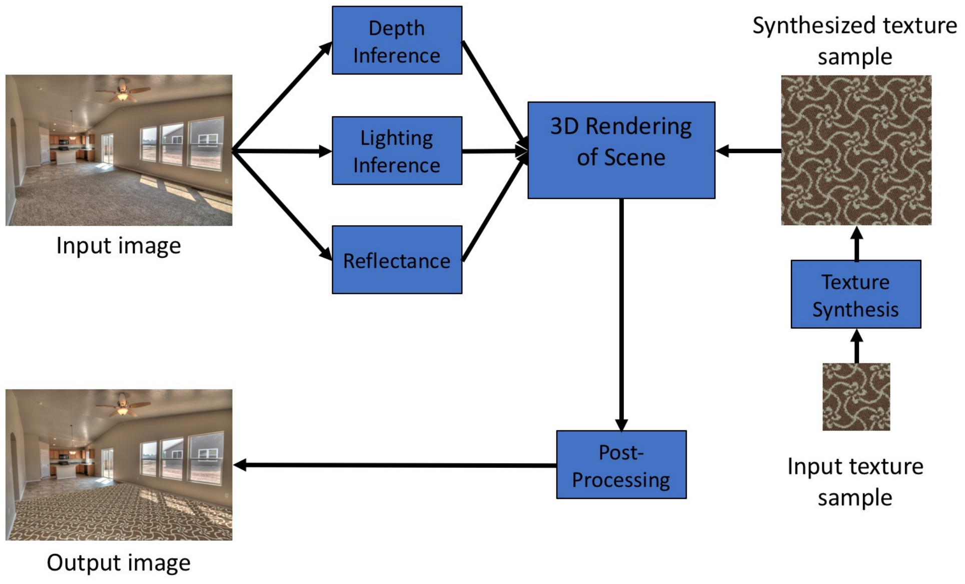

Fig. 3.1. High level system diagram.

3.2.1 Depth

When we look at an image our prior information about the world allows us to infer depth. In sticking with our user-friendly approach to image editing, it is undesirable to require stereoscopic images for depth estimation. The accuracy of stereo vision is limited by the baseline distance between the two cameras, making using user generated stereoscopic images for depth estimation undesirable. We also do not want to require a user to do annotation which may require them to be trained. We are targeting an untrained user. With this in mind we follow the data-driven approach [18] with the changes detailed in [15] for automatic depth estimation.

Many single image reconstruction techniques have been proposed. In [19] they created 3D scene reconstructions from single 2D images by assuming each image could be broken down into coarse categories: “ground”, “sky” and “vertical”. A statistical approach was used to label pixels with one of these categories. These labels could then be used to morph the image into a pop-up model of the scene. In [20] they develop a Bayesian framework which is trained to recognize floor-wall boundaries throughout an image allowing for 3D reconstruction from a single image. A Markov Random Field, trained via supervised learning, is used to infer 3D location and orientation of image patches in [21]. The MRF models both depth cues as well as relationships between different parts of the image while only assuming that the environment in the image is made up of multiple small planes. A data-driven approach is used in [18] by matching scene level features and warping depth images from a groundtruth dataset to align features. Optimization is then done to constrain the output depth to some assumptions and prior conditions.

Following [15] our current geometry is in the form of a per-pixel depth map. This map is obtained by combining a data-driven model, one with no explicit parameters but instead uses a database of images, and geometric information present in many images. The database used is the NYU Depth Dataset V2 [22].

First, given a database of RGBD images candidate images are found from the database which are “similar” to the input image. This is done by using GIST [23] features for high-level context based scene level recognition. The appeal of using GIST is that there is no processing of individual objects. An image is broken down into a set of low dimensional perceptual dimensions including: naturalness, openness, roughness, expansion and ruggedness. Using these dimensions the top K(=10) candidates are found from the depth dataset using a nearest neighbor approach. The candidate depth maps are then used in an optimization process to achieve the final depth map.

A pixel-to-pixel correspondence is then found between the input image and each of the candidate images using SIFT flow [24]. SIFT flow matches per-pixel SIFT features to estimate warping functions, $\psi_i, i \in \{1, ..., K\}$, which map pixel locations from a candidate's domain to pixel locations in the input image's domain. These warping functions are then used to also warp the depth map for each candidate. An optimization process takes place where the warped candidate depth maps contribute

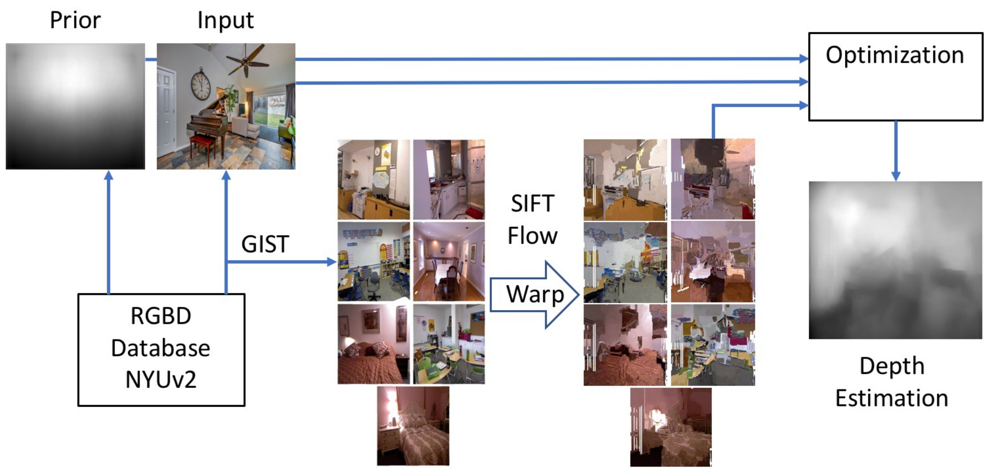

Fig. 3.2. Flow chart of depth estimation process.

to one of the terms in the objective function, the data term. Figure 3.2 shows a diagram for the depth estimation process. The optimization procedure aims to minimize the following function:

$$\sum_{i \in pixels} E_t(D_i) + \lambda_m E_m(N(D)) + \lambda_o E_o(N(D)) + \lambda_{3s} E_{3s}(N(D)) \tag{3.1}$$

The terms in the objective function are as follows: $E_t(D_i)$ is the data term, using the database of RGBD images, $E_m(N(D))$ is a Manhattan prior term, this is used to ensure patches of a scene are always oriented along one of the three dominant directions and so encourages parallel and perpendicular surface normals from the estimated depth, $E_o(N(D))$ is a surface orientation constraint that forces surface normals to coincide with a surface orientation estimate computed using [25], and $E_{3s}(N(D))$ is a 3D smoothness term which encourages nearby surface normals to point in the same direction unless there is a strong edge in the original image.

Data term

The data term is used to require the estimated depth to be similar to the warped candidate depths as in [18]. The distance used to measure similarity between the current estimated depth at pixel i (D_i) and warped candidate depth j, $\psi_j(C_i^{(j)}) : j \in \{1, ..., K\}$, is defined by $\phi(x) = \sqrt{x^2 + \epsilon}$. The data term is then defined as:

$$\sum_{j=1}^{K} \omega_i^{(j)} [\phi(D_i - \psi_j(C_i^{(j)})) + \gamma[\phi(\nabla_x D_i - \psi_j(\nabla_x C_i^{(j)})) + \phi(\nabla_y D_i - \psi_j(\nabla_y C_i^{(j)}))]] \quad (3.2)$$

Where $\omega_i^{(j)}$ is a confidence measure of the accuracy of the j^{th} candidate's warped depth at pixel i. This term encourages similarity not only in depth, but also in depth gradients in both the x and y directions.

Manhattan Prior

Under a Manhattan world prior patches of a scene should be oriented along one of the three dominant directions. The directions are defined by $R = (R_x, R_y, R_z)^T$ which is the rotation matrix which will rotate identity to the set of vanishing points. This term adds a penalty for surface normals ($N(D)$) not lying parallel or perpendicular to one of these directions.

$$pp(N, V) = \frac{1}{2} - ||N^T V| - \frac{1}{2}| \quad (3.3)$$

$$E_m(N(D)) = \sum_{i \in pixels} pp(N_i, R_x) + pp(N_i, R_y) + pp(N_i, R_z) \quad (3.4)$$

Where the *pp* function is small if input vectors are parallel or perpendicular and large otherwise.

Surface Orientation

Using the estimate of surface orientations found by using [25] this term enforces surface normals to be consistent with this estimate where the confidence of the estimate is high. Letting Ω be the set of all pixels where the confidence is high and O_i^{map} be the predicted surface orientation at pixel i, the surface orientation term is then:

$$\sum_{i \in \Omega} 1 - |N_i^T O_i^m ap| \tag{3.5}$$

This term will be small when the vectors are aligned and large otherwise.

3D Smoothness

Finally, the smoothness term enforces normal vectors to be pointing in the same direction unless there is a strong edge in the original image. This minimizes discontinuities from the warping functions applied to the candidate depth maps.

$$\sum_{i \in pixels} s_i^x ||\nabla_x N_i|| + s_i^y ||\nabla_y N_i|| \tag{3.6}$$

Where $s^x = (1+e^{||\nabla_x I|| - 0.05)/0.01)})^{-1}$ and $s^y = (1+e^{||\nabla_y I|| - 0.05)/0.01)})^{-1}$ are threshold functions for the input image I's derivatives.

This setup is an unconstrained, non-linear optimization. The method of iteratively re-weighted least squares is used to minimize the objective function. An example input image along with it's estimated depth map can be seen in Figure 3.3.

3.2.2 Lighting

Lighting plays one of the most important roles in creating photo-realistic renderings. Inconsistent shadows or highlights are fairly easy for humans to perceive. Lack of shadows and highlights can give a rendering a cartoon-like appearance. Estimating

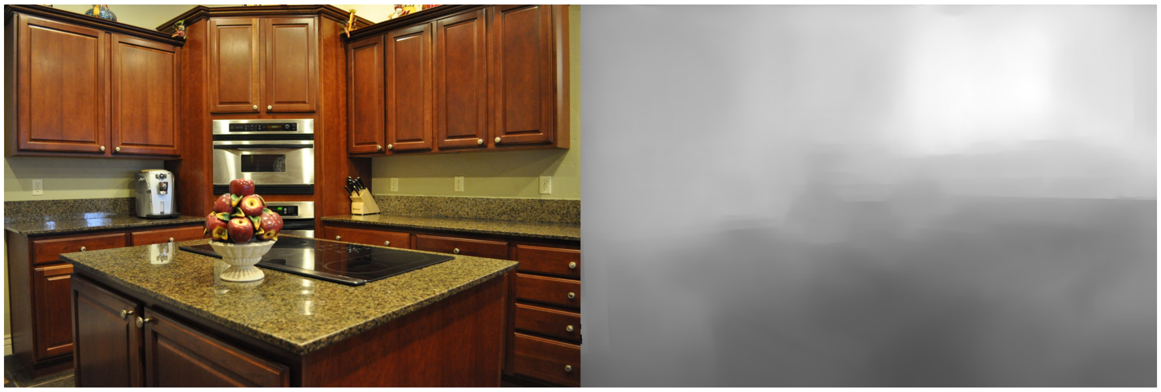

Fig. 3.3. Example of estimated depth map (right) from original image (left).

scene illumination is generally difficult and under-constrained given that illumination in real environments is often complicated.

Various illumination representations exist. Individual locations of light sources can be represented by point lights. Global scene illumination can be represented in the form of an environment map or image based lights (IBL) [26]. An IBL is typically captured by taking omnidirectional images at the physical scene. Some methods for taking omnidirectional images include: using a mirrored ball, fisheye lens images, panoramic cameras and stitching images together. An example of an IBL taken with a mirrored ball can be seen in Figure 3.4, while an example taken with a panoramic camera can be seen in Figure 3.5.

Point light sources have been detected by requiring a user to identify the silhouette of an object in the image and then using the contour and gradient information to infer light directions and relative intensities [14]. [27] show how reflections in eyes can be used to create an environment map. Environment maps have also been estimated using automatically detected shadows [28], but require depth information as input. [29] showed that wrapping an image to create an environment map can also be sufficient in certain applications. Again, following [15], we predict illumination both in view and out of view of the camera using a data-driven method.

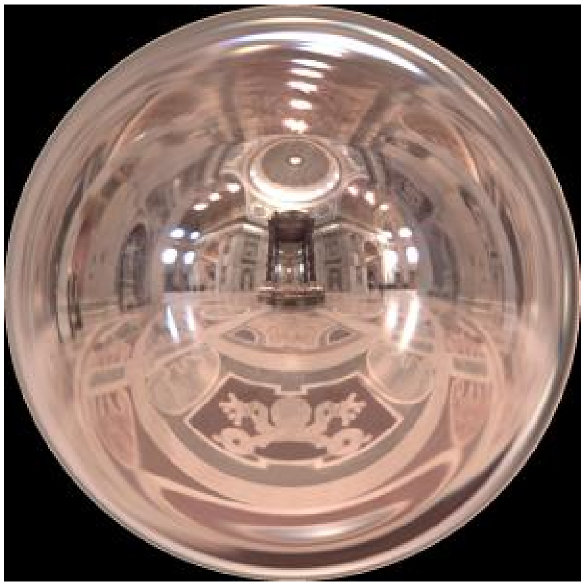

Fig. 3.4. Example of an environment map captured with a 360 degree panoramic camera, a similar image could generated using a mirrored ball.

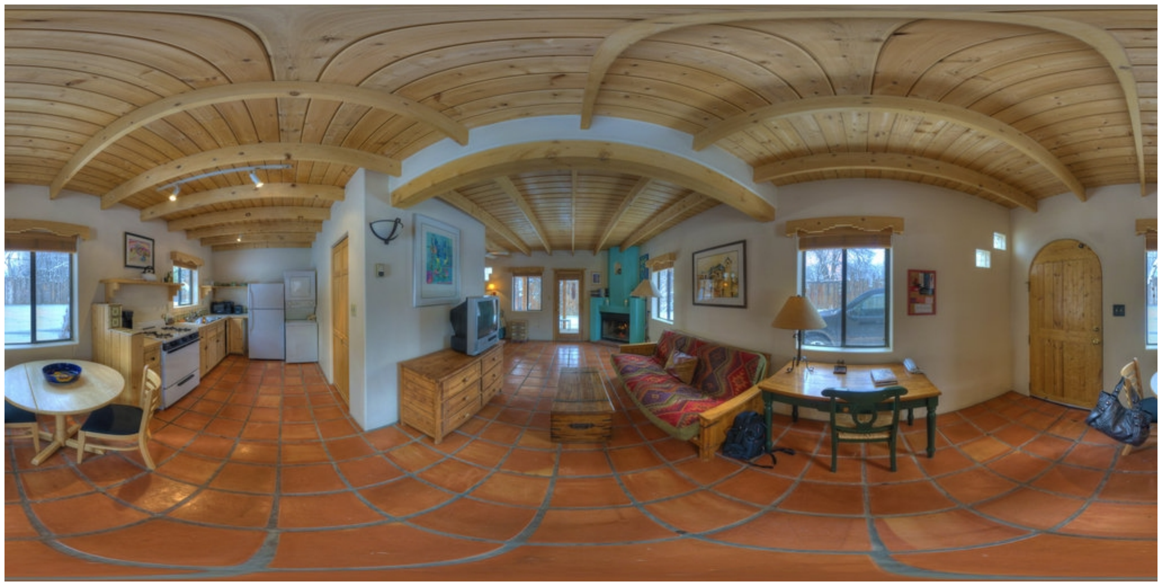

Fig. 3.5. Example of an environment map captured with a 360 degree panoramic camera, a similar image could generated using a mirrored ball.

To characterize scene illumination fully light sources must be split into two categories: in-view, those actually seen in the image, and out-of view, the illumination environment which contributes to the scene from behind the camera's field-of-view. Different approaches are used to identify each type of light source for the input image. In-view sources are found by using a support vector machine (SVM) performing a binary classification. Out-of-view illumination is estimated using a data driven model with a similar idea used in the depth estimation [15]. Annotated panoramic images from the SUN360 dataset [30] are used as training data for both in-view and out-of-view source estimation.

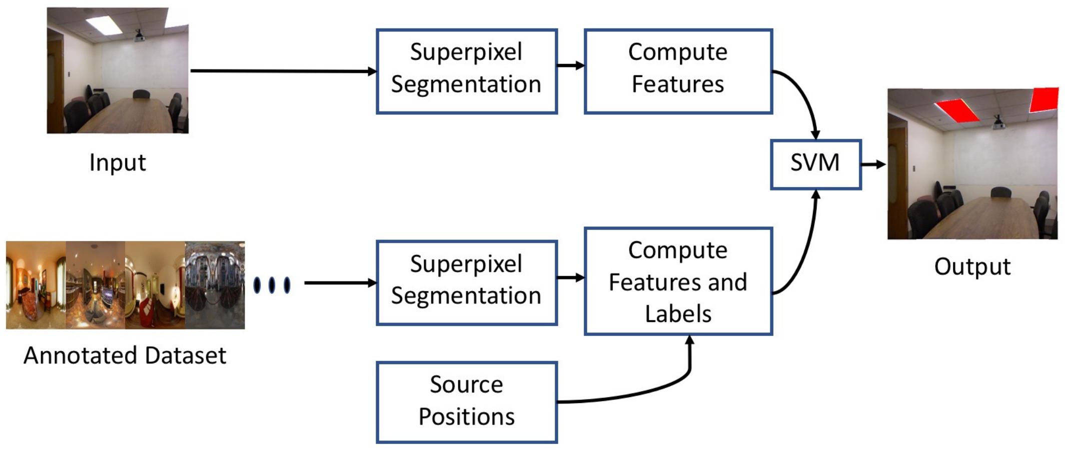

Fig. 3.6. Flow chart for in-view light source estimation.

In-View Source Estimation

To detect in-view light sources the input image is first oversegmented into superpixels using SLIC [31]. Then the following features are computed for each superpixel: height of the superpixel in the image, computed by averaging the height of all the pixels in each superpixel, along with features used in Make3D [32]. The Make3D features are used to determine the 3D structure of an image as modeled by a number of small planes. These features aim to capture monocular depth cues as well as meaningful image boundaries such as occlusions or folds. The features are the output of 17 filters computed using a 3x3 mask on each superpixel. The 17 filters include the 9 Laws masks applied to the Y channel of the image in YCbCr space, the first Laws mask is also applied to the Cb and Cr channels, and 6 oriented edges. The Laws masks are generated by multiplying combinations of the following vectors: $L3 = (1, 2, 1), E3 = (-1, 0, 1)$, and $S3 = (-1, 2, -1)$. The visual representation of these masks can be seen in Figure 3.7. The oriented edge detectors are spaced out at 30 degrees apart, shown in Figure 3.8. The 15 masks are applied to the Y channel of the image, while the L3L3 Laws mask is also applied to the Cr and Cb channels. This makes 17 filter responses for each superpixel. For each of the filter responses both the energy and the kurtosis is calculated, $\sum_{(x,y)\in S_i} |I(x, y) * F(x, y)|^k$ where $k = 2, 4$ gives the energy and kurtosis respectively. This results in 34 features for each superpixel. The four neighboring superpixel features are concatenated to the current superpixel feature making $5x34 = 170$ features for each superpixel at a particular scale. This process is then carried out at 100% and 50% scale, making the final feature count 340. A diagram of the in-view classifier can be seen in Figure 3.6.

Using the annotated training data a binary classifier can be trained to predict whether or not a superpixel in the input image contains a light source. An example output of the in-view light source classifier can be seen in Figure 3.9.

Every source superpixel in the input image must now be matched to a 3D location in the scene. Using our previous depth estimation we can find the source's 3D position by:

$$\mathbf{X} = D(x, y)K^{-1}[x, y, 1]^T \tag{3.7}$$

Where K is the camera projection matrix and D is the depth found previously.

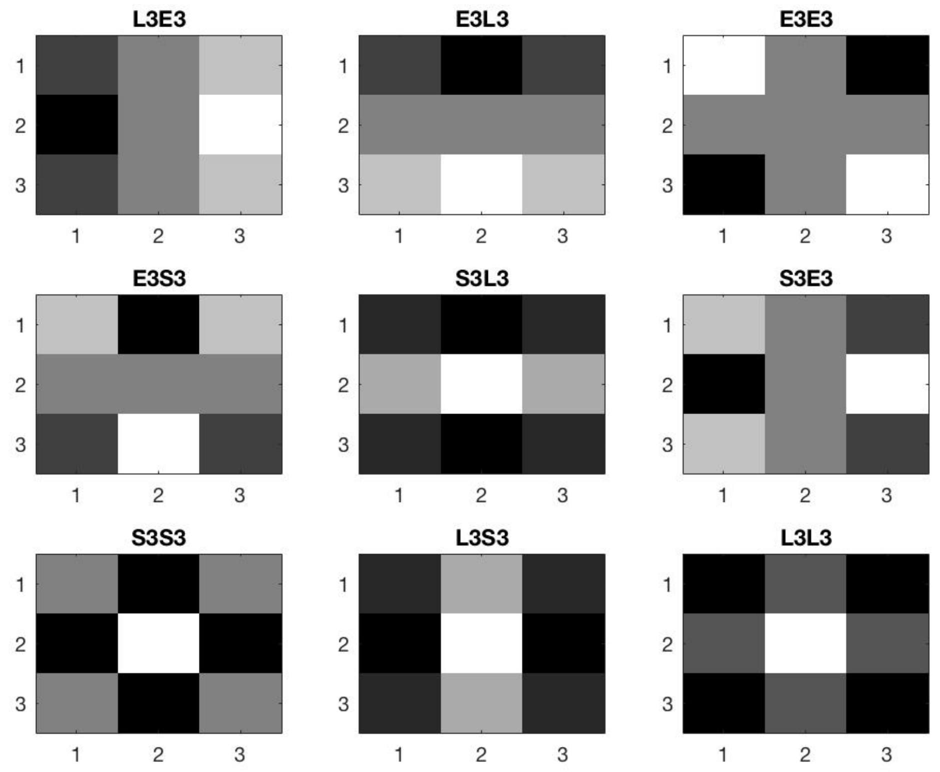

Fig. 3.7. The 9 3x3 Laws masks used as part of the Make3D features. Vector combinations labeled above each image.

Out-of-View Light Source Estimation

Out-of-view light source estimation is the most difficult part as the correct estimate is unclear. Following [15] we assume that if two images have a similar appearance, then the illumination behind the camera should also be similar. With this assumption we again use a data-driven approach with the SUN360 panorama dataset [30]. Each IBL from the dataset is sampled into N(=10) rectilinear images at varying points and fields-of-view on the sphere. These projections are used as the ground truth data, since we know the corresponding illumination is the parent panoramic image. The sampled images are matched to the input image using a similarity feature.

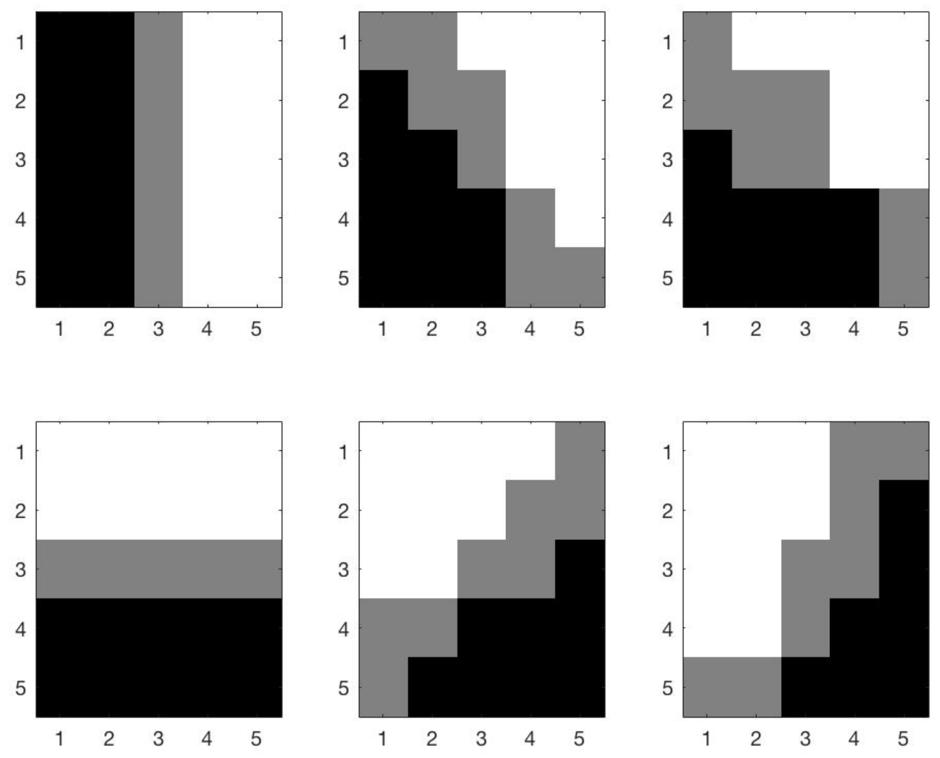

Fig. 3.8. Filter masks for oriented edge filters spaced at 30 degrees apart.

We want to rank pairs of images so the final feature used describes how two images match in feature space. There are a total of six features computed for each image: orientation maps [25], spatial pyramids [33], HSV histograms (three), and the output of the in-view light classifier. For the orientation map and the light classifier a normalized dot product is used to determine the similarity in feature space for the final feature. For the HSV histograms and the spatial pyramid a histogram intersection score is used to compare the two image features. This leaves us with our final 6D similarity feature vector.

We want to be able to discriminate between different IBLs. To do this we need to define a distance metric to tell us how similar one IBL is to another. This distance

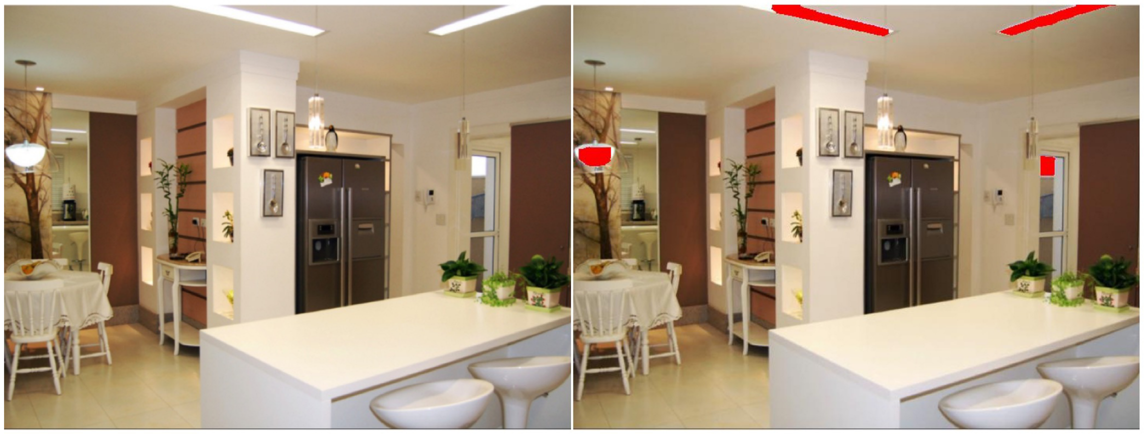

Fig. 3.9. Example output of the in-view light classifier. Left is the original image while right is the output with detected sources marked in red.

metric will be used as the loss for our rank training optimization. Our objective is to use the IBL to light a scene during the re-rendering so a pixel-wise comparison does not make sense. Instead we will define our metric based on how a set of standard objects look when they are illuminated by these IBLs. We will render three standard objects with varying materials into each of the environments defined by the panoramic image. The distance between the IBLs will then be the mean L2 error over the pixels belonging to the objects in the renderings for all object and material combinations. Equation 3.8 is the distance metric where Ω is the set of all objects rendered into the panoramic image environments P_i and P_j, and I_i and I_j are the two rendered images coming from IBL i and j respectively. An example rendering of one of these objects in two different panoramic environments can be seen in Figure 3.10.

$$d(P_i, P_j) = mean_{o \in \Omega} ||I_{i,o} - I_{j,o}||_2 \tag{3.8}$$

The goal is to find a ranking function ω such that $\omega^T x_j^i > \omega^T x_k^i$ whenever the illumination in panoramic image j is better suited for input image i than the illumination in panoramic image k. Here x_j^i and x_k^i are the features between the input

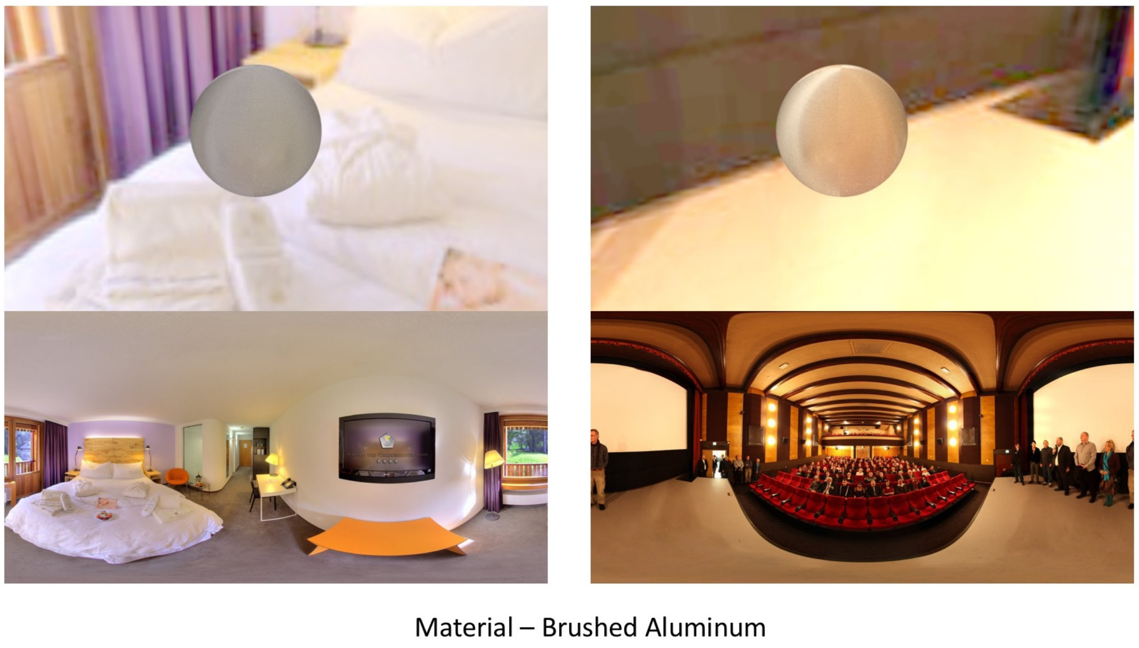

Fig. 3.10. Example of objects rendered into panoramic image environments, material used is brushed aluminum. The top image is the rendered object, the bottom image is the environment map used during rendering.

image and sampled rectilinear images from the respective panoramic images from the dataset. This function is found using a standard 1-slack, linear SVM-ranking optimization [34]. The objective function and constraints are detailed in Equation 3.9. $\delta^i_{j,k} = max(d(P_i, P_k) - d(P_i, P_j), 0)$ is a hinge loss.

$$argmin_{\omega,\epsilon}||\omega||^2 + C\epsilon, s.t.\omega^T x^i_j < \omega^T x^i_k + \delta^i_{j,k} - \epsilon, \epsilon > 0 \tag{3.9}$$

To predict the top ranked illumination for a new input image i we first compute the similarity feature vector pairwise for all sampled images in our dataset, $x^i_j, \forall j$. Since we have found our ranking function as detailed above we just need to sort the output of that function for all the computed features, $\omega^T x^i_j$, in decreasing order. Then we can select the top k IBLs to use when rendering. In our work we only select the number one ranked IBL.

3.2.3 Reflectance

The observed color at any point on an object is influenced by many factors. Barrow and Tenenbaum proposed breaking an image into separate intrinsic scene property images [35]. These include: range, orientation, reflectance and illumination. Many algorithms focus on a separation of images into three components: illumination, reflectance and specular [36]. The illumination component comes from shading effects such as object occlusion and lighting. The reflectance component, also referred to as albedo, represents how the material of an object will reflect incident light independent of viewpoint and illumination. The specular component corresponds to highlights which come specifically from viewpoint, illumination or geometry. Example intrinsic images can be seen in Figure 3.11. We can then express the incident light intensity at an point in an image by the following:

$$I(x) = S(x)R(x) + C(x), \tag{3.10}$$

Where $I(x)$ is incident light at pixel location x, $S(x)$ is illumination, $R(x)$ is reflectance and $C(x)$ is the specular component.

Retinex theory was proposed in [37] showing that albedos could be separated from illumination when illumination was assumed to vary slowly. Small gradients are assumed to correspond to illumination and large gradients are assumed to correspond to reflectance. Different heuristics have been devised based on real world assumptions for classifying edges as illumination or reflectance.

The intrinsic image is estimated based on the following:

1. Compute the horizontal and vertical gradients, i_x and i_x, of the log (grayscale) input image.

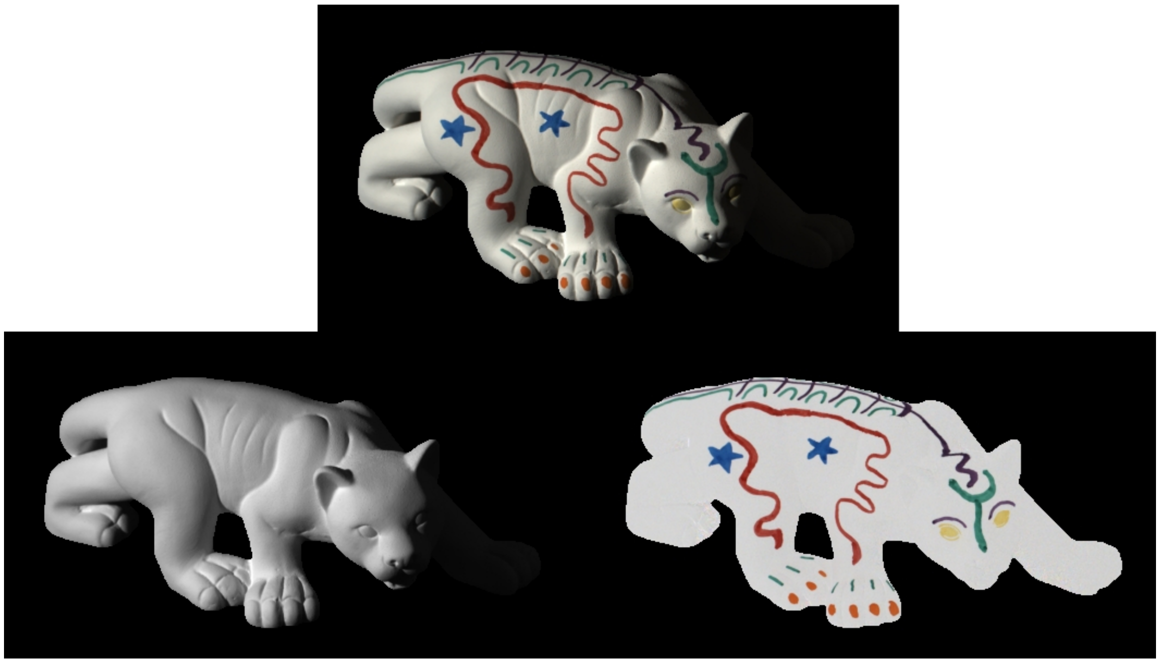

Fig. 3.11. Intrinsic image breakdown example. Top image is the original, bottom left is the shading component, bottom right is the reflectance component.

2. Interpret these gradients by estimating the gradients $\hat{r}_x$ and $\hat{r}_y$ of the log reflectance image using the heuristic:

$$\hat{r}_x = \begin{cases} i_x, & \text{if } |i_x| > T \\ 0, & \text{otherwise} \end{cases} \tag{3.11}$$

3. Compute the log reflectance image $\hat{r}$ which matches these gradients as closely as possible:

$$\hat{r} = argmin_r \sum_{x,y} |r_x(x,y) - \hat{r}_x(x,y)|^p + |r_y(x,y) - \hat{r}_y(x,y)|^p \tag{3.12}$$

Extensions of this algorithm to color images use two thresholds for estimating the reflectance image gradient. The illumination changes lie in the span of the vector $(1,1,1)^T$, called the brightness subspace. The chromaticity subspace is the null space of that vector. The thresholds used in the color Retinex algorithm are for changes in brightness and color.

These scene level properties will allow us to reconstruct the 3D environment using any off-the-shelf graphics renderer. Within the reconstruction we will then be able to make changes and re-render a modified image. The next section will provide implementation details about how this is done using the WebGL library ThreeJS.

4. IMPLEMENTATION

This section details the implementation of the re-rendering of an image using the Javascript WebGL library ThreeJS. Depth data needs to be formatted in a way which can be loaded by ThreeJS, lighting needs to be applied, and then material properties and textures need to be added to the scene. After these steps are taken the scene can be rendered to an image. We will see how replacing a surface is slightly different than inserting an object as in [15].

4.1 Depth Data to Object File

The first step is to convert the estimated depth data into a format that can be readily loaded by rendering software. One of the most common ways of doing this is to write the data to a wavefront object (.obj) file. The object file is a data format which represents 3D geometry. An object in a 3D graphics setting is a set of vertices. The object file includes information such as vertex position, texture coordinate positions (used for texture mapping), vertex normals and the faces that make each polygon defined as a list of vertices. A description of how an object file can be written can be found in [38].

We take our scene depth estimate and create two object files by splitting the original image into a foreground and a background. The foreground is the surface which we are interested in replacing, and the background is everything else in the scene. This segmentation can be done using a click based method allowing the user to easily select the surface of interest in the scene [2]. An image along with it's foreground and background models is shown in Figure 4.1. The object models are textured with the original image.

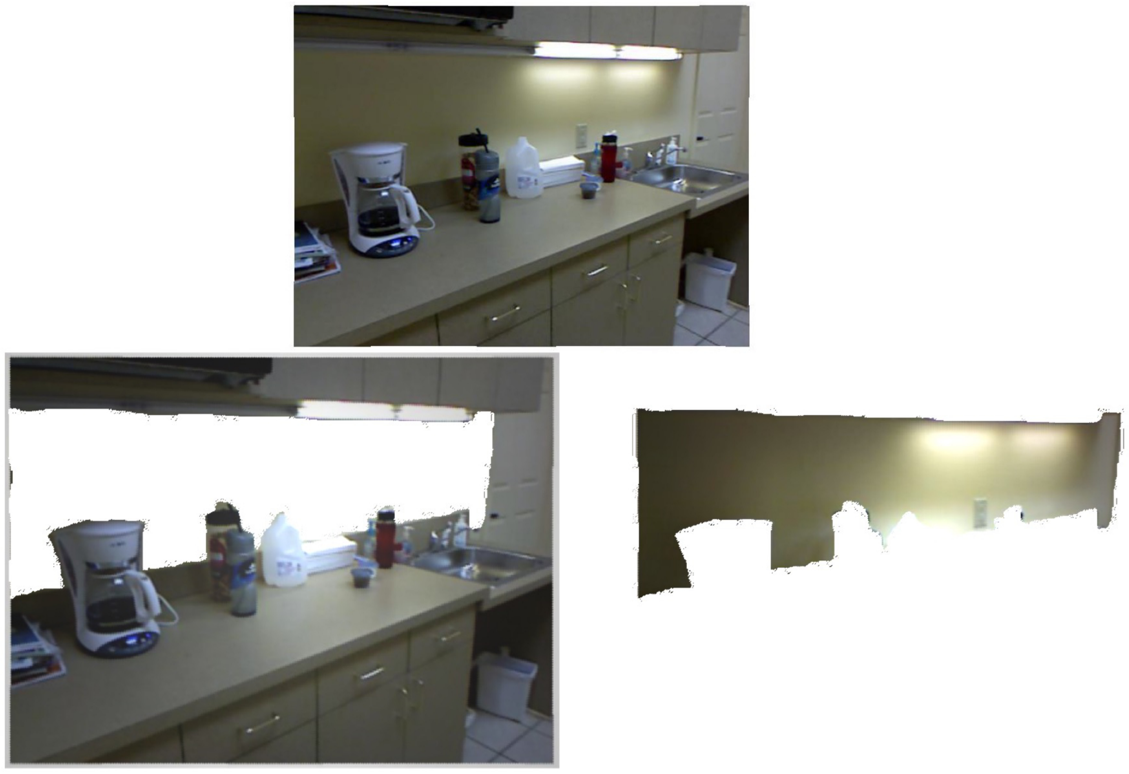

Fig. 4.1. Example of an image split into foreground and background. The foreground and background are then loaded as 3D models and textured with the original image. Top is the original image, bottom left is the background model, bottom right is the foreground model.

4.2 Texturing the Objects

Now that the depth data has been used to create and load 3D objects into the scene the objects must be textured. Texturing objects is a fast and efficient way to add detail to objects like the ones shown in Figure 4.1. In our case we are interested in adding color detail, however texture mapping can also add details in the form of: specular highlights, reflections, diffuse reflections, surface normals, refraction and transparency.

Texture mapping is essentially applying a patterned paper to a plain white box. The paper is the texture image, in our case some surface material, and the box is the object. A 2D to 3D mapping is defined so that each vertex of the object corresponds to a pixel value in the texture image. Figure 4.2 shows an example of the difference between an object with a default material and one with a texture mapped material. The texture used on the cube object can be seen in Figure 4.3.

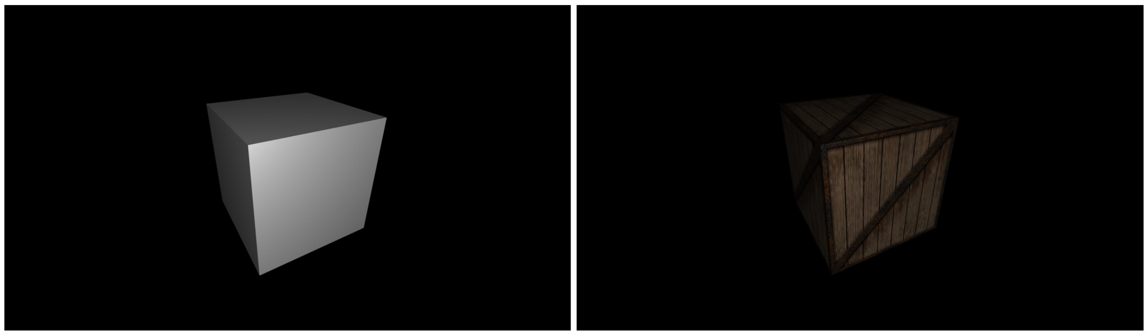

Fig. 4.2. Example of a 3D object with and without texturing. On the left is a cube with a standard material, all settings default. On the right is the same cube with a standard material but a texture map was provided.

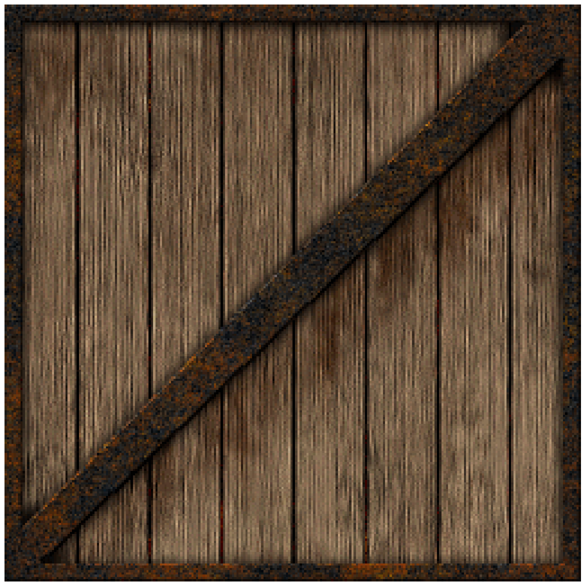

Fig. 4.3. Texture used during the texture mapping for the cube in Figure 4.2.

4.3 Lighting

Once the objects are loaded into the scene and textured we must add light sources to the scene. There are many different light types available in ThreeJS. One such light is the point light. A point light takes up one 3D location and emits light in all directions. A point light is commonly used to replicate the light emitted from a bare bulb. Another light type is the directional light. Directional lights emit light in a specific direction as though the light source is infinitely far and the rays are parallel to each other. A common use for direction lighting is to simulate daylight. The last commonly used light is ambient light. Ambient light is a global light which illuminates all objects of a scene equally. Since ambient light does not have a direction it is unable to cast shadows. A basic object lit by three different ThreeJS light types is shown in Figure 4.4.

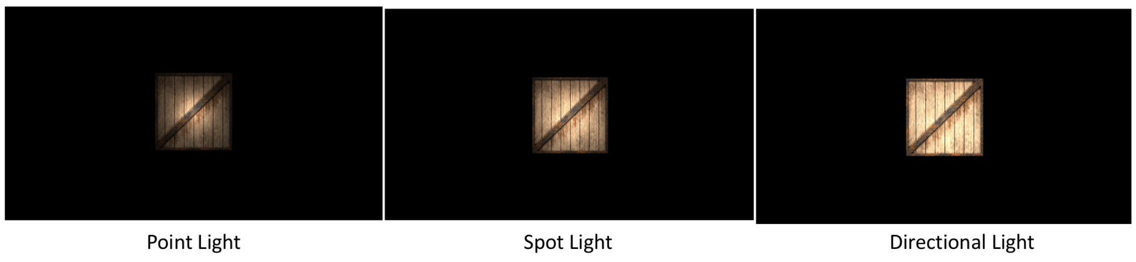

Fig. 4.4. Illustration of the different types of lights available in ThreeJS.

By combining our in-view light estimation and depth estimates we can automatically instantiate any scene lighting visible in the image. Ideally we would like to have irregularly shaped lights which would correspond to the superpixels detected as light sources. However this option is not available with the ThreeJS library. Instead we create a point light for each pixel detected to belong to a light source. Using this method can lead to too many point light sources being in the scene causing the rendering to slow down and can create saturation. Instead of creating a point light for every single pixel we evenly distribute a predetermined amount of point lights over all detected light source pixels. We can then vary the point light illumination strength to best match the original scene. An example of a rendering with varying point light intensity is shown in Figure 4.5. Every scene shown was lit with ten point lights located uniformly on the window. The intensity of the point lights were then varied from 4 watts up to 60 watts.

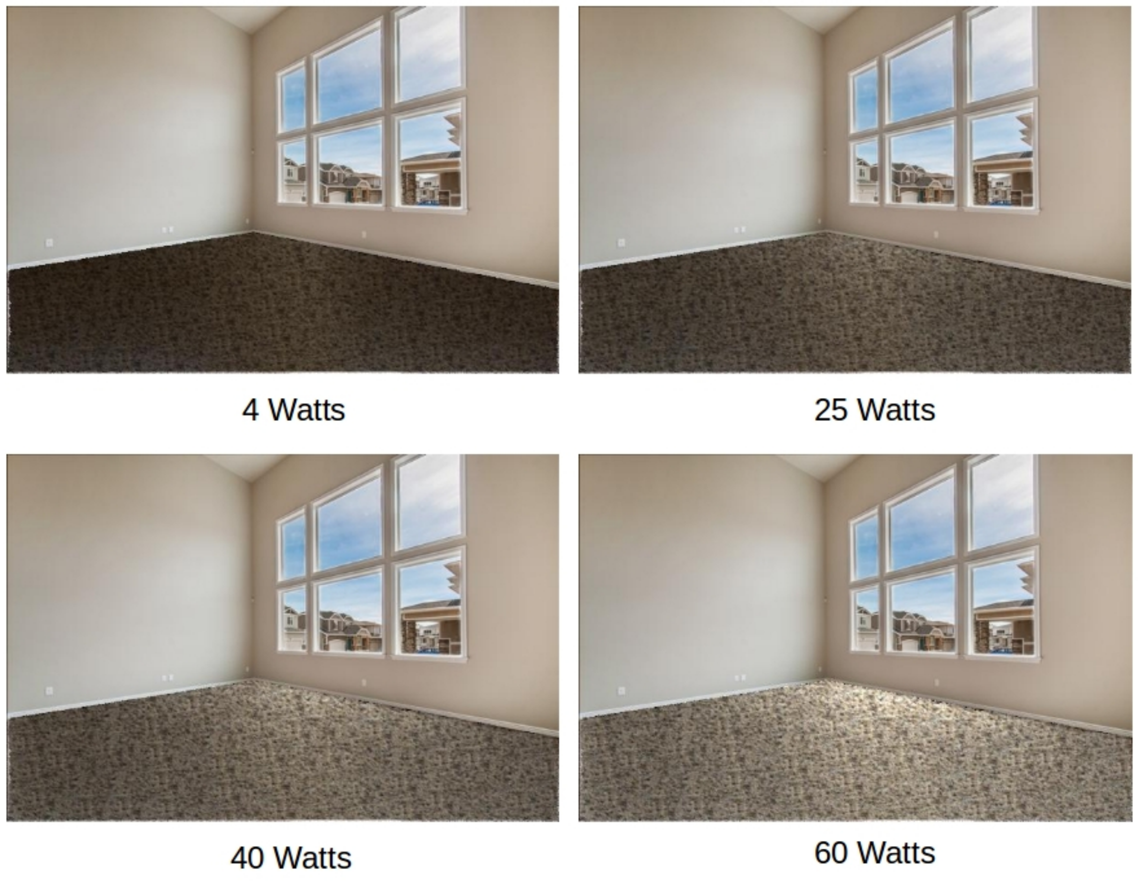

Fig. 4.5. Examples of a rendered scene with varying illumination intensity from 4 watts to 60 watts.

Figure 4.6 illustrates the lighting effects in comparison to a simpler method such as a homography transform. It can be seen that in the homography image on the right the orientation of the surface is correct, however the floor is uniformly lit without any lighting changes coming from the window. Notice that in our result on the left a gradual lighting change can be seen when moving away from the window.

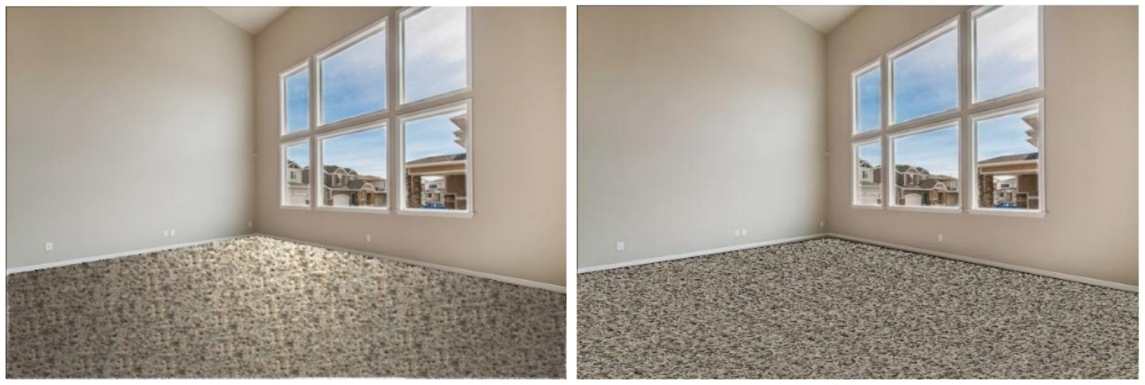

Fig. 4.6. Comparison of homography and our result. Left image is our result, right image is homography.

4.4 Material Properties

A material defines the artistic qualities of the substance that an object is made of. Materials can be used for basic effects such as object color, reflectance or shininess. Materials can also be used for complicated effects such as transparency, diffraction and subsurface scattering, where light enters the surface of an object but scatters and leaves at a different point. Many renderers contain preset material types often including: brass, skin, gold, glass or linen. It is important to note the difference between textures and materials. Textures are often used to describe more information about the material. For example, an object material may be brass, using a texture may make the object has a polished brass look. Textures are like additional layers added on top of a base material.

Material properties contribute to many of the effects seen in the final render. The rendered image is a projection of the scene onto an imaginary surface or viewing plane, analogous to film in a camera. To determine what is rendered at each point of the viewing plane we need to know the interaction the light ray reaching the point has had with objects in the scene. This is done by tracing that simulated light ray backwards from the viewing plane through the scene until it encounters a renderable object. At this point two types of interactions can take place: diffusion and specular reflection. Diffuse reflection happens when light encounters a rough surface and the light is reflected in all directions. Specular reflection takes place when light encounters a smooth surface and is reflected at a definite angle. An illustration of specular vs. diffuse reflection can be seen in Figure 4.7. The color that the object is rendered as will be determined by a combination of the diffuse and specular properties of the material along with the incident light intensity to modify the base object color. Reflections for reflective surfaces can also be rendered in this way. The incident light will be modified by the base object color and surrounding environment color depending on the reflectivity of the material. In a similar fashion transparency can be rendered with a mix of object color and background color depending on the transparency of the object.

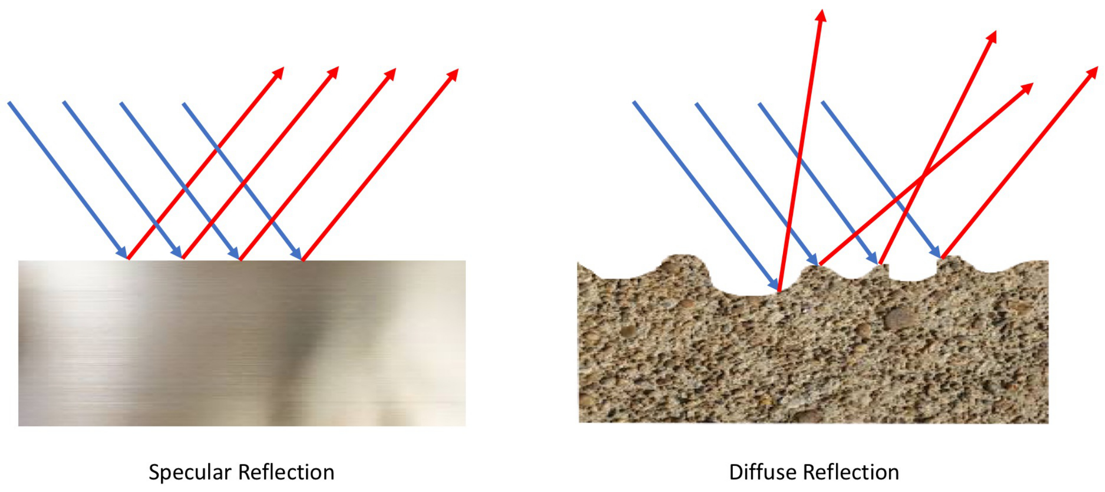

Fig. 4.7. Illustration of diffuse vs. specular reflection. Arrows represent incident light rays.

There are a variety of materials available for use in ThreeJS which all have different properties. The first is called a basic material. This is the most basic material allowing you to send a color value as well as an opacity value. Objects with this material do not have shading; they do not respond to scene lighting. The next material of interest is the Lambert material. A Lambertian surface is one which is an ideal ”matte” or diffusely reflecting surface. The brightness of one such surface is the same to a viewer no matter what angle it is viewed from. As the name implies, the Lambert material is one which can respond to lights, but only by diffuse reflection resulting in a dull shading. Another material available is the Phong material. The Phong reflection model describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. The Phong material is another material which responds to scene lighting but adds specular reflection, reflecting light with more intensity. Finally we have the standard material. In ThreeJS the standard material combines the Lambert and Phong materials into a single material. The standard material can create surfaces which look either dull or metallic. The standard material is what we use in our implementation. Examples of a single object rendered with each material type can be seen in Figure 4.8. The scene lighting in these renderings consisted of ambient light and a single point light attached to the camera.

4.5 Post-processing

Our estimates of scene properties are rough but sufficient to create more details than other methods. Since we are replacing an existing surface rather than adding a new object to the scene we want to preserve every part of the image that is not the surface of interest. If we were inserting an object we would need to worry about which parts of the original image the new object will occlude. In our case we do not need to worry about this scenario. The ground truth lighting effects already present in the original image will always be better than our scene estimates.

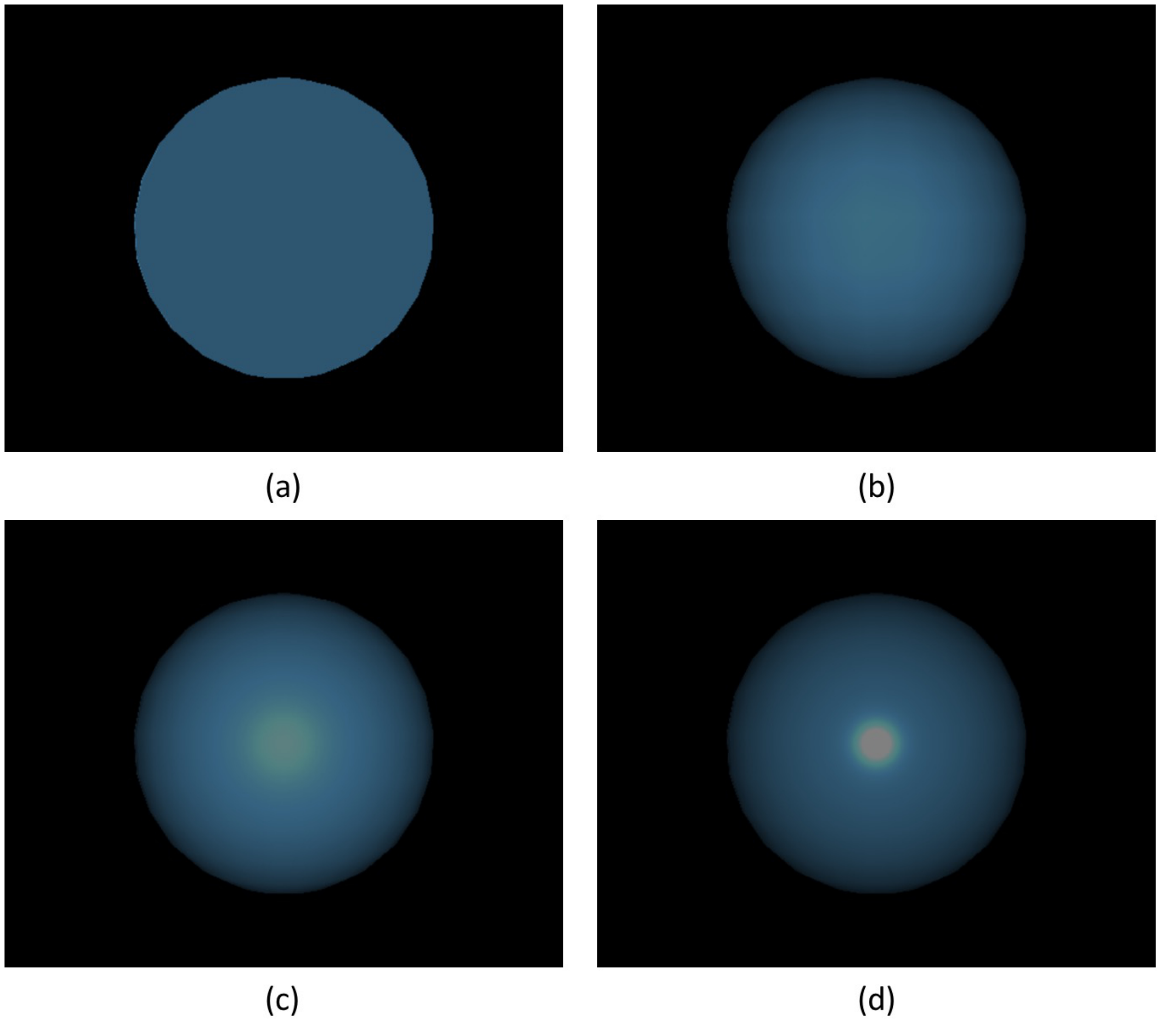

Fig. 4.8. Example renderings of different ThreeJS material types. (a) is the basic material, (b) is Lambert material, (c) is Phong material and (d) is the standard material.

To keep the existing effects while inserting the new surface we render the full image and then do some post-processing using the foreground-background mask. We need to render the full image to incorporate reflections from the background in the future. Once the rendering is done we recombine the two images into the final output by replacing the foreground pixels in the original image with the pixels from the rendered image. This leaves the background unchanged while replacing the surface of interest.

5. RESULTS

Here we provide results of our method. All figures contain the original image, the new surface texture and the re-rendered output image.

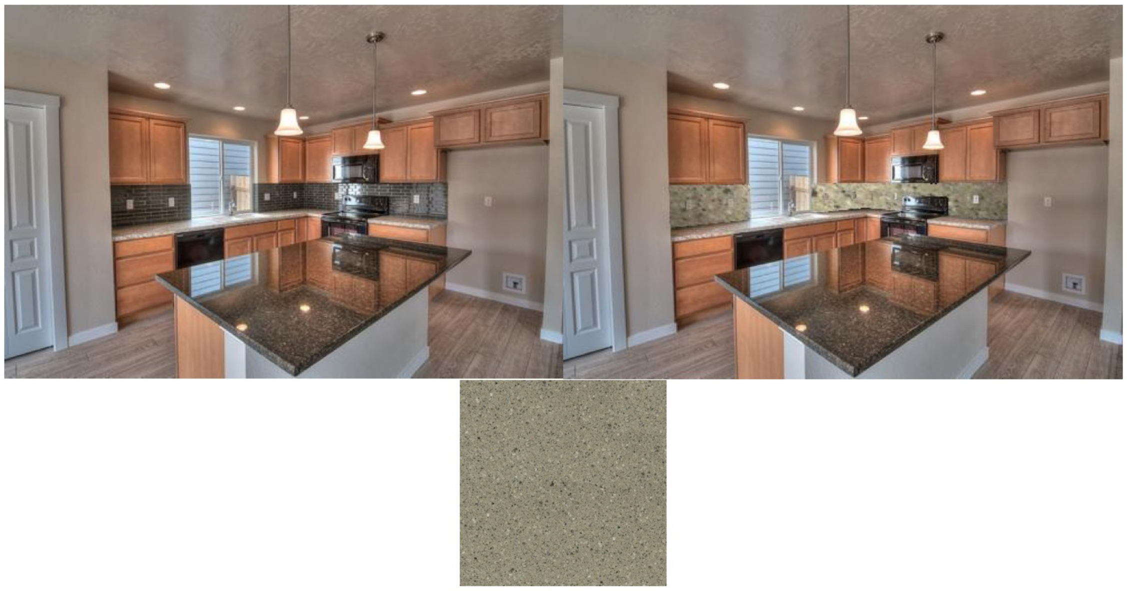

Fig. 5.1. Final output for a kitchen scene, backsplash is the replaced surface.

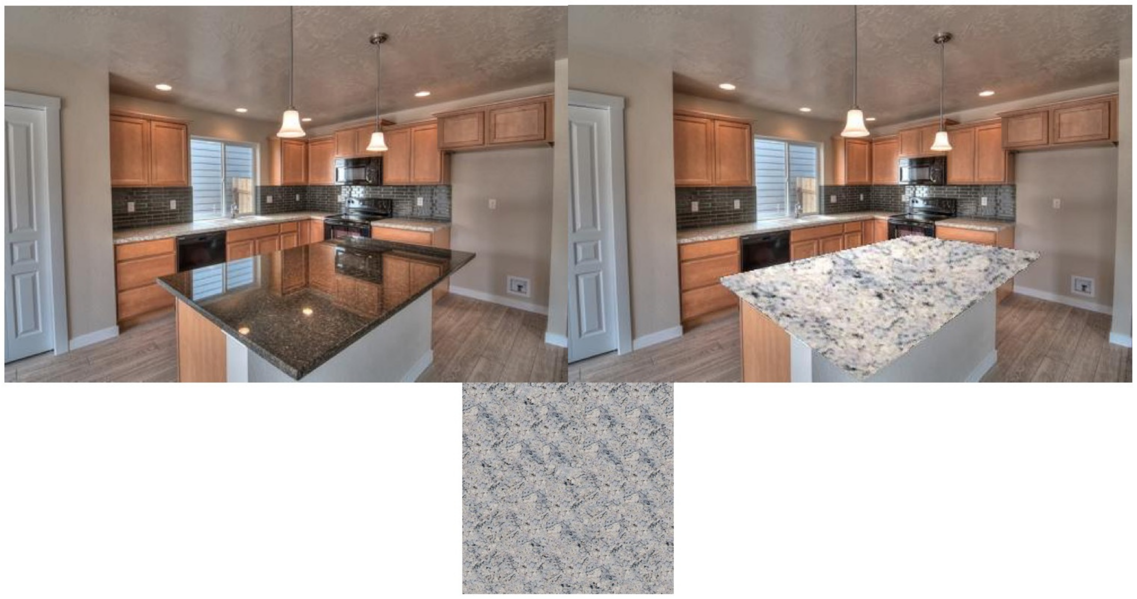

Fig. 5.2. Final output for a kitchen scene, island counter is the replaced surface.

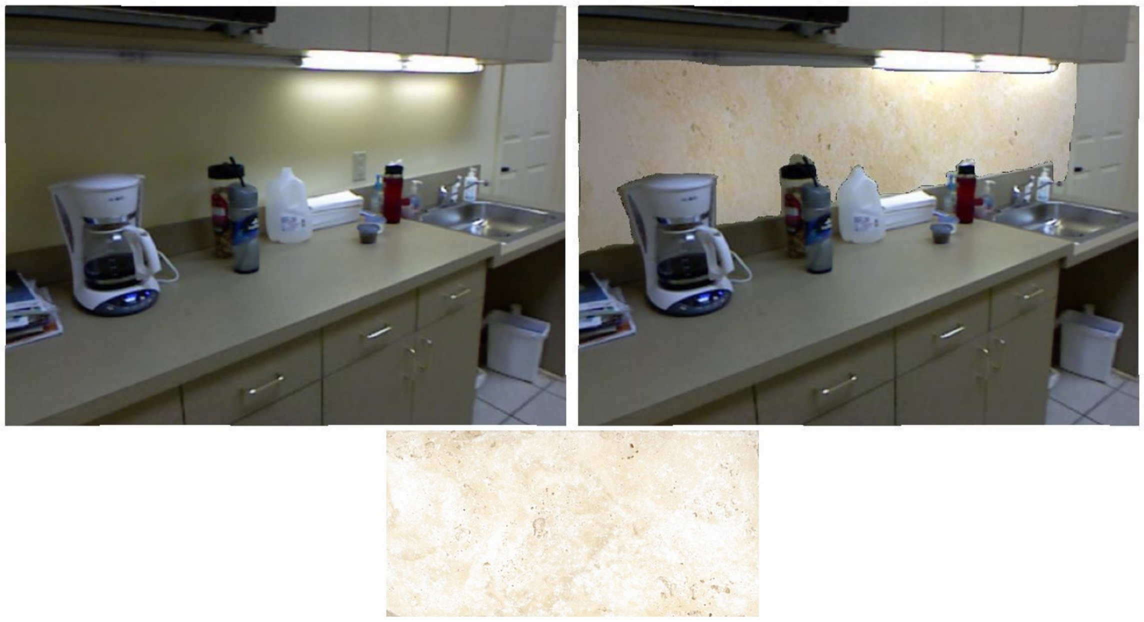

Fig. 5.3. Final output for kitchen scene, wall is the replaced surface.

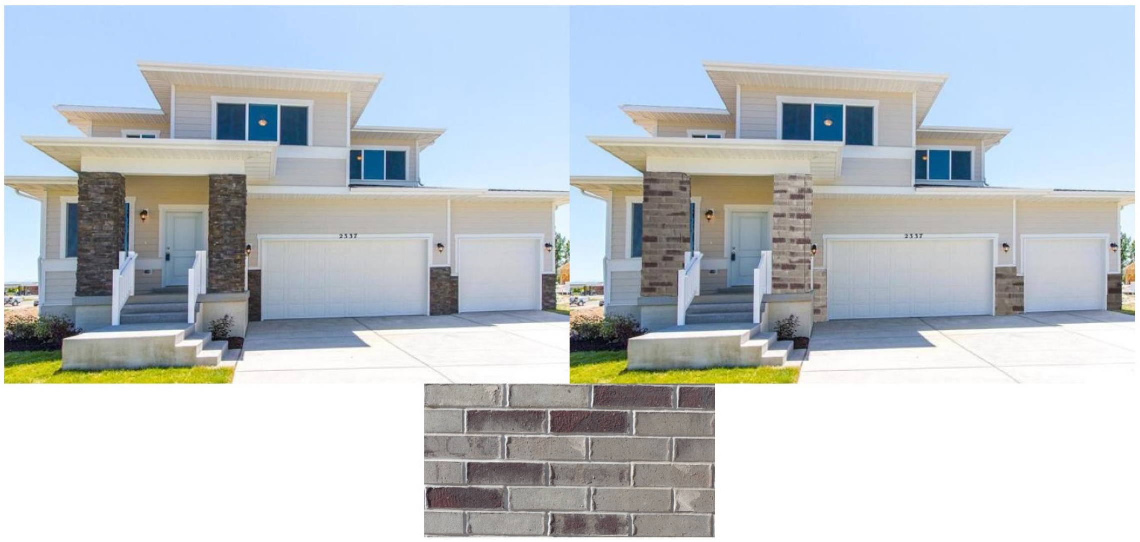

Fig. 5.4. Final output for outdoor scene, brick pillars are the replaced surfaces.

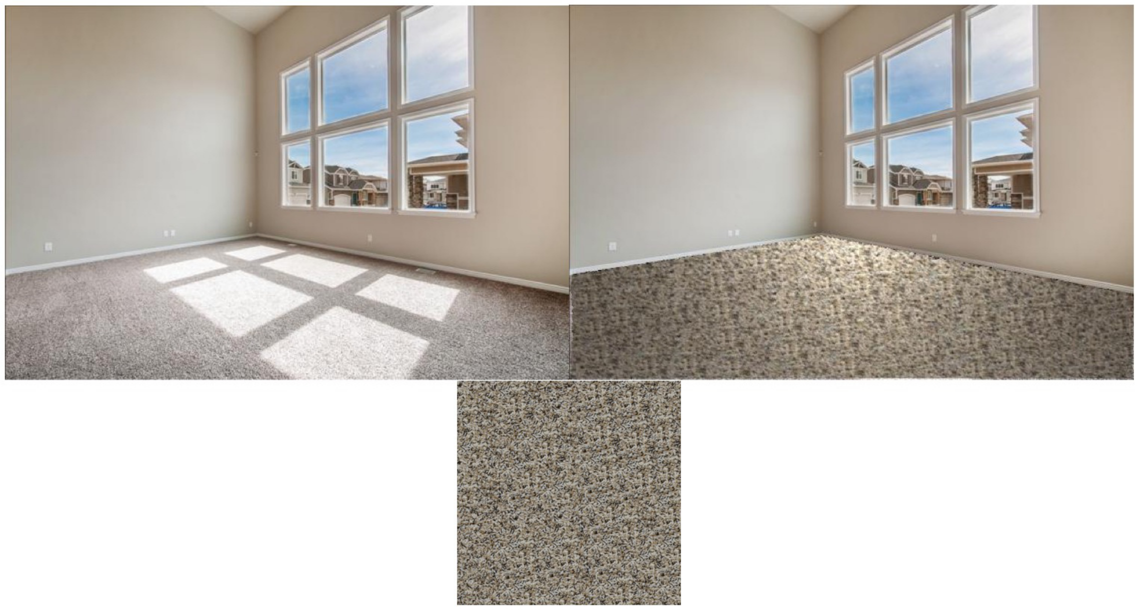

Fig. 5.5. Final output for indoor living room scene, carpet is the replaced surface.

6. FUTURE WORK

We have laid the foundations for a path towards creating photo-realistic re-renderings of indoor room scene images. There are many places where improvements can be made leading to more realistic renderings.

One of the most important improvements that can be made to complex scenes is estimating surface material properties. Reflections play a major role in photo-realistic renderings. In order to allow the new surface texture to reflect the surrounding scene we must be able to estimate the properties of the new material (metallic, ceramic, matte, etc).

A common lighting effect in indoor images is a light shaft entering a window. Our method does not currently account for this type of lighting. The actual source, typically the sun, is often not present in the image. This could possibly be done by being able to identify windows as light sources and then taking the appropriate steps to recreate the effect on the new surface. This may include creating an infinitely far light source in the scene and masking the light rays in the same shape as the identified window.

REFERENCES

REFERENCES

[1] T. Perdue, *Applications of Augmented Reality*, 2018 (accessed June 22, 2018). [Online]. Available: https://www.lifewire.com/applications-of-augmented-reality-2495561

[2] C.-J. Tai, T. Liu, J. Bagchi, F. M. Zhu, and J. P. Allebach, "Interactive segmentation for indoor scenes," *Electronic Imaging*, vol. 2017, no. 10, pp. 51–59, 2017.

[3] P. S. Heckbert, "Survey of texture mapping," *IEEE computer graphics and applications*, vol. 6, no. 11, pp. 56–67, 1986.

[4] L.-Y. Wei, "Texture synthesis by fixed neighborhood searching," 2002.

[5] M. Segal, C. Korobkin, R. Van Widenfelt, J. Foran, and P. Haeberli, "Fast shadows and lighting effects using texture mapping," in *ACM Siggraph Computer Graphics*, vol. 26, no. 2. ACM, 1992, pp. 249–252.

[6] A. A. Efros and T. K. Leung, "Texture synthesis by non-parametric sampling," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2. IEEE, 1999, pp. 1033–1038.

[7] L.-Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques.* ACM Press/Addison-Wesley Publishing Co., 2000, pp. 479–488.

[8] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Transactions on Graphics (ToG)*, vol. 20, no. 3, pp. 127–150, 2001.

[9] A. A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques.* ACM, 2001, pp. 341–346.

[10] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: image and video synthesis using graph cuts," in *ACM Transactions on Graphics (ToG)*, vol. 22, no. 3. ACM, 2003, pp. 277–286.

[11] W.-C. Lin, "A comparison study of four texture synthesis algorithms on regular and near-regular textures," 2004.

[12] T. Ojala, M. Pietikainen, and T. Maenpaa, "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, pp. 971–987, 2002.

[13] G. Ramanarayanan, J. Ferwerda, B. Walter, and K. Bala, "Visual equivalence: towards a new standard for image fidelity," in *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3. ACM, 2007, p. 76.

[14] J. Lopez-Moreno, S. Hadap, E. Reinhard, and D. Gutierrez, "Compositing images through light source detection," *Computers & Graphics*, vol. 34, no. 6, pp. 698–707, 2010.

[15] K. Karsch, K. Sunkavalli, S. Hadap, N. Carr, H. Jin, R. Fonte, M. Sittig, and D. Forsyth, "Automatic scene inference for 3d object compositing," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 3, p. 32, 2014.

[16] J. Wolfe, K. Kluender, D. Levi, L. Bartoshuk, R. Herz, R. Klatzky, S. Lederman, and D. Merfeld, *Sensation and Perception*, 2nd ed. Sinauer, 2009.

[17] L. Lipton, *Foundations of the Stereoscopic Cinema - A Study in Depth.* Van Nostrand Reinhold, 1982.

[18] K. Karsch, C. Liu, and S. B. Kang, "Depth transfer: Depth extraction from videos using nonparametric sampling," in *Dense Image Correspondences for Computer Vision.* Springer, 2016, pp. 173–205.

[19] D. Hoiem, A. A. Efros, and M. Hebert, "Automatic photo pop-up," in *ACM transactions on graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 577–584.

[20] E. Delage, H. Lee, and A. Y. Ng, "A dynamic bayesian network model for autonomous 3d reconstruction from a single indoor image," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2. IEEE, 2006, pp. 2418–2428.

[21] A. Saxena, M. Sun, and A. Y. Ng, "Make3d: Learning 3d scene structure from a single still image," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 5, pp. 824–840, 2009.

[22] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, "Indoor segmentation and support inference from rgbd images," in *ECCV*, 2012.

[23] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International journal of computer vision*, vol. 42, no. 3, pp. 145–175, 2001.

[24] C. Liu, J. Yuen, and A. Torralba, "Sift flow: Dense correspondence across scenes and its applications," in *Dense Image Correspondences for Computer Vision.* Springer, 2016, pp. 15–49.

[25] D. C. Lee, M. Hebert, and T. Kanade, "Geometric reasoning for single image structure recovery," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on.* IEEE, 2009, pp. 2136–2143.

[26] P. Debevec, "Image-based lighting," in *ACM SIGGRAPH 2006 Courses.* ACM, 2006, p. 4.

[27] K. Nishino and S. K. Nayar, "Eyes for relighting," in *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 704–711.

[28] A. Panagopoulos, C. Wang, D. Samaras, and N. Paragios, "Illumination estimation and cast shadow detection through a higher-order graphical model," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on.* IEEE, 2011, pp. 673–680.

[29] E. A. Khan, E. Reinhard, R. W. Fleming, and H. H. Bülthoff, "Image-based material editing," in *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3. ACM, 2006, pp. 654–663.

[30] J. Xiao, K. A. Ehinger, A. Oliva, and A. Torralba, "Recognizing scene viewpoint using panoramic place representation," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on.* IEEE, 2012, pp. 2695–2702.

[31] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "Slic superpixels compared to state-of-the-art superpixel methods," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 11, pp. 2274–2282, 2012.

[32] A. Saxena, M. Sun, and A. Y. Ng, "Make3d: Learning 3d scene structure from a single still image," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 5, pp. 824–840, 2009.

[33] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, vol. 2. IEEE, 2006, pp. 2169–2178.

[34] T. Joachims, "Training linear svms in linear time," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2006, pp. 217–226.

[35] H. Barrow and J. Tenenbaum, "Recovering intrinsic scene characteristics," *Comput. Vis. Syst., A Hanson & E. Riseman (Eds.)*, pp. 3–26, 1978.

[36] R. Grosse, M. K. Johnson, E. H. Adelson, and W. T. Freeman, "Ground truth dataset and baseline evaluations for intrinsic image algorithms," in *Computer Vision, 2009 IEEE 12th International Conference on.* IEEE, 2009, pp. 2335–2342.

[37] E. H. Land and J. J. McCann, "Lightness and retinex theory," *Josa*, vol. 61, no. 1, pp. 1–11, 1971.

[38] M. B. Zinck, *OBJ File Format*, (accessed May 15, 2018). [Online]. Available: http://www.cs.cmu.edu/~mbz/personal/graphics/obj.html