REAL TIME COLOR PROJECTION FOR 3D MODELS

Bruno Ferraz de Melo

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Ricardo Guerra Marroquim

Rio de Janeiro
Março de 2017

REAL TIME COLOR PROJECTION FOR 3D MODELS

Bruno Ferraz de Melo

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Claudio Esperança, Ph.D.

Prof. Leandro Augusto Frata Fernandes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2017

Melo, Bruno Ferraz de

Real time color projection for 3d models/Bruno Ferraz de Melo. – Rio de Janeiro: UFRJ/COPPE, 2017.

XV, 54 p.: il.; $29,7$cm.

Orientador: Ricardo Guerra Marroquim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2017.

Referências Bibliográficas: p. 53 – 54.

1. Real time texturing. I. Marroquim, Ricardo Guerra. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico à minha família

Agradecimentos

Em primeiro lugar gostaria de agradecer ao VCG por ceder os datasets utilizados e ao Museu Histórico Nacional por ceder a urna que foi escaneada e fotografada para geração de um novo dataset.

Agradeço também à Glória por todo incentivo e companhia nas noites insones para que conseguisse focar adequadamente.

À Susan, meu agradecimento por seu incentivo e por sua paciência. Além disso, desculpas por tê-la imposto a necessidade da paciência.

Ao Anderson, à Silvia e ao Pablo, obrigado pelas ideias de projeto e discussões desde muito antes de ingressar no mestrado.

Para a Monica, Cristina, Caito, Aline, Chris e Paola, um agradecimento especial pois, por meio de exemplos cotidianos, conseguiram mudar minha visão a respeito do mundo acadêmico e me fizeram acreditar que esse pode ser um ambiente real de troca e construção de conhecimento.

Minha gratidão a todos do LCG, não só por terem conseguido construir esse raro ambiente mas também por confirmarem essa minha nova expectativa. Além disso, por terem me aceitado e ajudado apesar de todas as minhas dificuldades, afinal, esse mestrado não teria sido possível sem as seguintes ajudas:

- Do Luiz Santos e do Flavio que me orientaram nos primeiros passos com Python e me introduziram ao mundo do linux.

- Das dicas de C++ do Leo, do Felipe e do Marcelo.

- Do Vinicius e do Luiz Maurilio em visão computacional

- No suporte do Bernardo com instalações e formatações.

- Na inestimável ajuda do Lucas, Tarsus, Nilton e Andrea para me integrar ao grupo.

- Do Pedro e Daniel, a quem costumeiramente recorria nos momentos de desespero. Que me ensinaram muito e me ajudaram em quase todos os trabalhos ao longo dos 3 anos.

- Da Lurdes que diariamente preparava o café sem o qual não teria me mantido acordado.

Agradecimentos especiais:

- Aos cuidados do professor Esperança de, por incontáveis vezes, mudar a abordagem do conteúdo, simplificando até que esse se fizesse claro independente do tempo e do trabalho que fosse necessário.

- Aos cuidados do professor Marroquim em me ajudar não só na adaptação no início do curso mas também na adequação do escopo das minhas ideias ás minhas capacidades, orientando e cobrando de forma muito eficaz, buscando sempre me direcionar apesar das minhas dificuldades e constantes atrasos.

Por fim, gostaria de agradecer aos professores que promovem a manutenção do ambiente do LCG sempre se mostrando solícitos e presentes. Por me ensinarem muito mais que computação gráfica ao me mostrar que atenção ás necessidades individuais dos alunos pode ser o diferencial para o entendimento de qualquer contedo. Prática essa que passei a utilizar em sala de aula.

Correndo o risco de parecer redundante, sou muito grato pela oportunidade que recebi, pela atenção no decorrer do período e pela ajuda no desenvolvimento das qualificações que me faltaram ao longo do caminho. Admiro muito o trabalho de todos, não só pela qualidade e habilidade técnica, mas principalmente, por jamais se descuidarem do lado humano do processo. Algo raro hoje em dia.

Obrigado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROJEÇÃO DE CORES PARA MODELOS 3D EM TEMPO REAL

Bruno Ferraz de Melo

Março/2017

Orientador: Ricardo Guerra Marroquim

Programa: Engenharia de Sistemas e Computação

Essa dissertação apresenta uma solução para visualizar, em tempo real, datasets compostos por um modelo 3D e um conjunto de fotos calibradas. Nossa solução seleciona, projeta e compõe as fotografias em função da posição e da direção da câmera de forma a maximizar a percepção de detalhes e, ao mesmo tempo, atingir taxas interativas de visualização. O método funciona como um gerador dinâmico de texturas, onde para cada novo ponto de vista a melhor combinação das fotos é buscada. A principal vantagem da nossa abordagem é tentar preservar as informações originais das fotos da melhor forma possível. Além disso, os resultados do método proposto foi comparado com o tradicional texture mapping. Revelando, assim, mais precisão e menos artefatos para datasets extensos com câmeras distribuídas não uniformemente.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

REAL TIME COLOR PROJECTION FOR 3D MODELS

Bruno Ferraz de Melo

March/2017

Advisor: Ricardo Guerra Marroquim

Department: Systems Engineering and Computer Science

In this work, we present a solution for interactive visualization of virtual objects composed of a 3D model and a set of calibrated photographies. Our approach selects, projects and blends the photos based on a few criteria in order to improve perception of details while maintaining an interactive performance. It works as a dynamic texture map generator, where for each new view position and direction the best combination of the photos is sought. The main advantage of our technique is that it tries to preserve the original photo information as best as possible. Furthermore, the proposed method were compared with a popular texture mapping technique. Our method produced less artifacts in general, and was able to handle better large and non uniform datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the yet brief history of computer graphics, photo-realism has always been one of its holy grails. Despite the impossibility to recreate all physical aspects and conditions in a virtual environment, physically based renders are capable of achieving a very convincing representation of the real world in many situations. Nevertheless, in a real-time scenario, shortcuts to approximate the visual appearance have been proposed in a myriad of different manners. Among these, one of the most popular methods is texture mapping, the central issue discussed in this work. In spite of the multitude of texture mapping techniques and applications, in this work we will limit our scope to texture maps created from a real physical object, that is, generated from a set of photographs of the object.

On one hand, textures provide a compact and powerful appearance representation of a real object. On the other hand, since it does not model the complete underlying reflectance behaviors, it is a limited representation of how light interacts with each point on the object's surface. The texture also imposes a bound in how much detail can be represented. Quoting LEMPITSKY e IVANOV [5]: "... a texture map is an important component of a geometric model, and the texture quality and resolution have a key impact on the model realism.". Effects such as pixelation when surpassing the resolution limit of the texture are practically unavoidable without limiting other aspects of the visualization system.

This work touches a similar, but different aspect of this resolution bound. The main idea revolves around the fact that when compressing the information of a set of photographs into a single texture, which is nothing more than an image itself, information loss in inevitable. As an alternative visualization paradigm, methods have been proposed to navigate directly within the photo set, and are able to profit from the full resolution of the photos by only displaying one at a time overlaid over the 3D model. The drawback in this case is that the feeling of a virtual replica is lost.

We propose an approach that tries to bring together the best of these two worlds:

create the texture representation in real-time using the most appropriate photos of the dataset for any given view direction and position in space.

Briefly, we first pre-compute weights for each pixel of each photo using previous image blending approaches for generating textures. Then, during the interactive visualization, we blend the images using the pre-computed pixel weights and view dependent camera weights, where the latter gives the overall contribution of each photo to generate a texture representation given a view position and direction.

The main goal is to preserve as best as possible the original resolution of the photos. The intuition behind this approach is that when inspecting the model in a detailed manner, close up photos should be prioritized, while when navigating from farther away, the high-resolution details will not be visible, so a set of photos with a more general view should be used. Note that when generating a single texture from a set with photos taken from a wide range of distances, some level of detail information is lost when blending the pixels.

This work is divided in the following manner: in Chapter 2 we describe the most relevant related works; Chapter 3 describes the proposed method while Chapter 4 reports implementations details. Results and discussions are presented in Chapter 5, and Chapter 6 concludes this dissertation.

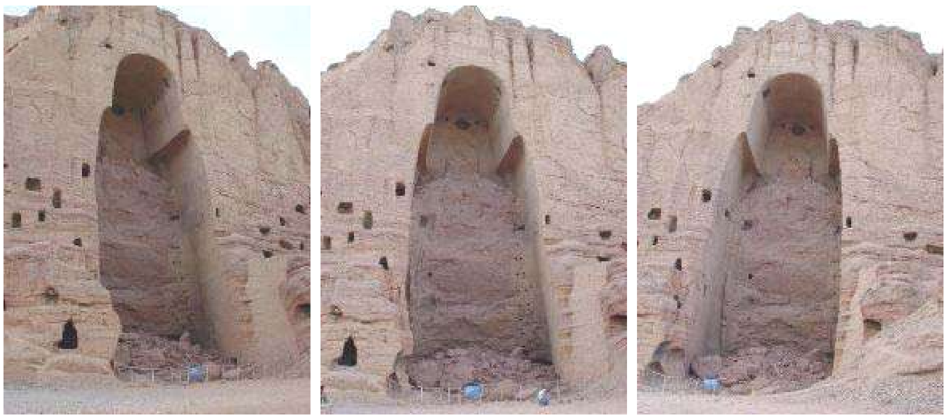
Chapter 2

Related Works

This section first reviews some closely related works in a more general sense. Then, describes in more detail two works that most inspired ours.

ORTIN e REMONDINO [1] proposed a method to deal with undesired occlusions when texturing 3d models based on a set of pictures. It is not rare to find moving or static objects occluding the central theme, such as moving pedestrians. They begin by proposing a solution for façades, and exploiting the fact that they can often be locally approximated by planar facets. In this case, homographies between pairs of adjacent images are enough to generate a new virtual texture. Due to the redundancy in the data, and the difference in parallax movement between the planar face and the impostors, a median filter is used to remove the occluders. Although this approach is limited to planar features, it can be extended to general scenes and complex 3d objects by considering that it is composed of small roughly planar manifold patches. Moreover, this paper offers a good survey on issues that can affect the photo-realism of textured 3d virtual models, as well as advices and references on how to solve them. Figure 2.1 shows a result of the proposed method [1].

PREVITALI *et al.* [2] describe an approach to reduce human intervention for texture mapping while obtaining an accurate and photo-realistic result. They focus on two different problems: occluded areas, and sharp radiometric transitions between images due to different illumination conditions. To solve the first issue they perform a visibility analysis by projecting the triangles and testing for intersections in image space. If two triangles intersect, the closest one to the camera is the occluder, and the farthest is the occluded. The visibility test is exemplified in Figure 2.2. Since this is a costly procedure, they employ a series of extra steps to reduce complexity, such as view frustum, back-face, and triangle distance culling. Once the visibility is computed, they pick the best texture for each triangle by assigning scores based on two parameters: the image resolution in object space; and the camera view direction.


(a) Three reference images from different points of view with occlusion.

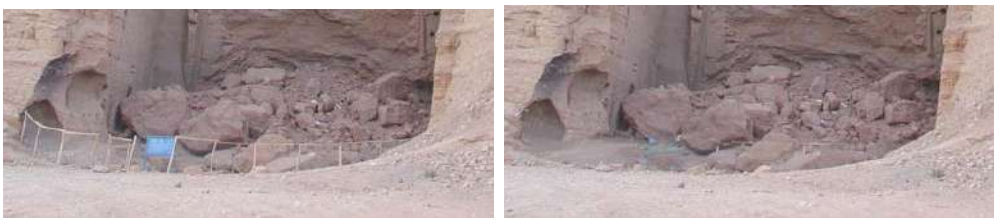

(b) A close-up view of one reference image (left), and the occlusion free generated image (right).

Figure 2.1: Example of a semi-automatic occlusion free generated image. Images extracted from ORTIN e REMONDINO [1].

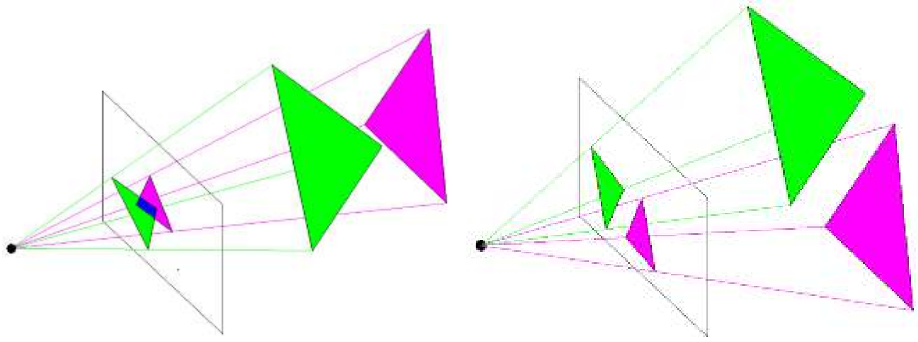


Figure 2.2: Visibility test proposed by the authors. By analyzing the triangles projections, it is possible to know if there is occlusion (left), or not (right), since intersection on the projection plane only occurs where there is occlusion. Images extracted from PREVITALI *et al.* [2].

In addition to the visibility test, they also propose a method to minimize illumination variation between adjacent triangles, since they may be mapped to different textures. The color / brightness correction is performed on $L*a*b$ color space, rather than traditional RGB, since the L component better approximates the human perception of lightness and allows for a more accurate contrast adjustment, while the a and b components are used for color balance. The user chooses a reference image, and a common feature point is used to pointwise estimate color and brightness differences. These samples are used to compose brightness variation functions between a given image and the reference one, and to interpolate over the rest of the image. Figure 2.3 illustrates the color/brightness correction method.

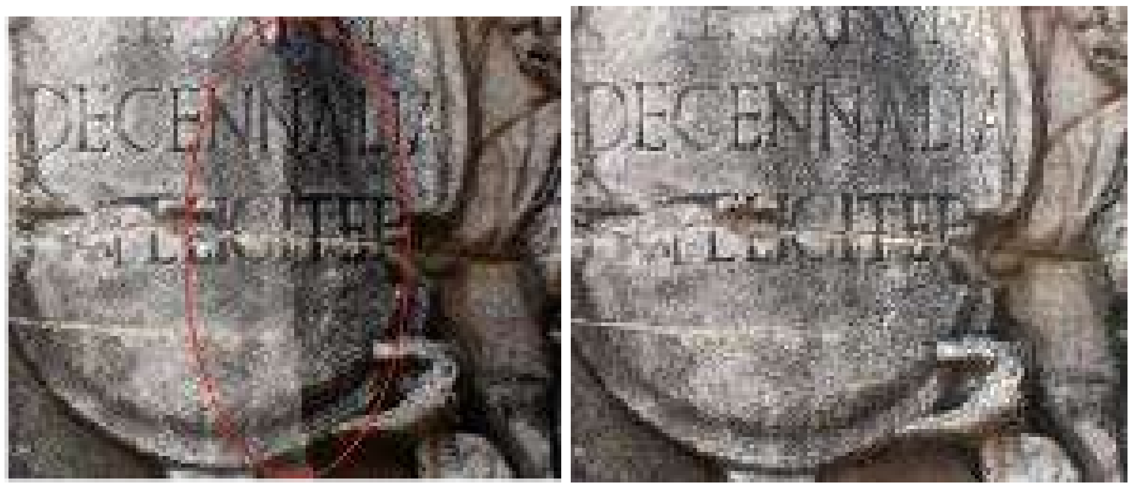


Figure 2.3: Color and brightness correction to remove seams. Texture model without correction (left) and with correction (right). Images extracted from PREVITALI *et al.* [2].

BAUMBERG [3] developed a system that builds a seamless texture map from an arbitrary surface topology obtained from a real object and a sparse set of photos and their respective camera parameters. The texture "splining" in 3d technique starts by rendering a gray scale weight image for each camera, in order to compose a weighted function. The shades of gray represent each triangle's ratio between the area of the projected triangle and the surface area of the triangle.

The raw gray image is Gaussian blurred and zero clamped in order to smooth the transitions, while the internal silhouette is extracted with black edges and white background and then feathered in order to mask the blurred image. This guarantees a continuous weight function without sharp transitions. Furthermore, Burt and Adelson's multiresolution spline [10] for blending two images is extended to 3d surfaces. The original images are split into low and high frequency bands and separately projected to a texture map representation along with the respective weights, and blended using pixel-wise operations. The low band images are blended using an weighted average while the high band images are blended with a nonlinear filter. The bands are finally combined generating the final texture. Figure 2.4 illustrates their results and a comparison with simpler methods.

BERNARDINI *et al.* [4] propose an acquisition method that combines geometric and texture information to achieve a more accurate registration. Correspondences from the range images are used to align the geometric scans into a single mesh. Each scan is divided into patches, where each patch is assigned a single range image.

The final texture is generated by combining data from multiple range images.

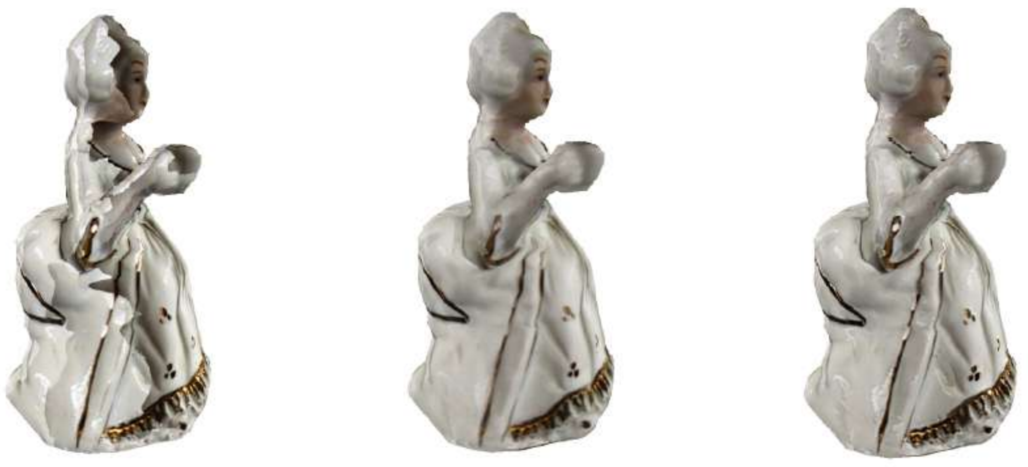
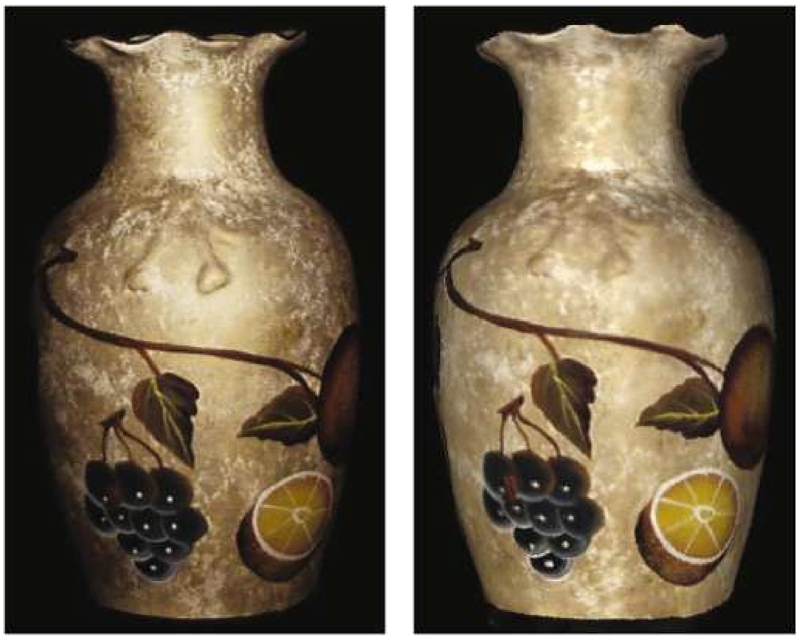


Figure 2.4: Comparing results using only the best camera (left), a simple average (middle), and the proposed multiband strategy (right). Images extracted from BAUMBERG [3].

The weights are based on a combination of two maps: the first contains the ratio between the cosine of the angle between the surface normal and the camera direction and the square distance to the camera; and the second is a photometric confidence value. More specifically, the second map assigns a high weight to pixels where the surface normal can be recovered from the photometric data, and a low weight otherwise. Both maps are smoothed to avoid discontinuities, and multiplied and rescaled to generate a single weight map in range $[0, 255]$. Since scans overlap, patches from different scans will also overlap. To blend the information into a single texture map, the final texel color is computed as the weighted average of the corresponding pixels from each overlapping scan, and a smooth transition between patches and scans is achieved. Figure 2.5 illustrates some results for this method.

LEMPITSKY e IVANOV [5] propose a mosaicing approach for creating a texture map from multiple photos. Their goal is not only to separate the geometry into patches, where each one is assigned the best image, but to create the patches in such a way that the transition between them is as less visible as possible. In a first step their approach creates the mosaic using a Markov Random Field energy minimization strategy, to enforce that two adjacent patches are as similar as possible at their common border. As a subsequent step they perform seam leveling, since not all transitions generated from the optimization method are completely unnoticeable. A result of their method is depicted in Figure 2.6.

In a similar manner, GOLDLUECKE e CREMERS [6] proposed an energy minimization strategy to generate superresolution texture maps from multiple images. The authors focus on solving the following problem: on one hand, using few images a sharp texture is achieved but with visible seam, on the other hand, using many


(a) A reference image (left) and the reconstructed texture model (right).

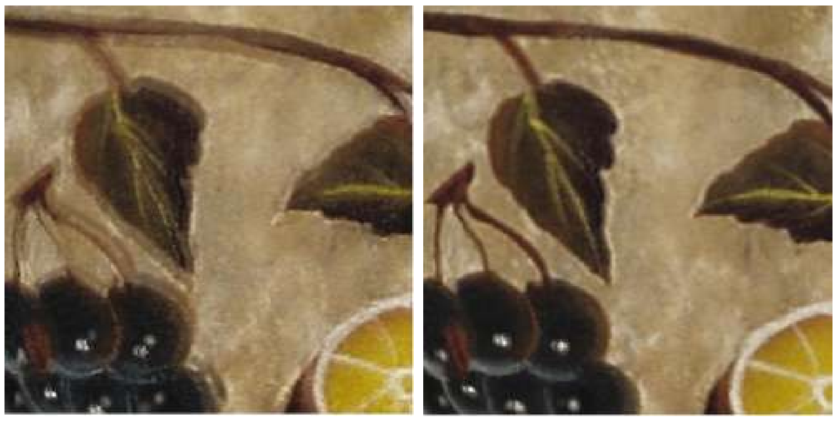

(b) A close-up view of the reconstructed model without the image-based registration (left), and with the image-based registration (right).

Figure 2.5: Images extracted from BERNARDINI *et al.* [4].

images do not create visible seams but usually blurs the final result. The main innovation is the solution for superresolution on curved surfaces, that deblurs results from blending multiple images, thus achieving a sharper result even when using a large set of photos. A resulting texture model can be seen in Figure 2.7.

OISHI *et al.* [7] exploit the by-product from laser scanners, the reflectance image. This is a measure of the reflected laser intensity at each pixel of the range image. The main idea is to first colorize the reflectance images using the photos, and then transfer this color information to the model. Their method is based on creating small patches on both the reflective image and the photos, and finding correspondences between them. Color information is then assigned to the center of each patch of the reflective image and spread to fully color the image, as illustrated in Figure 2.8.

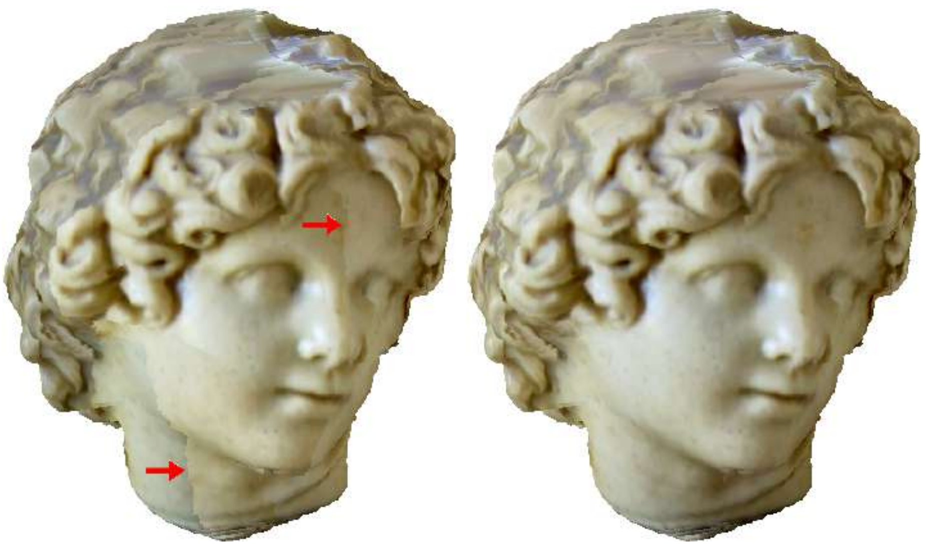
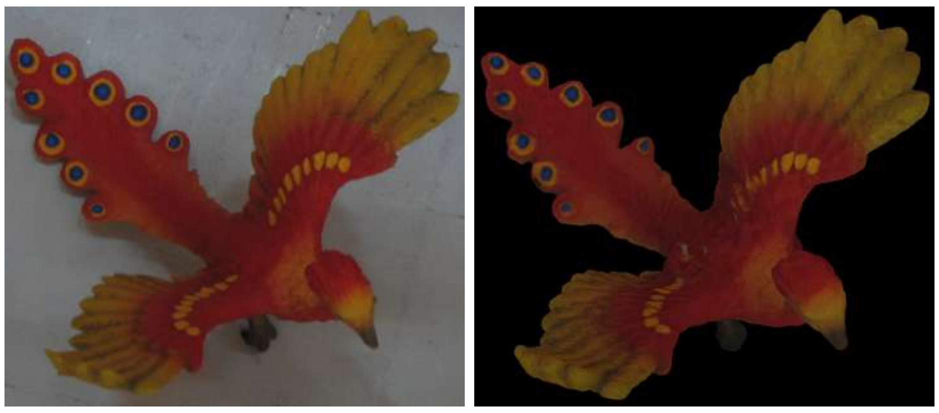
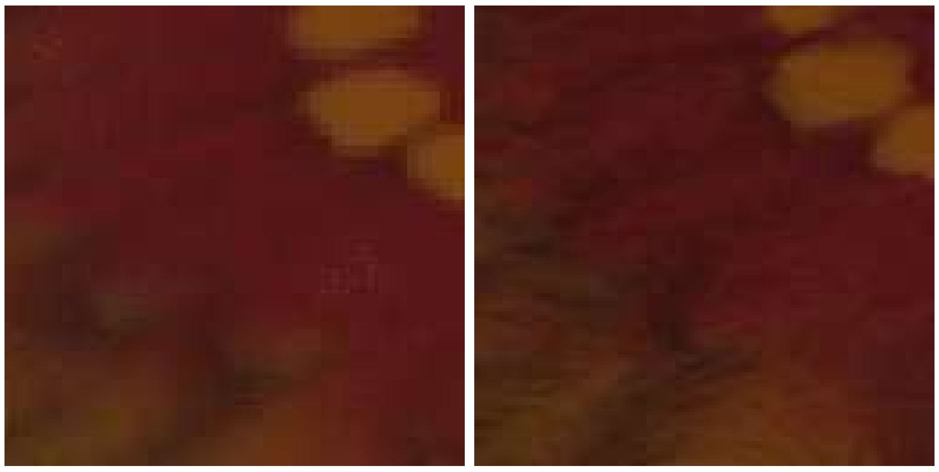


Figure 2.6: Texture produced without the minimization strategy (left), and with the proposed strategy (right). Images extracted from LEMPITSKY e IVANOV [5].



(a) A reference image (left) and the reconstructed texture model (right).



(b) A close-up view of a single input image (left), and the resulting superresolution texture (right).

Figure 2.7: Images extracted from GOLDLUECKE e CREMERS [6].

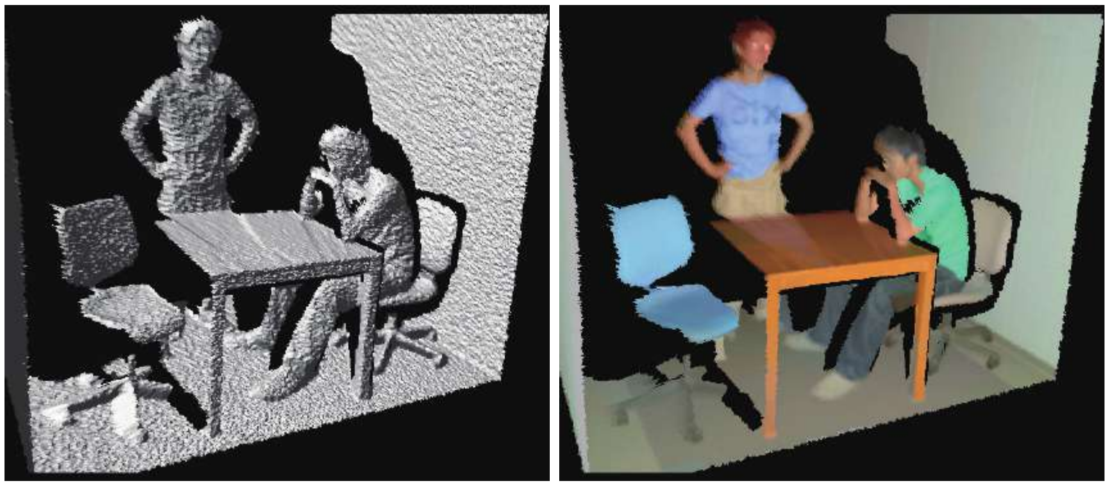


Figure 2.8: Input 3d geometry (left) and the result of the automatic colorization method (right). Images extracted from OISHI *et al.* [7].

2.1 Masked Photo Blending: Mapping Dense Photographic Dataset on High-Resolution 3D Models

CALLIERI *et al.* [8] proposed a multivariate blending function that operates in image space by mixing data from multiple images, called Masked Photo Blending. Weights are attributed per pixel in order to maximize the contribution regarding geometric, topological, and colorimetric criteria. In a first step, a depth map is computed from each calibrated camera. It is then used to solve occlusions and detect which vertices are visible from each camera in order to compute texture coordinates for the visible ones.

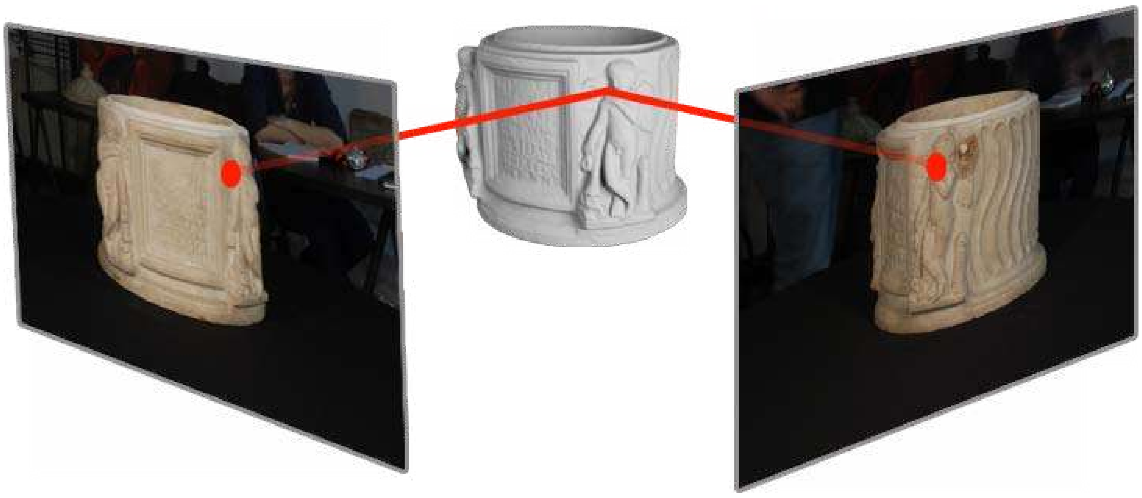


Figure 2.9: A vertex is visible from multiple cameras and the projected pixel color may vary between them. The method tries to solve this ambiguity by generating one single texture map.

Since it is not unusual that a vertex is visible from multiple cameras (see Figure 2.9), choosing a single source or computing a naive average might result in artifacts such as blurring, ghosting, seams or discontinuities. The approach tries to deal with these issues by computing weight masks for each image in order to prioritize reliable information. More specifically, three pixel masks are proposed: angle, depth, and border.

Angle Mask: This criterion takes into account the angle between the camera view direction and the surface normal for each pixel. Following the idea of Lambertian illumination, the weight achieves maximum value when the view direction is coincident with the surface normal, as illustrated in Figure 2.10. The angle weight is defined as the cosine between the two normalized vectors, and is in range $[0, 1]$.

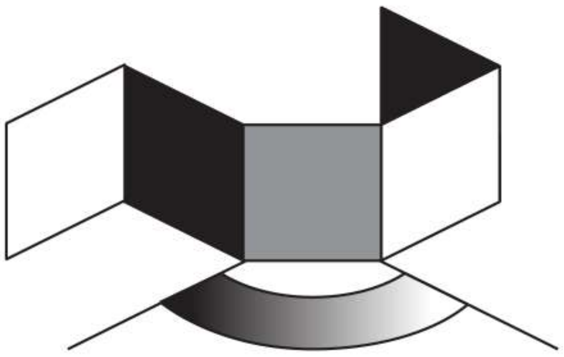


Figure 2.10: Angle Mask: computed as the dot product between normalized view direction and surface normal vectors. Black regions have zero angle weight, and white regions have maximum angle weight.

Depth Mask: This weight approximates the surface sampling rate during the image acquisition, that is, it is possible to encode more information when the surface is near the camera. The mask is computed from the depth map and the weight decreases quadratic in regards to the distance of the surface (see Figure 2.11). This mask is normalized in range $[0, 1]$.

Border Mask: This mask deals with discontinuities on the depth map, since the texture could have artifacts due to abrupt lighting differences between cameras. Thus, this mask evaluates how far pixels are from borders in the depth map.

Final Mask: The final mask for each image is computed by multiplying its angle, depth and border masks. Hence, the final weight is only as high as the lowest weight between the three masks. For instance, if a pixel has zero weight for any of the three masks, its final weight is zero independently of the other two weights. This is particularly important to remove outliers.

The three masks are exemplified in Figure 2.12. Finally, once the final mask is computed, a point on the 3d surface, or a texel on the final texture, is assigned a color by averaging the projected pixels weighted by their final mask values. A resulting textured model is illustrated in Figure 2.13.

This approach can be easily extended with other quality image estimators depending on the application. For example, in the paper, the authors propose two other criteria: Stencil mask and Focus mask. The first one could be used to remove any unwanted object, such as occluders. The second one could be used to assign more weight to areas on focus, as opposed to those far from the depth of field of the camera.

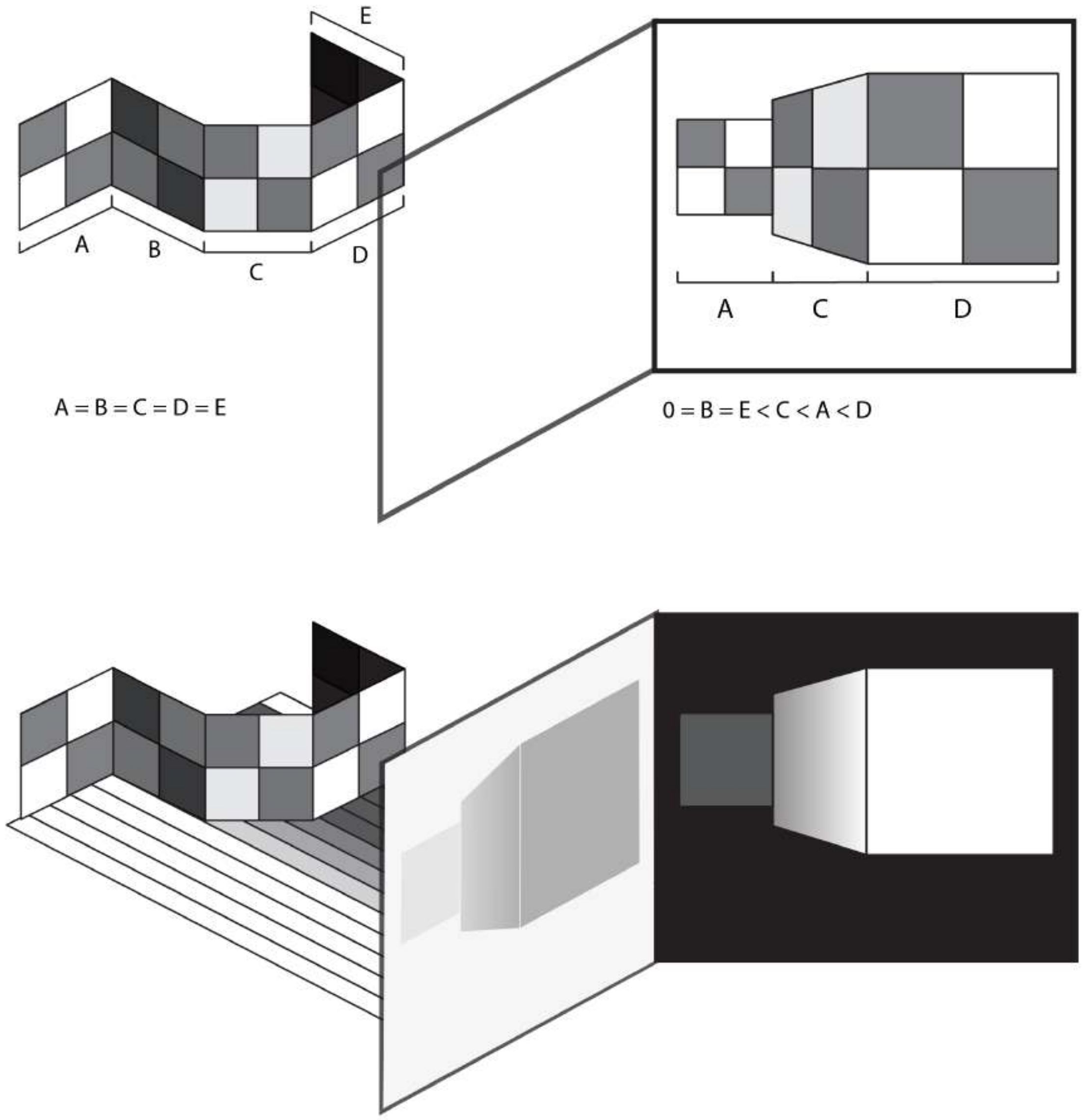


Figure 2.11: Depth Mask: objects far from camera are less sampled than near ones (top); a depth mask is created to represent the distance to the camera, where closer vertices have higher weights (bottom).

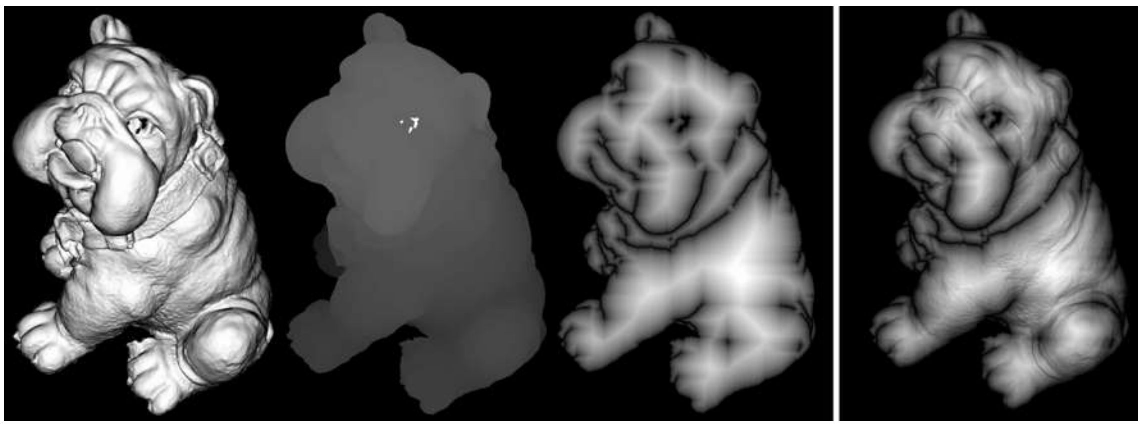


Figure 2.12: From left to right: angle mask, depth mask, border mask, and final fused mask. Images extracted from CALLIERI *et al.* [8].

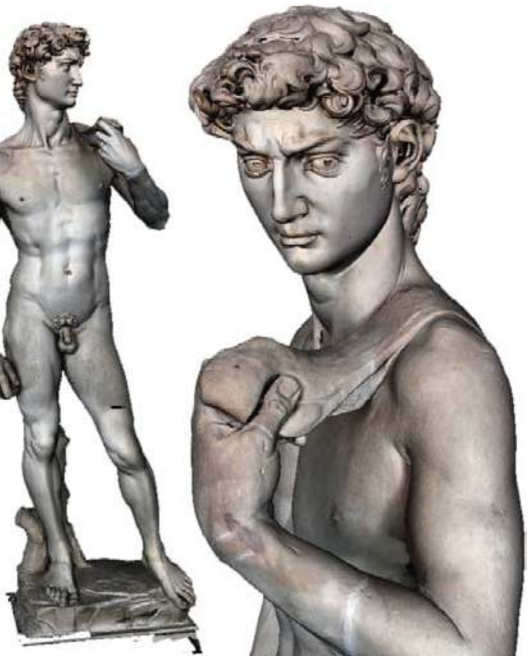


Figure 2.13: Result of a 3d model texture mapped with the Masked Photo Blending approach. Images extracted from CALLIERI *et al.* [8].

2.2 PhotoCloud: Interactive Remote Exploration of Joint 2D and 3D Datasets

As examples of a visualization system that works by navigating through a set of photos, as opposed to texturing 3d models, we can cite Photo Tourism by SNAVELY *et al.* [11] and PhotoCloud by BRIVIO *et al.* [9]. In this section we describe the latter and more recent work, PhotoCloud. Their goal is to propose a real-time client-server system in order to explore large datasets comprising of 3d models and registered photographs. The 3d model can be acquired using scanner, for example, or extracted directly from the photos. In fact, one of the advantages of the method is its flexibility in regards to the 3d geometry. The system actually uses a multiresolution representation of the geometric data, to improve performance using level-of-detail strategies.

The central idea of the approach is to create an integrated navigation system, using framelets to represent images from the datasets, and a navigation bar to order the photos based on their relation to the current view point. Only one photo is exhibited at a time, that is, the photo with highest weight, so no sophisticated blending is necessary. The weight criterion to choose the best image is based on the distance of each image to the virtual camera, and the angle between their view directions. They also create a smooth transition between images when moving the viewpoint to avoid abrupt jumps in the navigation, but never blend two images for more than a few milliseconds for viewing purposes. Another interesting approach employed is to add a sky-dome to create smooth transition for incomplete 3d models, objects that are not present in the 3d model (such as background objects or the sky), or to mask misalignments. Figure 2.14 illustrates the PhotoCloud system in motion.

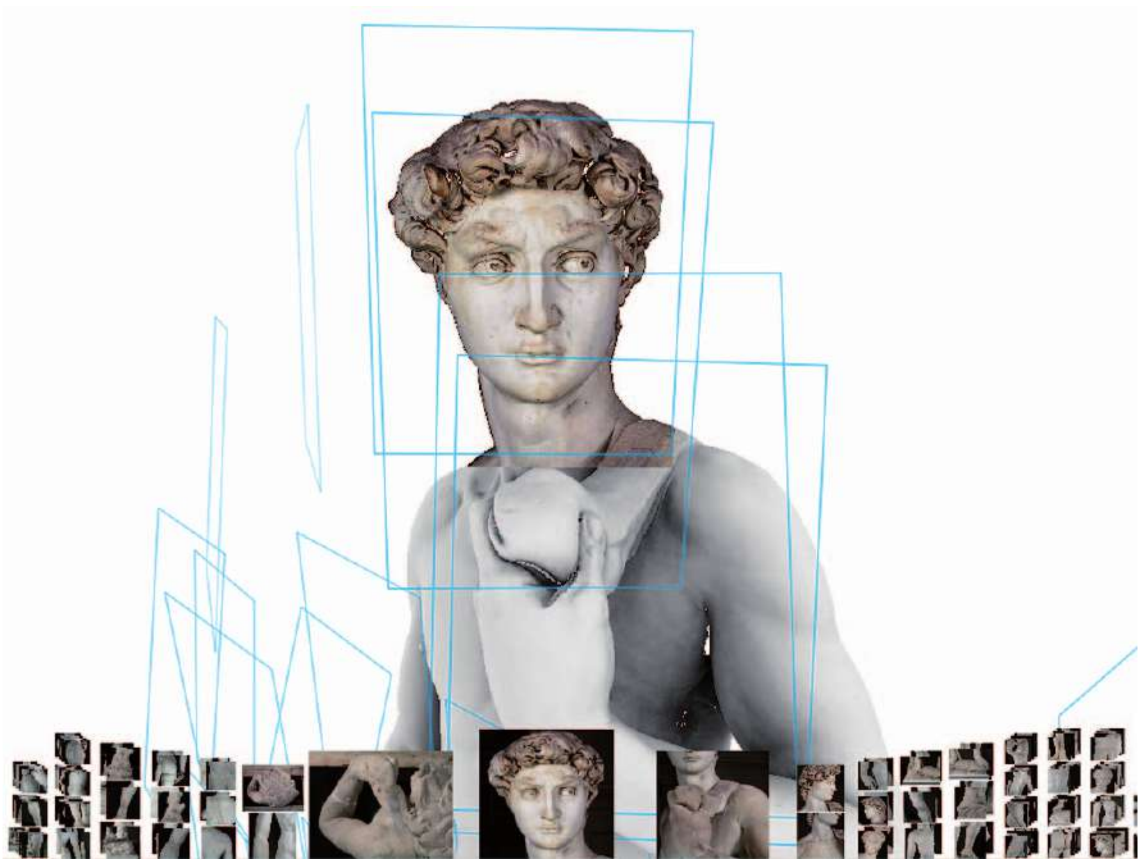


Figure 2.14: A resulting view of the PhotoCloud system. The framelets in blue around the 3d model depict the position of the front facing photos, and the navigation bar below. Only the central image is projected to the 3d model. Image extracted from BRIVIO *et al.* [9].

Chapter 3

Method

Our method aims at rendering in real time a geometric model with a set of associated photos, and applying static weights per pixel as well as dynamic weights per photo to generate color information on-the-fly. The idea is to not only attribute more weight to good pixels from each photo, but also decide in render time which are the best photos to use given a new viewpoint. Since a vertex may receive information from multiple photos, we compute a weighted average to determine the final color of each screen fragment related to each vertex and use a bilinear interpolation on texture coordinates to achieve the final color between vertices.

The weighted average has two main components: distance and angle weights for each camera computed for each new viewpoint; and precomputed depth, angle and border weights for each pixel of each photo, as in the work Masked Photo Blending [8] previously described in Section 2.1.

In this chapter we describe the method in a general framework, while in the next chapter implementation details are given. We start by pre-computing the pixel weights for each photo, as well as computing texture coordinates for each vertex for each photo, as will be described in Section 3.1. Whenever the viewpoint or direction changes, camera weights are recomputed (Section 3.2.1), and the model is rendered by interpolating the texture coordinates for every photo for each vertex, and computing the average weight for each interpolated fragment, as explained in Section 3.2.2. Figure 3.1 illustrates an overview of the main steps of our method.

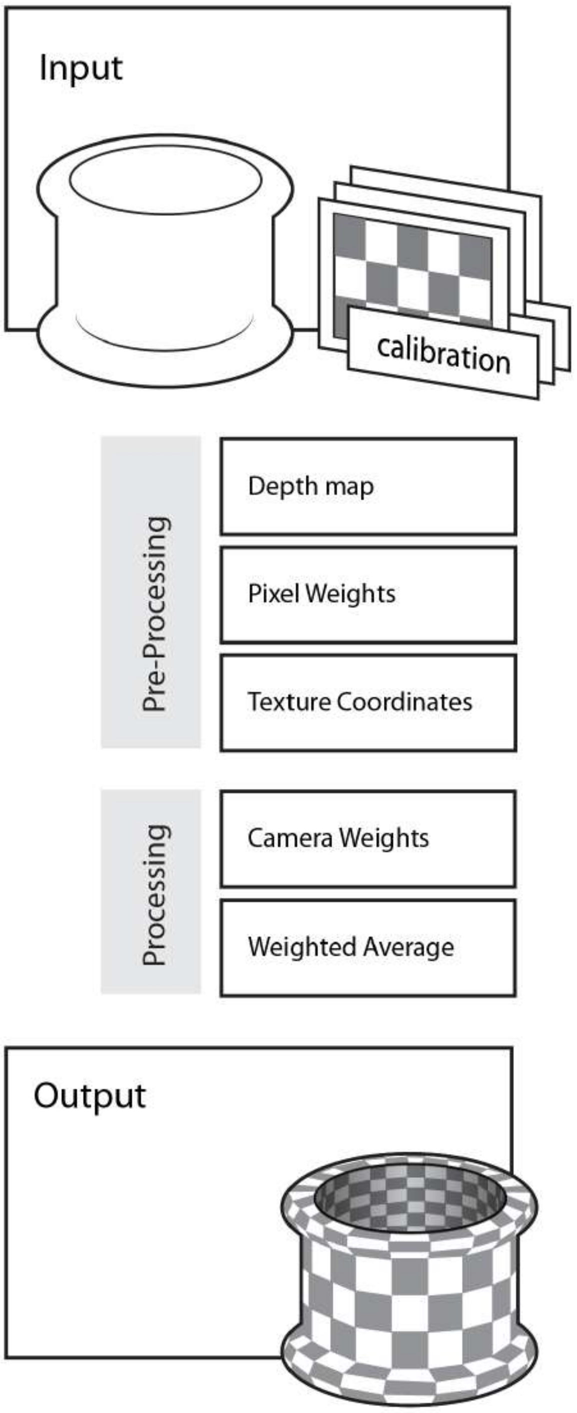


Figure 3.1: Method overview, from top to bottom: the input is the 3D geometry and a set of registered photos; for each photo a depth map, masks, and texture coordinates are computed in a pre-processing stage; during visualization camera weights are computed and an weighted average calculated for each fragment; the output is a colored 3d model.

3.1 Pre-Processing

During a pre-processing stage, we compute the following information: a depth map for each photo, pixels weights, and texture coordinates for each vertex. These three steps are better described next.

3.1.1 Depth Map

The depth map computation is straightforward, we render the model one time for every photo using its associated camera, and write to a new buffer the depth information of each pixel. We actually generate two depth maps, one with and the other without normalizing the depths in range $[0, 1]$, that will be used in different moments of the subsequent steps of the pre-processing stage.

3.1.2 Pixel Weights

To compute the weight of each pixel from each photo, we follow Callieri et al. [8] method with a few modifications. The depth mask and the angle mask are computed in the same way as the paper, while our implementation of the border mask will be described in the next Section 4.1.1.

Finally, we create one final texture for each photo, where the RGB channels contain the depth, angle, and border mask weights, respectively. Differently from the original method, we do not fuse the masks as a single value in order to visualize the effect of each mask individually in real time. But the fuse step could be trivially added to our method as well.

3.1.3 Texture Coordinates

In the original Masked Photo Blending method, a final texture is produced as the result. In our method, everything is dynamic, and the "final" texture is computed for each new viewpoint during a visualization session. Since we are introducing camera weights, the final weight of each pixel is modified in real time, so we cannot pre-compute a weighted color for each vertex. Thus, we need to know for each vertex its texture coordinates with respect to each photo to be retrieved in render time. Note, however, that a vertex is usually seen from only a subset of the photos, so it may not have an associated texture coordinate for every single one.

We again render the model for each camera, and compute the texture coordinates for each visible vertex. To determine if a vertex is visible we compare its projected depth with the unnormalized depth map for the current camera. If the vertex is visible, its texture coordinates for the current photo are stored as its normalized screen coordinates.

3.2 Render Time Processing

3.2.1 Camera Weights

For each camera we compute the distance and the angle weights. The first weight is simply the distance from the current viewpoint and the camera position, while the second weight is the angle between the view direction and the camera's view axis. Both weights are normalized in the range $[0, 1]$.

The reason behind these weights is very intuitive. The camera distance weight gives more priority to cameras near the viewpoint, while the camera angle weight gives more priority to cameras with the same direction as the view direction.

A non-linear weight can also be employed, to increase the weight of cameras nearby or with very close directions. So the normalized weights can be transformed using a non-linear function, such as an exponential factor.

3.2.2 Weighted Average During Renderization

During render time, for each new viewpoint we recompute the camera weights. We then follow the usual graphics pipeline to render the model. Each vertex is rendered with all its associated texture coordinates. At this point, we only render triangles whose three vertices have texture coordinates for at least one photo. The valid triangles are then interpolated generating fragments. Each fragment may receive texture coordinates for different photos. For each one we retrieve the texel color and its associate weights using bilinear interpolation, and multiply it by the camera's weight. Then, we average all texels modulated by their respective camera weights to compute the final fragment color. The final weighted average is given by:

$$C_{(x,y)} = \frac{\sum_{k=0}^{N-1} T_k(\tau_k(x,y)) * \Omega_k(\tau_k(x,y)) * \Theta_k}{\sum_{k=0}^{N-1} \Omega_k(\tau_k(x,y)) * \Theta_k} \tag{3.1}$$

where $\tau_k(x, y)$ is the interpolated texture coordinates (u, v) at fragment (x, y) for texture k, $T_k(u, v)$ is the color of texture T_k at position (u, v), $\Omega_k(u, v)$ is the multiplied depth, angle and border weights for pixel (u, v) in texture k, Θ_k are the multiplied distance and angle weights for camera k, and N is the number of valid textures projecting to fragment (x, y). Figure 3.2 illustrate the above concepts.

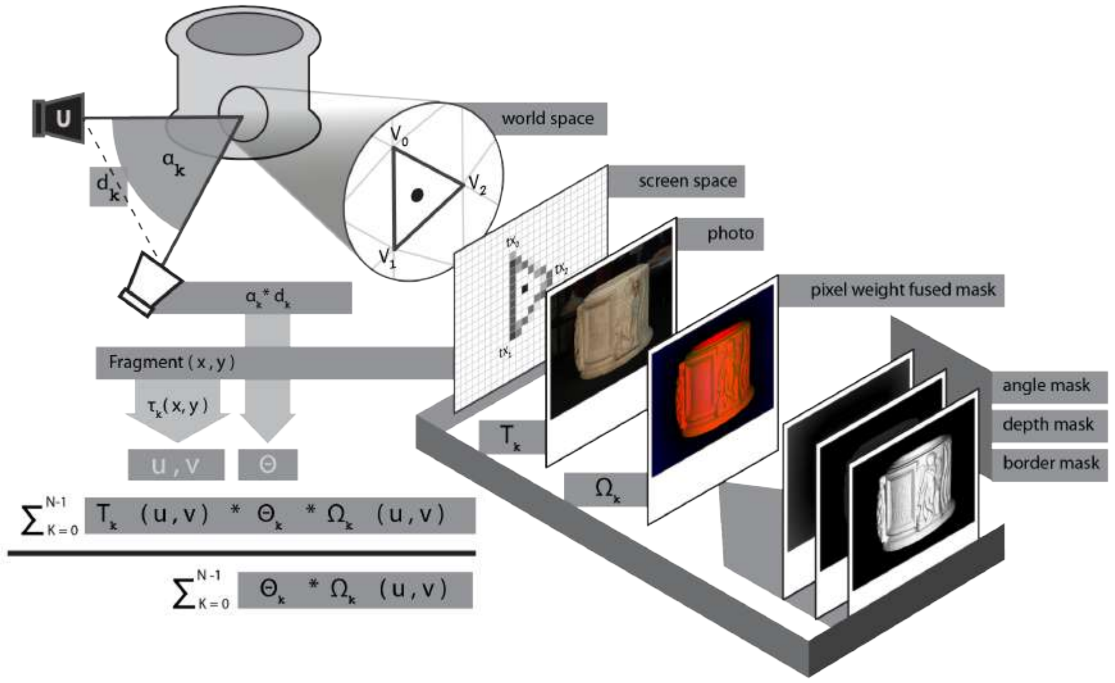


Figure 3.2: Visual illustration of Equation 3.1. For camera k and texel (u, v) final pixel weight $\Omega_k(u, v)$ is combined with camera weights Θ_k and multiplied by color $T_k(u, v)$.

Chapter 4

Implementation

In this chapter we describe the relevant implementation details of our method described in Chapter 3. In addition, we discuss alternatives to some steps that were tested but discarded in the final version.

Briefly, we start by registering each photo in regards to the 3D model and preprocessing the pixel weights (Section 4.1), and computing texture coordinates (Section 4.2). We then describe some implementation details about the real time render, such as the fusion of the masks in Section 4.3, camera and triangles discard criteria in Section 4.4 and Section 4.5, respectively, multi-pass approach in Section 4.7, and a field-of-view normalization strategy for the cameras in Section 4.6. In Figure 4.1 we highlight in which stage the implementation details are discussed in this chapter.

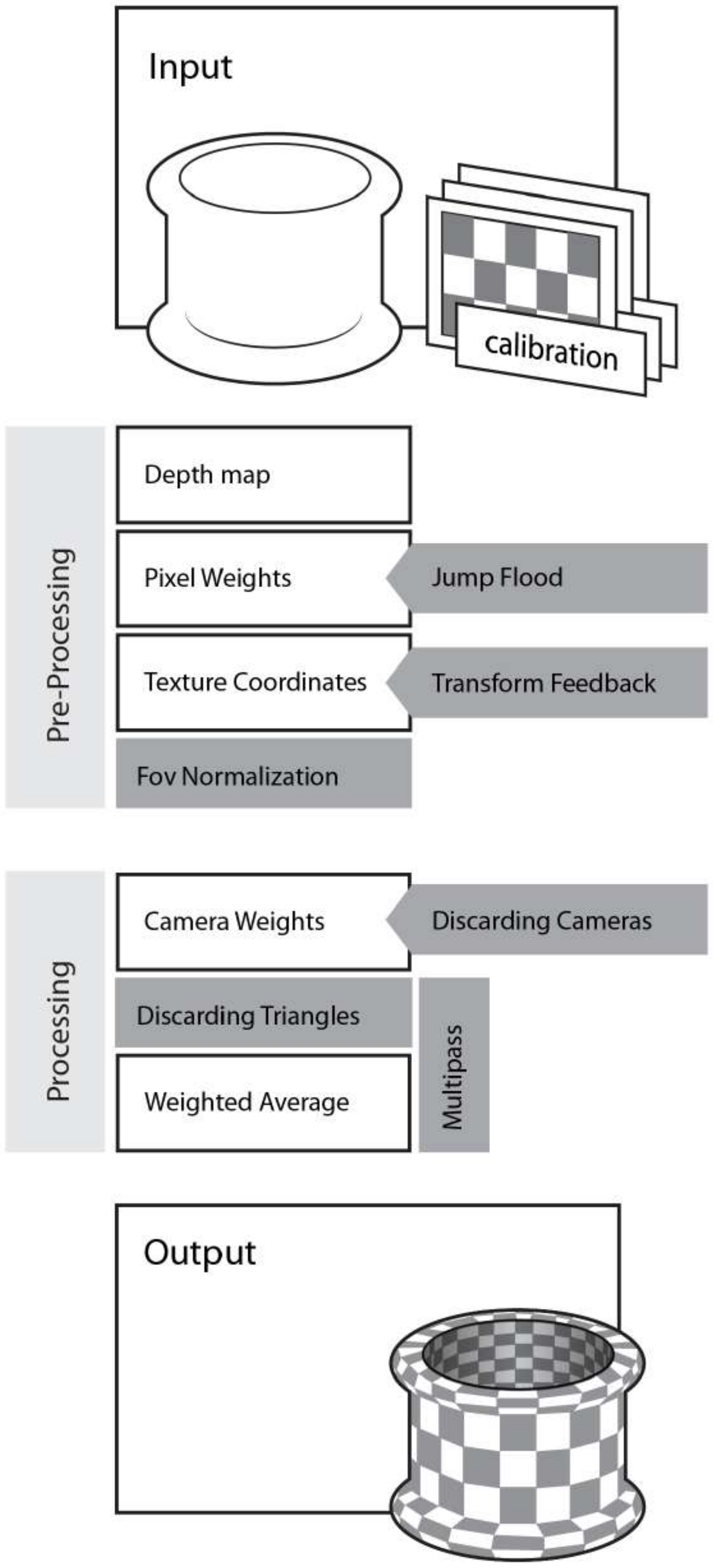


Figure 4.1: Implementation overview: in gray are the stages where implementation details are discussed in this chapter.

4.1 Photo Data Structure

Each photo in the dataset has the following information in our data structure:

- **image texture**: contains the original photo;
- **depth maps**: one texture with the normalized and another with the non-normalized depth map;
- **pixel weights texture**: contains the depth, angle and border weights for each pixel
- **camera weights**: distance and angle weights for the camera that are dynamically updated in regards to the viewpoint;
- **camera matrices**: intrinsic (FOV, pixel size, image size) and extrinsic (rotation and translation in world coordinates).

Figure 4.2 illustrates the contents of a photo in our data structure.

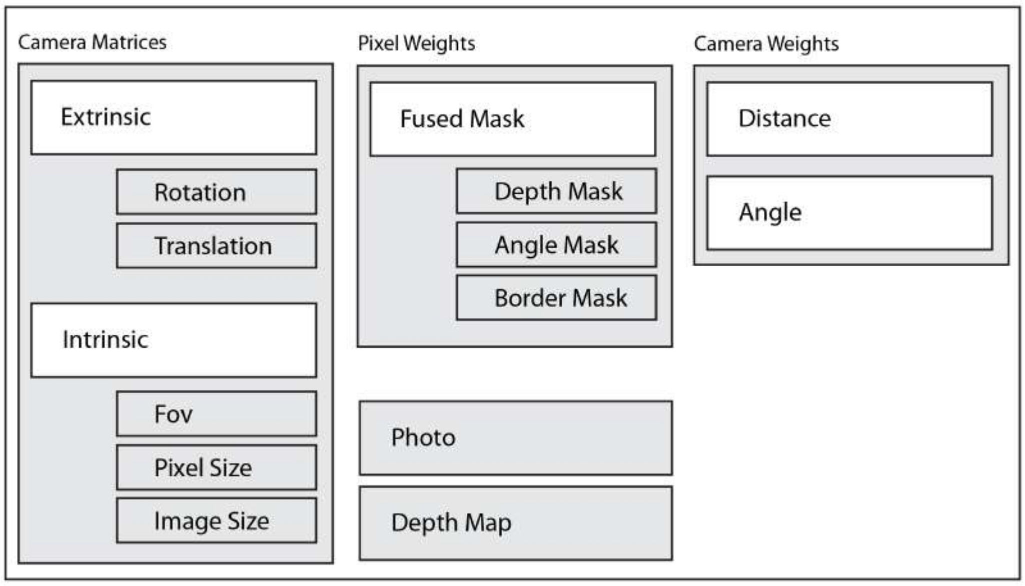


Figure 4.2: Diagram of the Photo data structure. Camera matrices are 4×4; pixel weights, photo and depth map are textures with the same resolution as the original photo; camera weights are scalar values.

4.1.1 Border Mask

As previously described, the depth and angle masks are computed as in the original work [8], using one render pass for each one. The border mask, however, cannot be computed in a single pass, since the computation of a distance field is necessary in addition to the border extraction step.

In the original paper, a Sobel filter is used to extract the borders. We tested with different kernel sizes for the Sobel and Laplacian filters. We had slightly distinct results for each dataset using different filters and parameters, but at the end we decided to use the Sobel filter with a 3×3 kernel size as default value that worked well for all examples.

After border extraction, it is necessary to compute the closest distance from each non-border pixel to a border pixel. In order to accelerate this step, even if this is still pre-processing, we implemented the Jump Flooding algorithm [12] to compute the distance field in parallel using the GPU.

4.2 Geometry Data Structure

For each vertex we need to know its texture coordinates for each image. We store all texture coordinates as vertex attributes. Note, however, that some vertices may not have valid texture coordinates for all images, since it may not be visible from some cameras. These invalid coordinates are tagged with values $(-1, -1)$. In total we have n vertex attribute arrays with m texture coordinates each one, where n is the number of photos and m is the number of vertices. Figure 4.3 illustrates the geometry data structure.

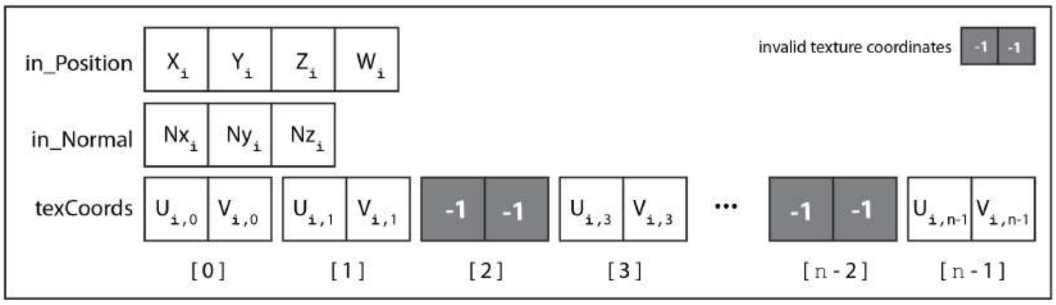

in_Position
X_i Y_i Z_i W_i
invalid texture coordinates -1 -1
in_Normal
Nx_i Ny_i Nz_i
texCoords
$U_{i,0}$ $V_{i,0}$ $U_{i,1}$ $V_{i,1}$ -1 -1 $U_{i,3}$ $V_{i,3}$ ... -1 -1 $U_{i,n-1}$ $V_{i,n-1}$
[0] [1] [2] [3] [n-2] [n-1]

Figure 4.3: Geometry data structure. For a camera where a vertice is not visible from its texture coordinates are set as $(-1, -1)$.

4.2.1 Computing Texture Coordinates

There are a few strategies to compute texture coordinates. We follow a simple projection strategy, where the model is rendered using the camera matrices of each photo, and registering the projected position for each vertex as its texture coordinates for the current image. Texture coordinates are normalized in the range $[0, 1]$.

Since we are not really interested in rendering the model, but just in projecting the vertices, we use OpenGL's *Transform Feedback* feature to write to the vertex attributes during the vertex shader stage, and discarding the rasterization step (see Figure 4.4). However, we have to deal with occlusion issues since we do not know which vertices are in fact visible at this point. We solve this by simply checking the projection position against the pre-computed depth map to discard occluded vertices.

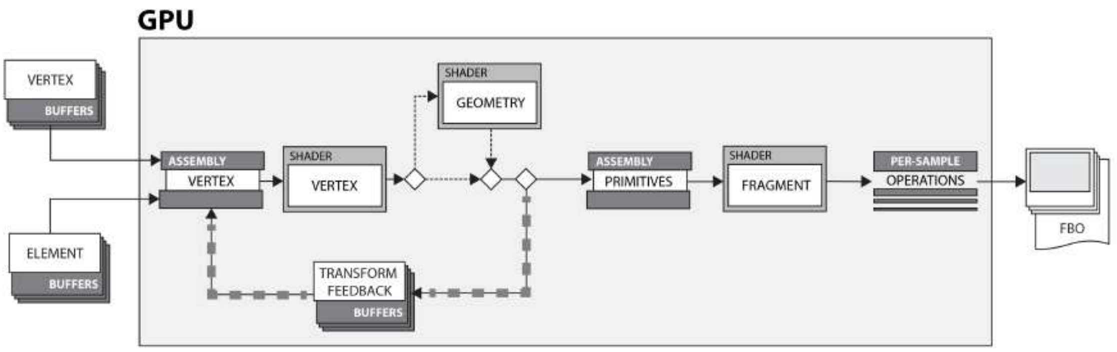


Figure 4.4: Transform Feedback is a rendering pipeline shortcut that allows skipping the rasterization and subsequent steps, and writing directly to a vertex attribute array.

4.2.2 Alternative Data Structure

We have also tested our method with an alternative way to store texture coordinates. Instead of creating vertex attribute arrays, we generate a single buffer (OpenGL's Shader Storage Buffer), that can be accessed from any shader stage. This is a single buffer that contains all texture coordinates for all textures. With this implementation we also load the textures using OpenGL's Texture Arrays, instead of separate single textures, thus removing the bound on the maximum number of texture slots, and possibly loading more images per pass. In practice, we observed an inferior performance, as will be illustrated in Chapter 5. Moreover, Texture Arrays has a limitation that all textures must have the same resolution. Even though that is true for all tested datasets, we would like to keep the method as generic as possible.

4.3 Real Time Weight Computation

In order to visualize the effect of each mask individually in real time, differently from the original method, we do not fuse the masks as a single value. So, we create one final texture for each photo, where the *RGB* channels contain the depth, angle, and border mask weights, respectively. Therefore, the proposed fusion only happens during the final rendering at the fragment level. In this step we also fuse the pixel weights with the camera weights, as depicted in Figure 4.5.

As with the pixel weights, we also do not pre-multiply the camera weights, so we can enable or disable each individual weight in real time to observe its influence.

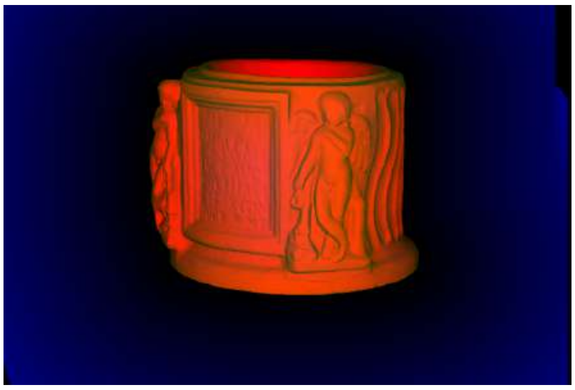


Figure 4.5: Real time weight fused mask, where the RGB channels are respectively: depth, angle and border weights.

4.4 Discarding Cameras

As mentioned in Section 3.2.1, it is possible to employ a non-linear decay on the dynamic weights to have an improvement on image detail. Therefore, while a few cameras have a gain in their participation in the fragment final color, others may have their relative contribution significantly reduced.

Furthermore, we also discard cameras using a maximum angle heuristic. If the view angle and camera normal are more than a given angle apart, we consider that they are in opposite directions and consequently can not share visual information. For our tests, we used a very conservative threshold of 150^o. Figure 4.6 illustrates this concept. We assemble a vector of indices indicating which cameras are valid, and pass it to the shader to avoid unnecessary texture accesses.

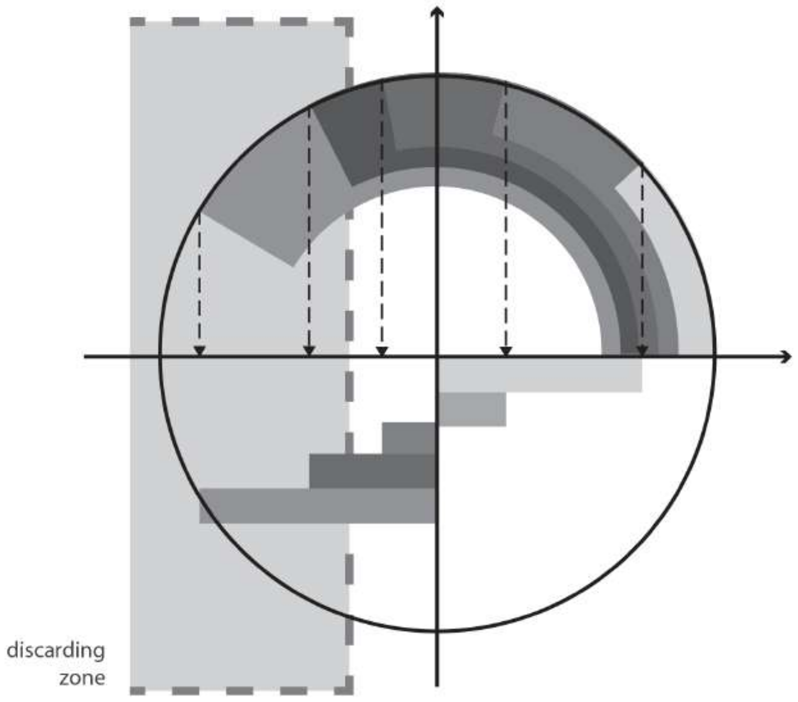
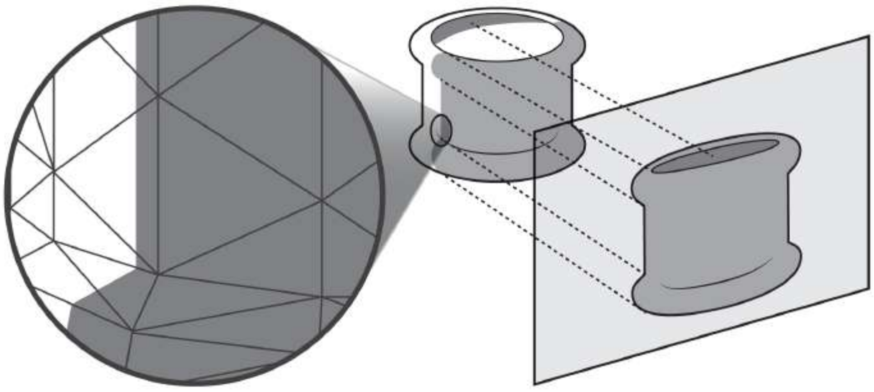


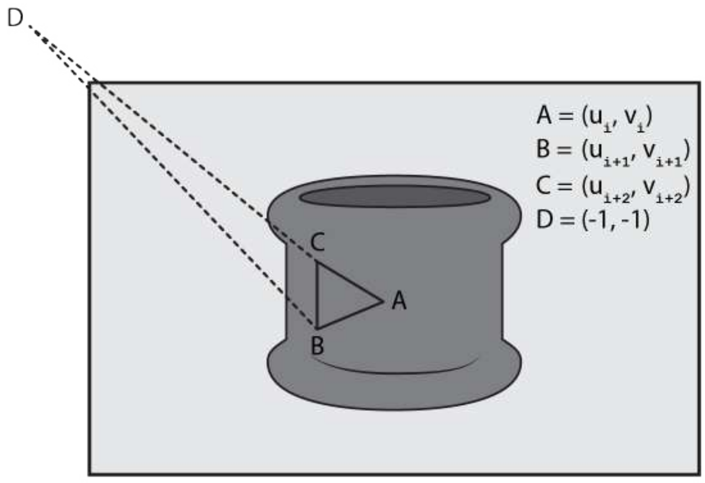
Figure 4.6: Cameras with angles above an empiric threshold are discarded. Current view direction is drawn in black.

4.5 Discarding Triangles

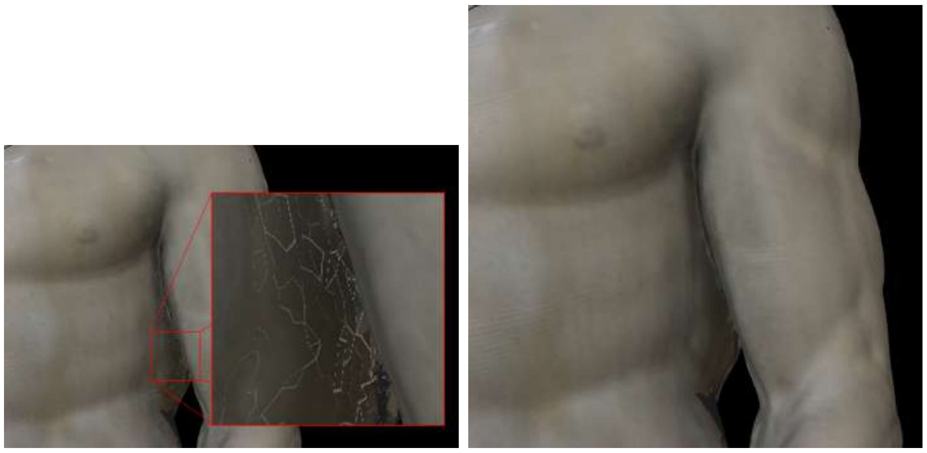
In render time, we check for each triangle in the Geometry Shader stage if for each texture its three texture coordinates are valid. This is a simple check since we flagged invalid texture coordinates as $(-1, -1)$. During the fragment stage, we only use in our weighted average textures whose all three coordinates are valid. This process is illustrated in Figure 4.7.



(a) For a given photo, some triangles might not have valid texture coordinates for all three vertices.



(b) These triangles are discarded during the weighted average procedure. In this case, vertex D does not have a valid texture coordinate, so the triangle CBD is discarded, while the triangle ABC has all three vertices with texture coordinates, so it is used.



(c) Biancone Dataset rendered without (left) and with (right) invalid texture coordinates test.

Figure 4.7

4.6 Normalizing Field-Of-Views

Considering a dataset where all photos have the same FOV, a good metric for the distance weight is the euclidean distance between the current viewpoint and the camera centers.

When cameras have different FOVs, however, the apparent distance from the camera to the model may be different for two cameras with same centers. For example, a camera with a wide FOV may seem farther away than a camera with a narrow FOV, even though both are located in the same position in space. Thus, when the viewpoint is close to model, we would like to receive more contribution from the camera with narrow FOV, for example.

To deal with this issue, we fix a common FOV value for all cameras. Each one is then translated along its view direction so that the projection is maintained with the new FOV. Figure 4.8 exemplifies the above concept, while Figure 4.9 illustrates the normalization with a real example.

Since all our models are normalized and centralized, we check the width of the view cone for each camera at the world space origin, i.e. the point $(0, 0, 0)$. Given the original FOV f of the camera, and the distance d from the camera to the origin, the width is computed as:

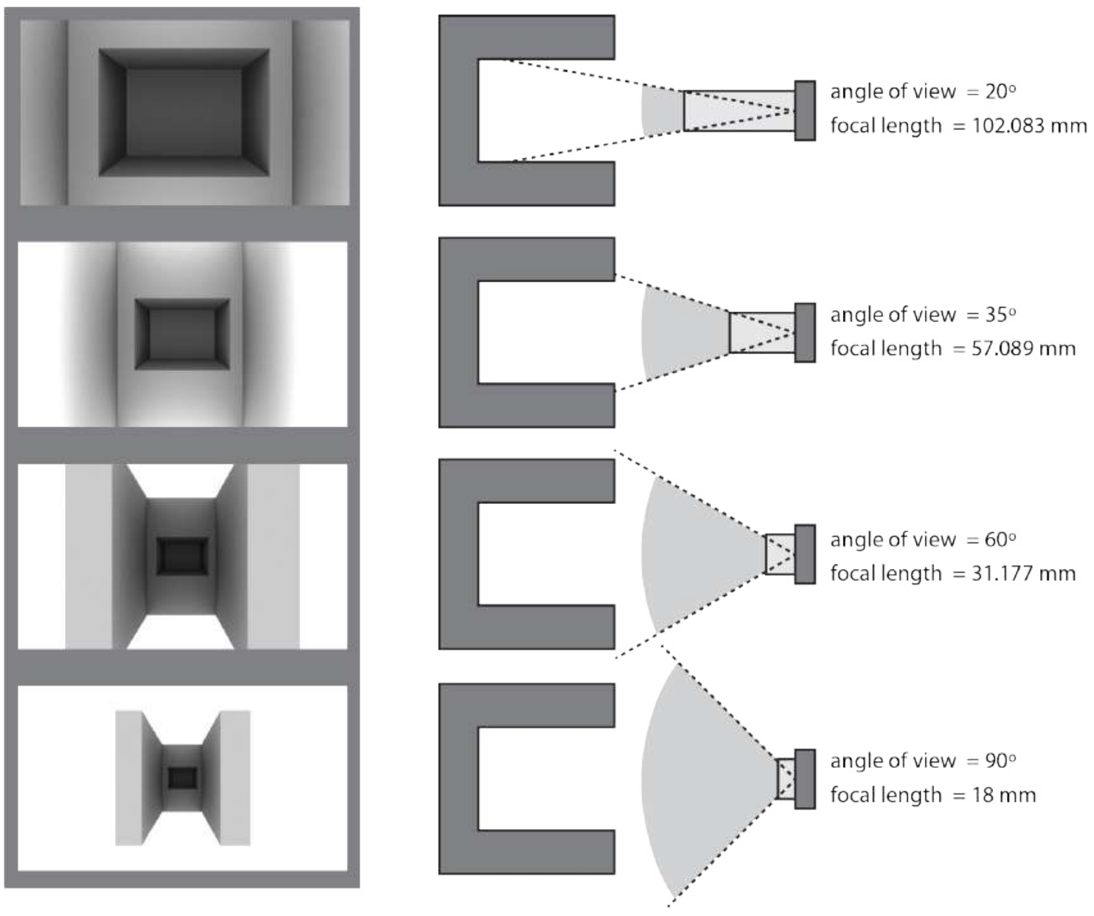
$$w = 2d \tan(f/2). \tag{4.1}$$

Then new distance given a target FOV f', is computed as:

$$d' = \frac{w}{2\tan(f'/2)}, \tag{4.2}$$

and finally, the new center for the camera is:

$$c' = d' \frac{c}{d}. \tag{4.3}$$



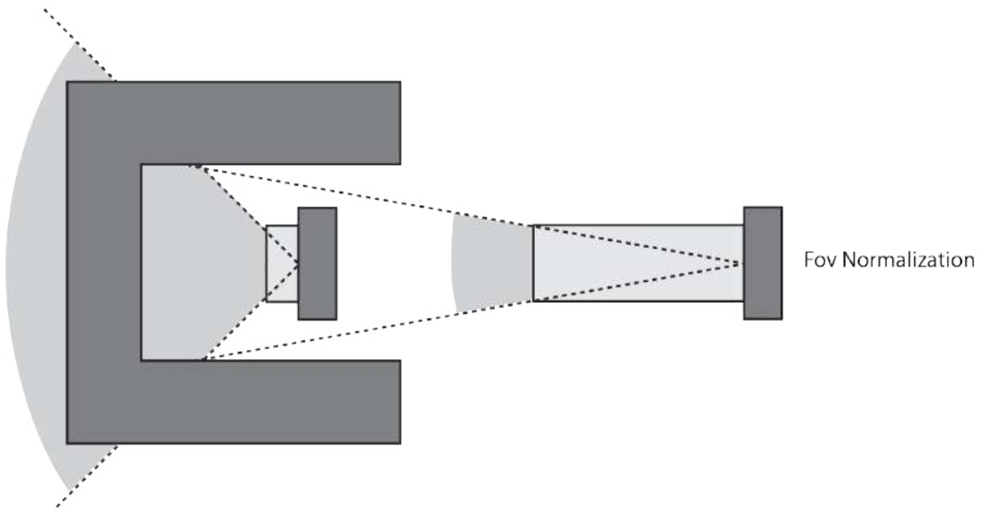
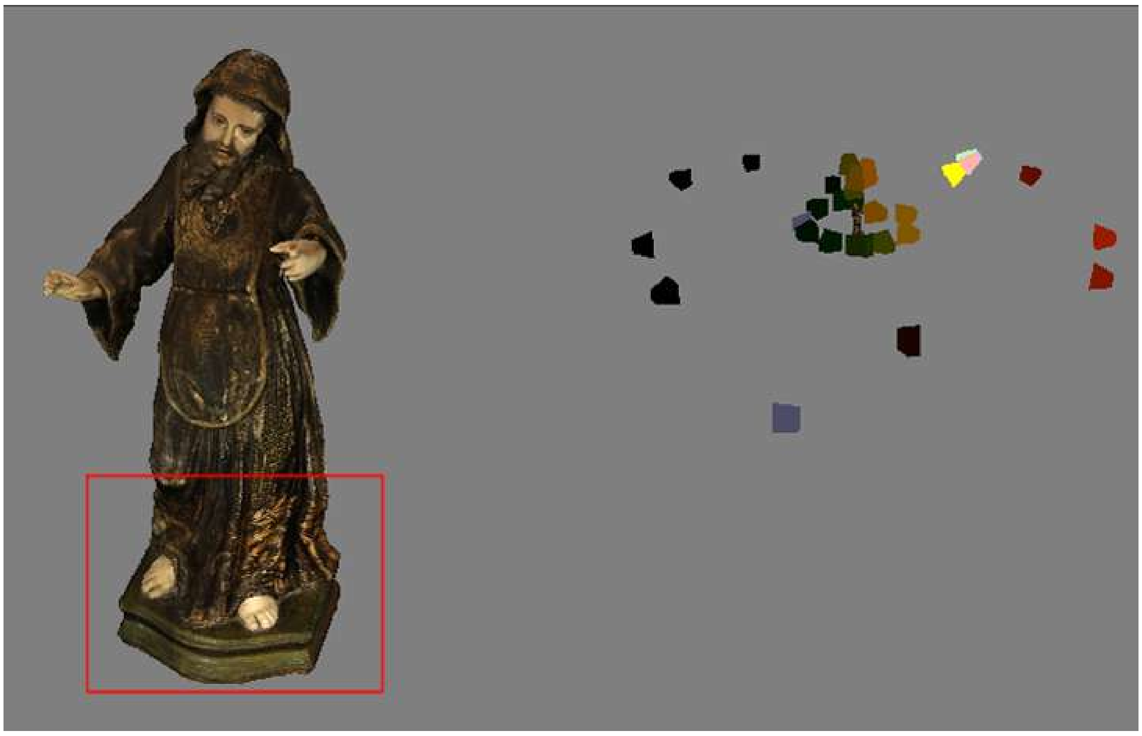
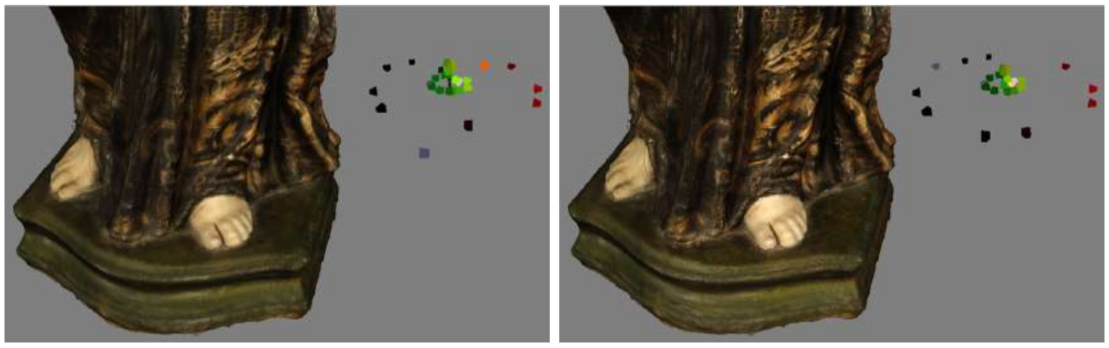


Figure 4.8: The top four images show the view from a camera with fixed position and direction, but varying the field of view. Note how narrower FOV are similar to approaching the camera to the model, while wider FOVs are similar to placing the camera farther away. In the bottom row an example of two cameras with same view direction but different positions to compensate different FOVs.



(a) The red box shows a region where the color projection was improved after normalizing the FOV for all cameras. On the right are shown the cameras in respect to the model.



(b) non-normalized (left), normalized (right)

Figure 4.9: A detail view of the lower part of the statue, note how the texture is sharper since a close-up camera has received more weight after normalization. The red non-normalized camera from the left image, after normalization, is very near to the current view position, thus increasing its weight.

4.7 Multi-Pass Render

When dealing with datasets with a large number of photos, we have to optimize the number of textures being processed in each render pass. Many GPUs impose a hard limit on this number, and rendering the maximum possible number of textures in each pass may not be the most efficient approach.

We thus employ a multi-pass strategy, where a fixed maximum number of textures is used in each pass. At the end of each pass we store the numerator and denominator of Equation 3.1, and pass it to next pass. These values are accumulated after each pass, and the division occurs only during the final one.

In practice, each render pass writes to a framebuffer texture, where the RGB channels are the accumulated pixel colors multiplied by the pixel and camera weights, and the α channel stores the accumulated weights. The final pass renders the result directly to the screen buffer. The multi-pass scheme is illustrated in Figure 4.10.

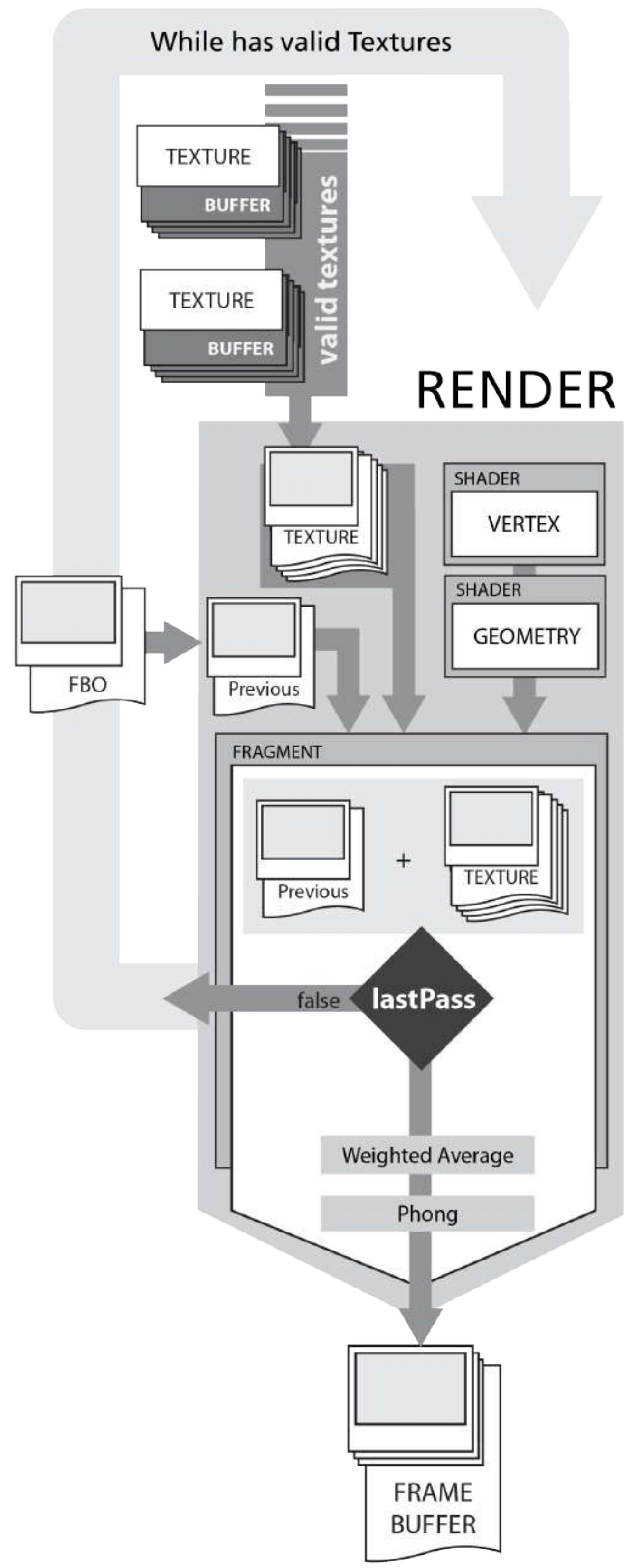


Figure 4.10: A multi-pass approach is used since GPUs impose a hard limit on the number of available textures units. For each pass at most k textures are allocated from the list of non-discarded cameras. Inside the shaders the result from the current pass is accumulated with the one from the previous pass, and written to a FBO texture. For the last pass the weighted average is finalized and Phong illumination applied.

Chapter 5

Results

We have performed tests with a few datasets varying many parameters, to better analyze the proposed method. A description of the computer used details is given in Table 5.1. A description of the datasets is given in Table 5.2. Note that the only difference between Urn High and Low is the images resolution.

Processor	Intel(R) Core(TM) i7-3770
CPU	3.40GHz
Cores	4
Mem	16 GB
GPU	GeForce GTX 660/PCIe/SSE2
OpenGl	4.4.0 NVIDIA 340.101

Table 5.1: Computer Specifications

Name	#verts	#photos	width	height
Biancone	375923	40	1728	1152
Duomo	644888	50	1936	1296
Saint	892263	25	1162	778
Urn High	911883	10	3872	2592
Urn Low	911883	10	968	648

Table 5.2: Datasets: Biancone and Duomo datasets were generously made available by the Visual Computing Group (CNR-Pisa); Urn and Saint datasets were scanned and photographed by the LCG group from UFRJ. The Urn is a piece from the National Historical museum's collection, while the Saint is from a personal item from the author's family.

In Figure 5.1 a general view of the cameras positions is given for each dataset. Some illustrative images of the datasets are shown in Figures 5.2, 5.3, 5.4, and 5.5. For all screen-shots a mini-view is shown on the top-right corner depicting all cameras in the dataset, and with a color-map representing the weights, where the green channel is the distance weight and the red channel is the angle weight. The more saturated the color, the higher the weight. Yellow cameras, for example, have high distance and angle weights. Figures 5.6, 5.7, 5.8, 5.9, and 5.10 show some comparisons with the texture maps produced by the method of CALLIERI *et al.* [8].

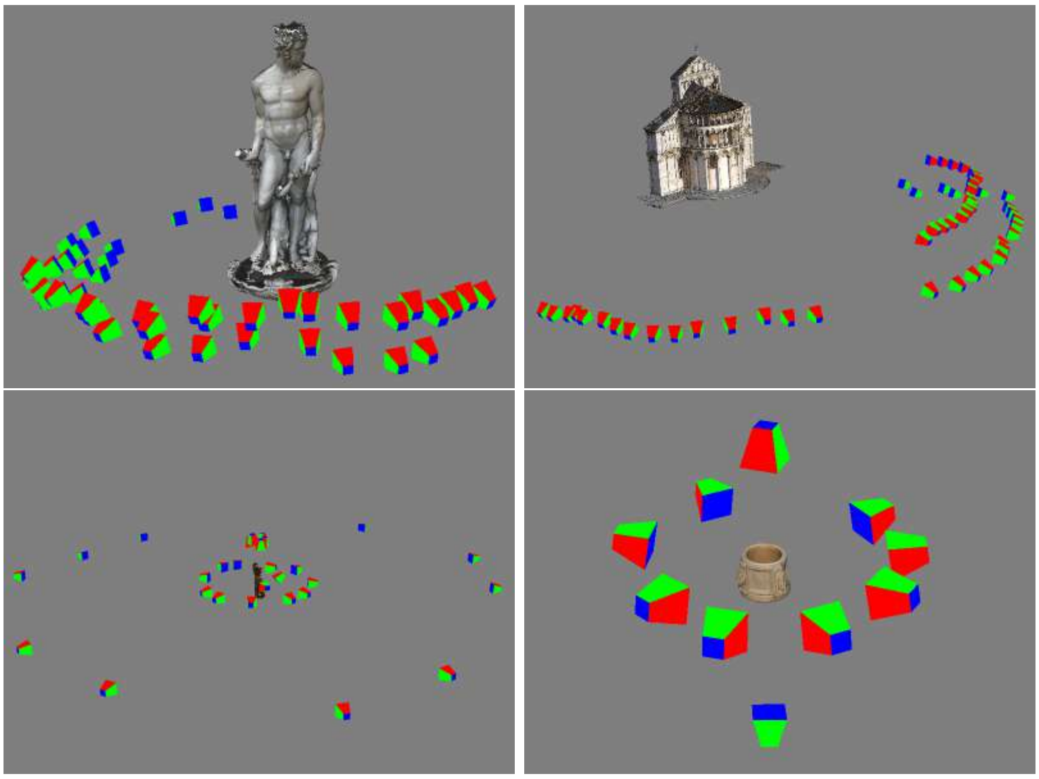


Figure 5.1: A general view of the cameras positions for each dataset. The FOV is normalized for all cameras.

In the following sections we show statistics for the datasets varying some parameters. For all experiments, we show results in two versions, using *vertex attributes* and using *storage buffer*. Timings were taken using a predetermined camera path in order to replicate the experiments changing only the parameters.

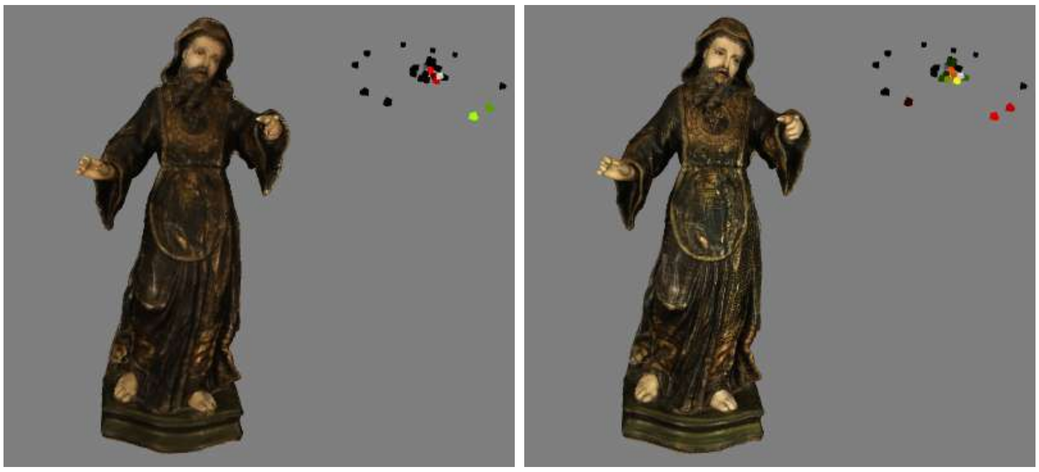


Figure 5.2: In the left image the weights for a camera far away are frozen and the camera is positioned close to the model, to illustrate the difference from using the real weights for that same position (right image).

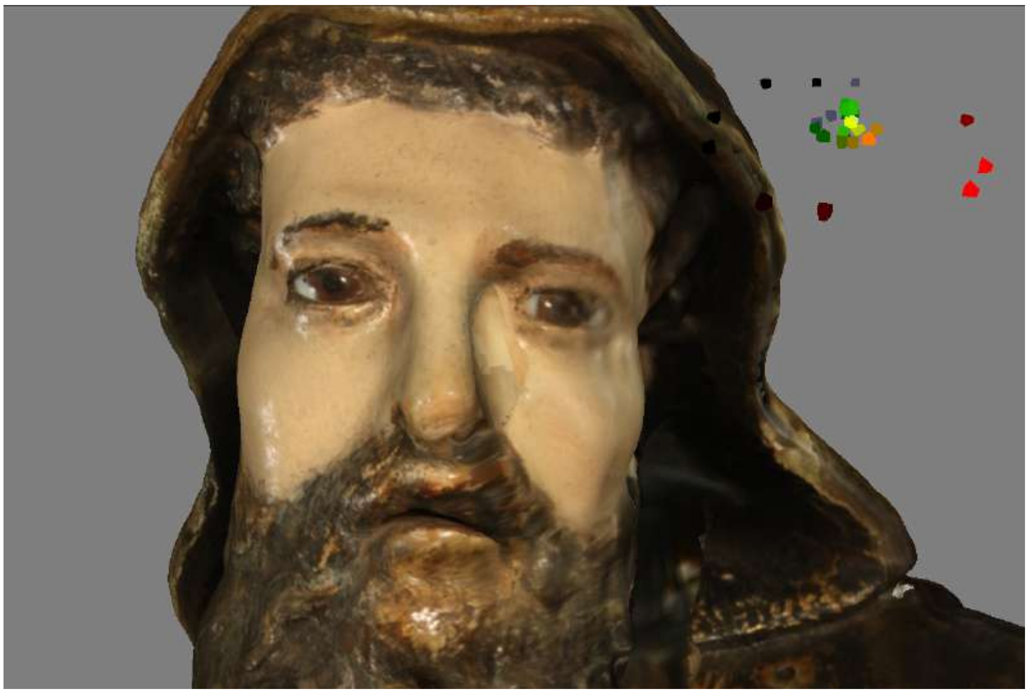


Figure 5.3: In a close-up view, weights are given for the photos taken from near the object or with zoom. Note, however, that the left side of the face was better sampled than the right, causing some artifacts on the right side.

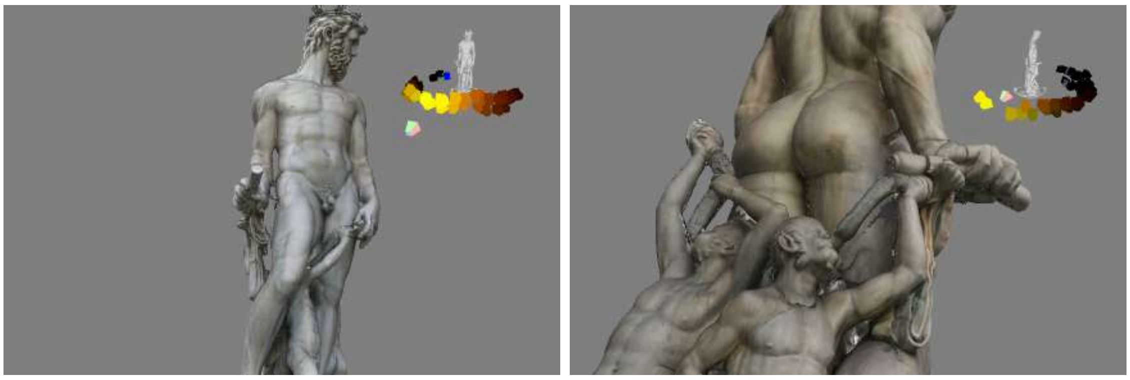


Figure 5.4: Some exemplary views of the Biancone dataset.

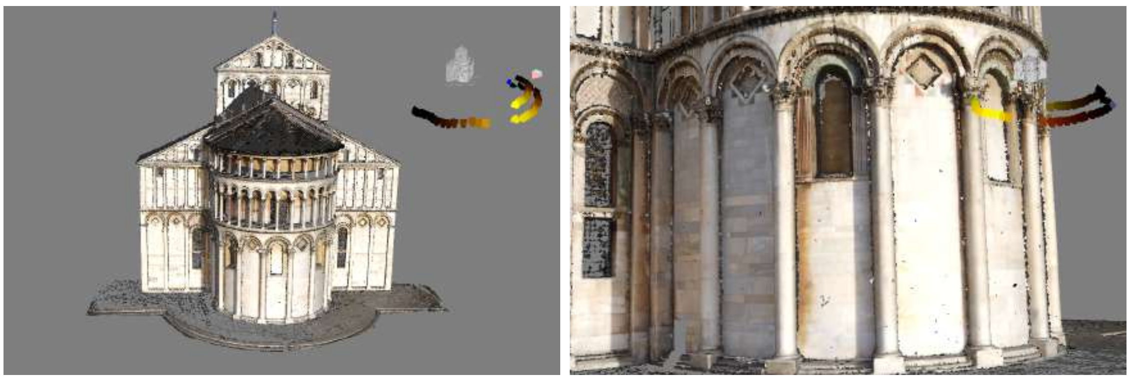


Figure 5.5: Some exemplary views of the Duomo dataset.

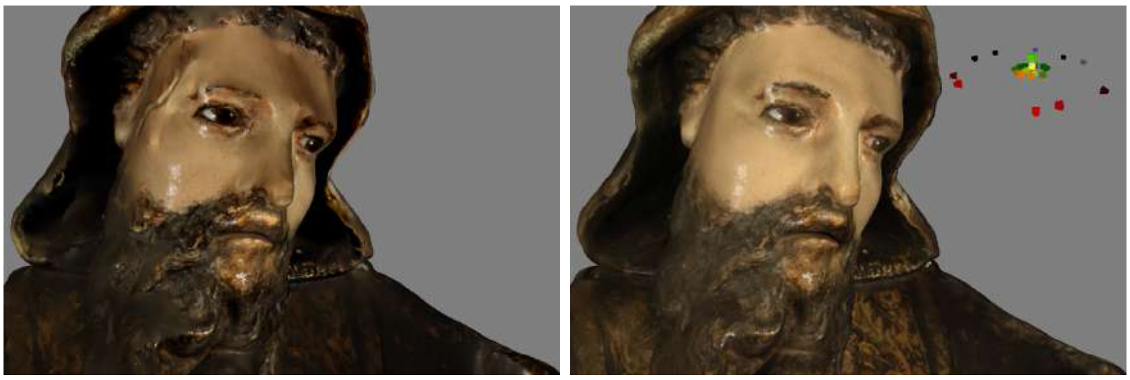


Figure 5.6: Traditional texture mapping (left) and our proposed method (right). Note how our method produces a much sharper visualization since only a few images have significant weight for the view position.

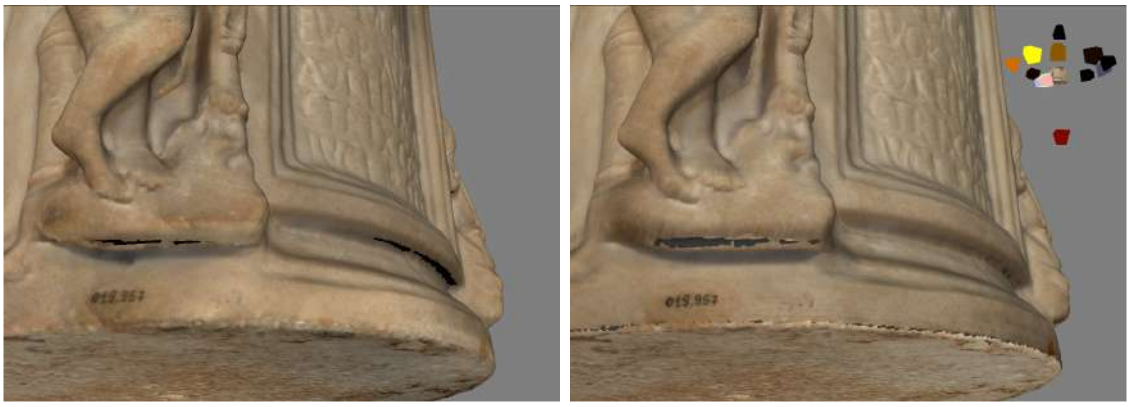


Figure 5.7: Traditional texture mapping (left) and proposed method (right). Note the sharper numbers at the base of the urn with our method.

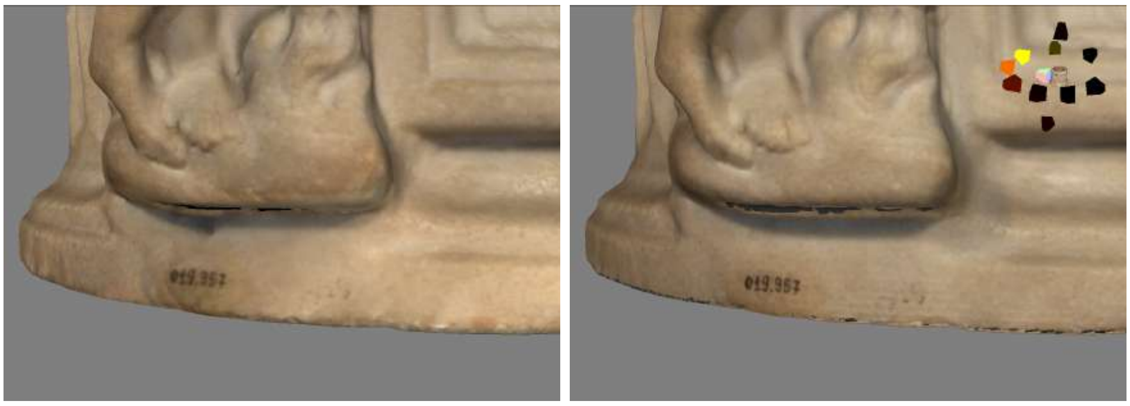


Figure 5.8: Traditional texture mapping (left) and proposed method (right)

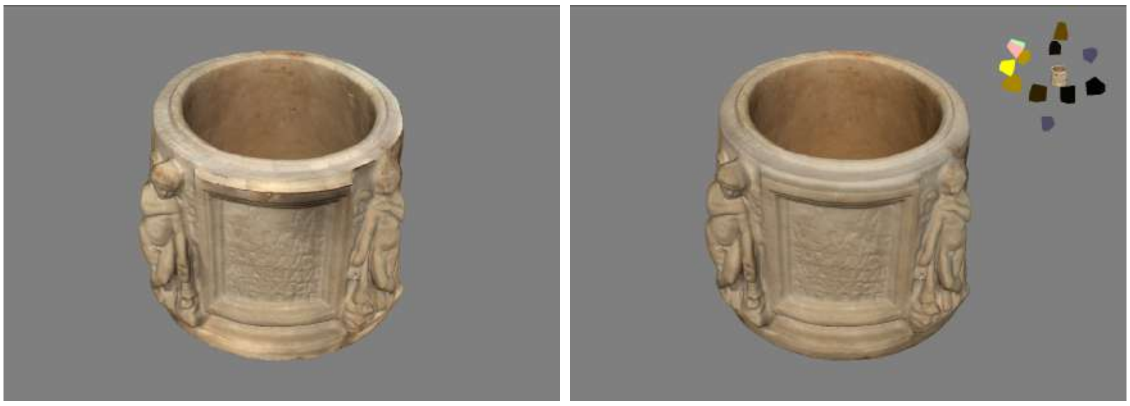


Figure 5.9: Traditional texture mapping(left) and proposed method (right). Illumination variations are better handled with our method, since oblique cameras receive very low weights. Note that artifacts are visible near borders in traditional texture mapping

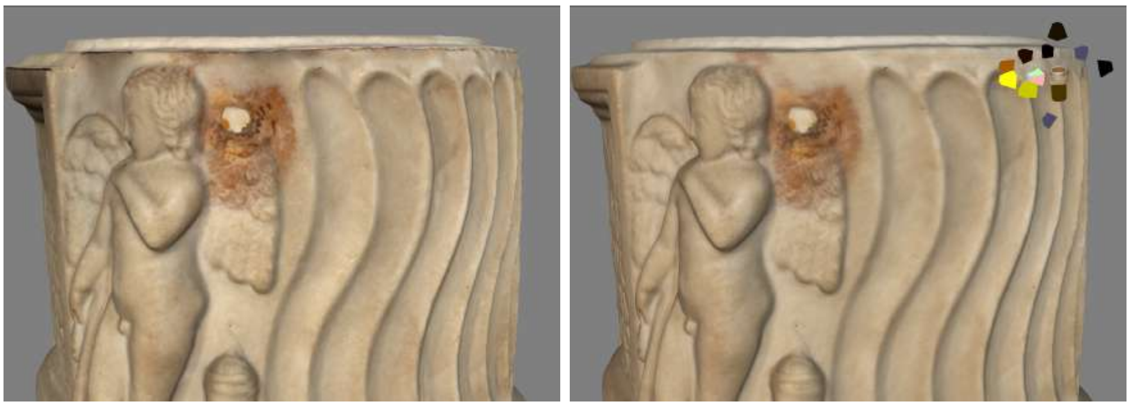


Figure 5.10: Traditional texture mapping (left) and proposed method (right). In this case texture mapping produced a sharper image, since only one photo was used for this part of the final texture, thus avoiding blurring artifacts from small misalignments.

5.1 Cameras Per Pass

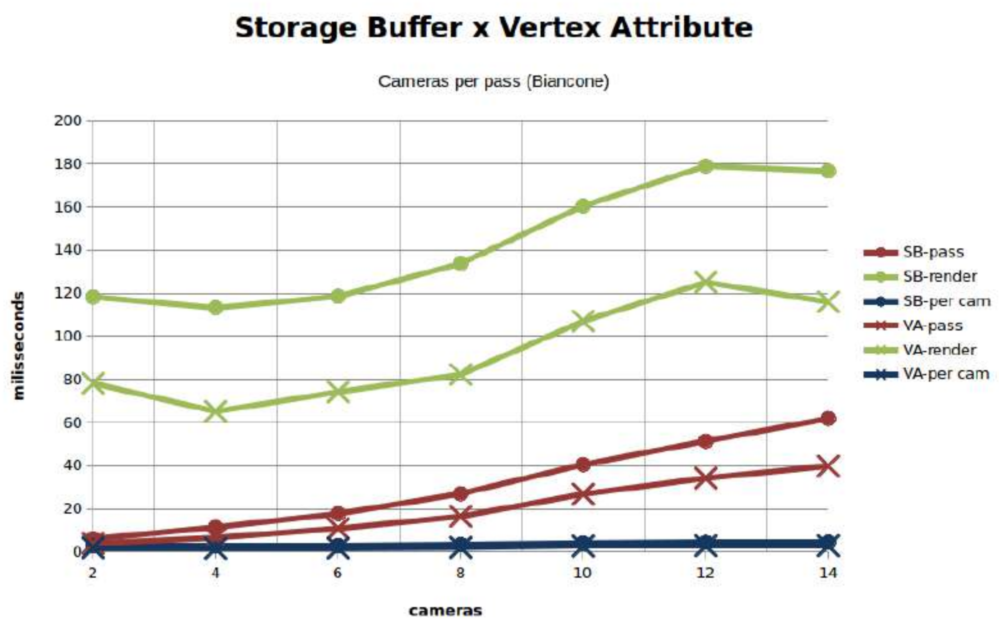
In this section we show the results when varying the maximum number of photos for each render pass. A summary of the optimal parameters are specified in Table 5.3.

Name	best #photos	best #passes	Fig.
Biancone	4	10	5.11
Duomo	4	13	5.12
Saint	4	7	5.13
Urn High	4	3	5.14
Urn Low	4	3	5.15

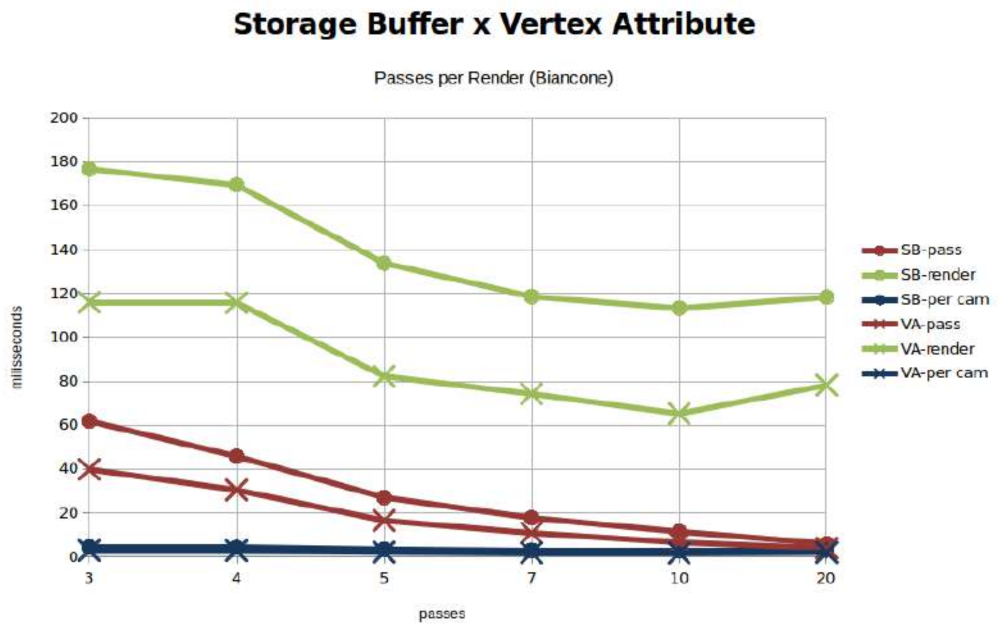
Table 5.3: Number of photos that results in best performance for each dataset.

As can be noted from the figures, for all datasets, the best performance was achieved when rendering four images per pass. This result is achieved regardless of the total number of passes. For example, for the Biancone it is achieved with 10 passes, while for the Urn it is achieved with 3 passes. This observation holds for both implementations, using *vertex attributes* or *storage buffers.*

One possible explanation for this optimal number of images per pass may be exposed in terms of balancing the data transfer load in the GPU. Four images probably results in the best balance between texture fetches and local operations that do not depend on memory access.

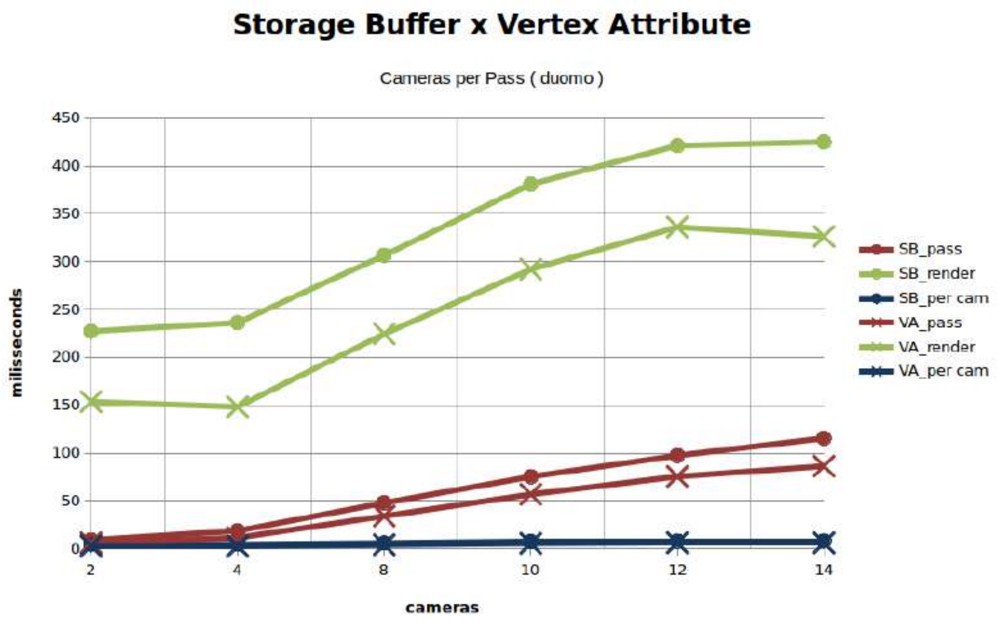


(a) Varying number of maximum photos per render pass.

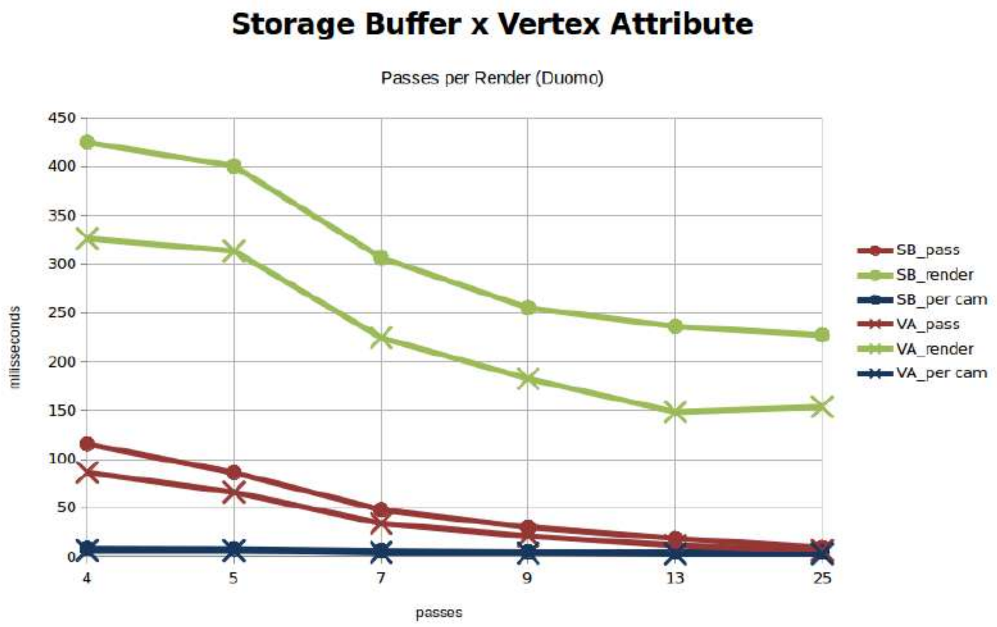


(b) Varying number of passes.

Figure 5.11: Biancone dataset.

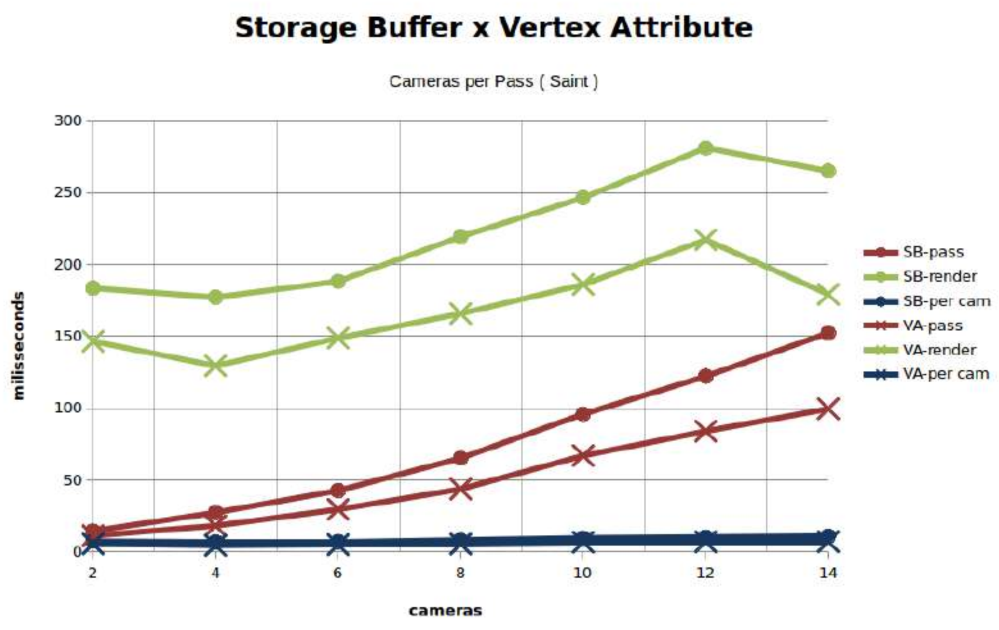


(a) Varying number of maximum photos per render pass.

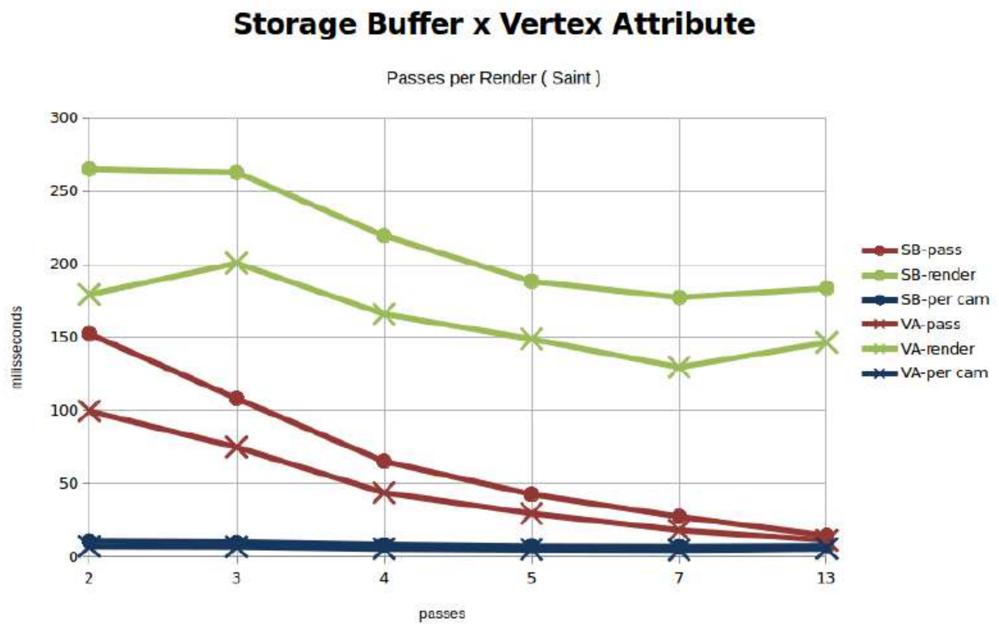


(b) Varying number of passes.

Figure 5.12: Duomo dataset.

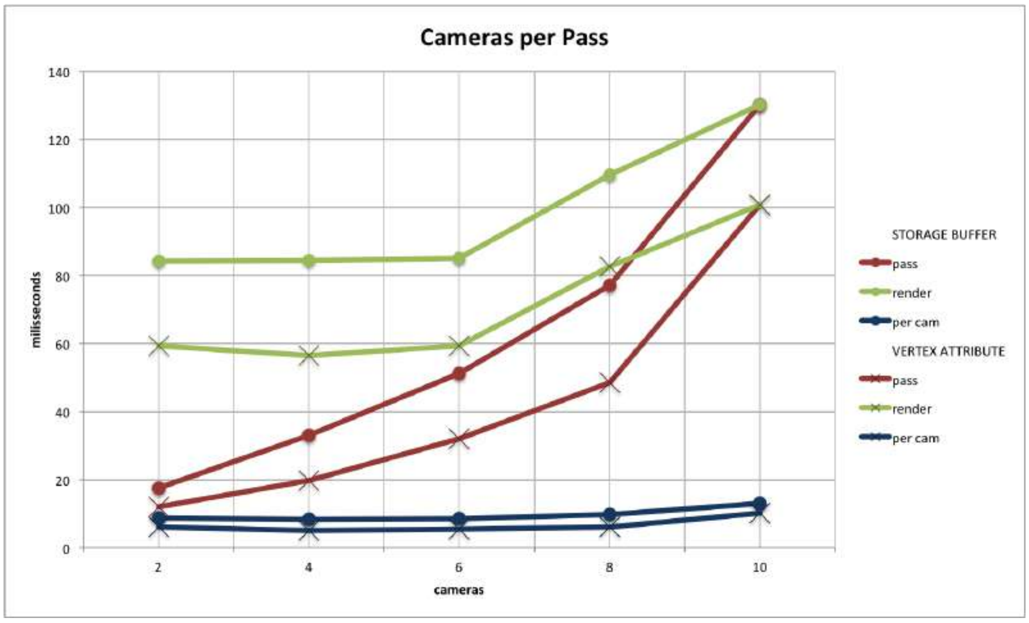


(a) Varying number of maximum photos per render pass.

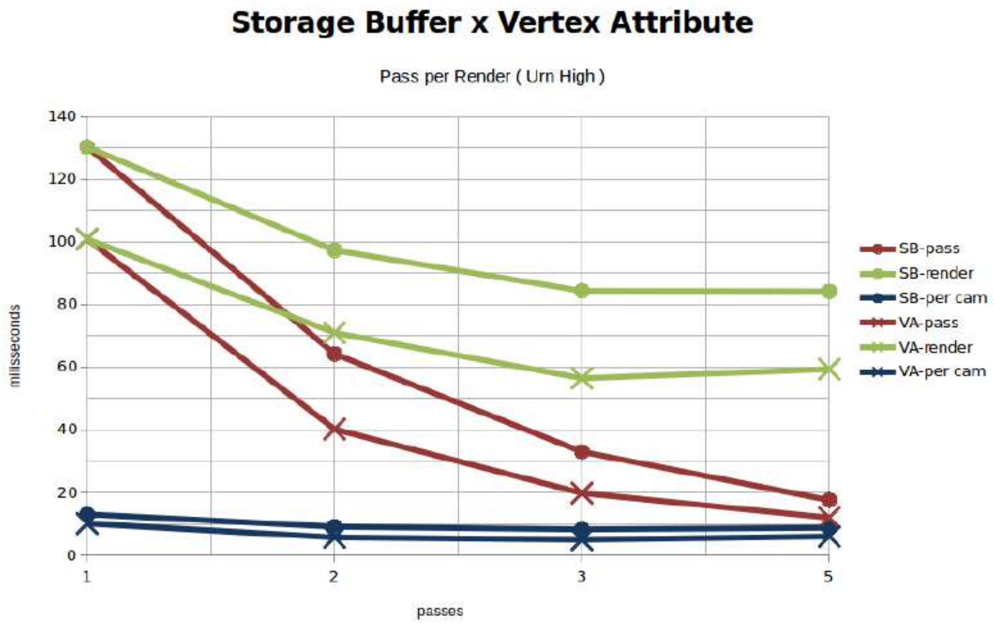


(b) Varying number of passes.

Figure 5.13: Saint dataset.

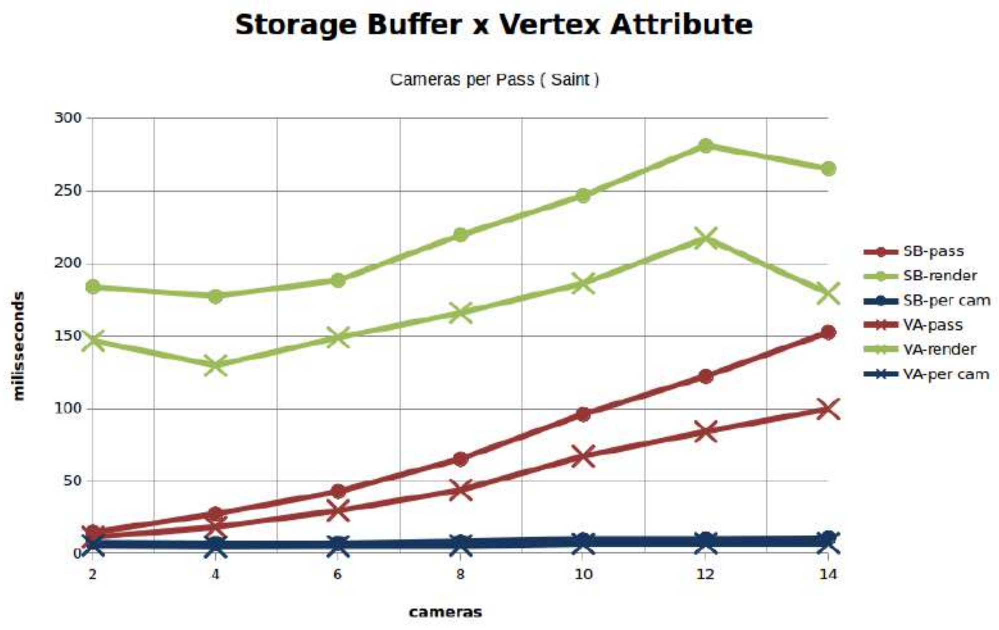


(a) Varying number of maximum photos per render pass.

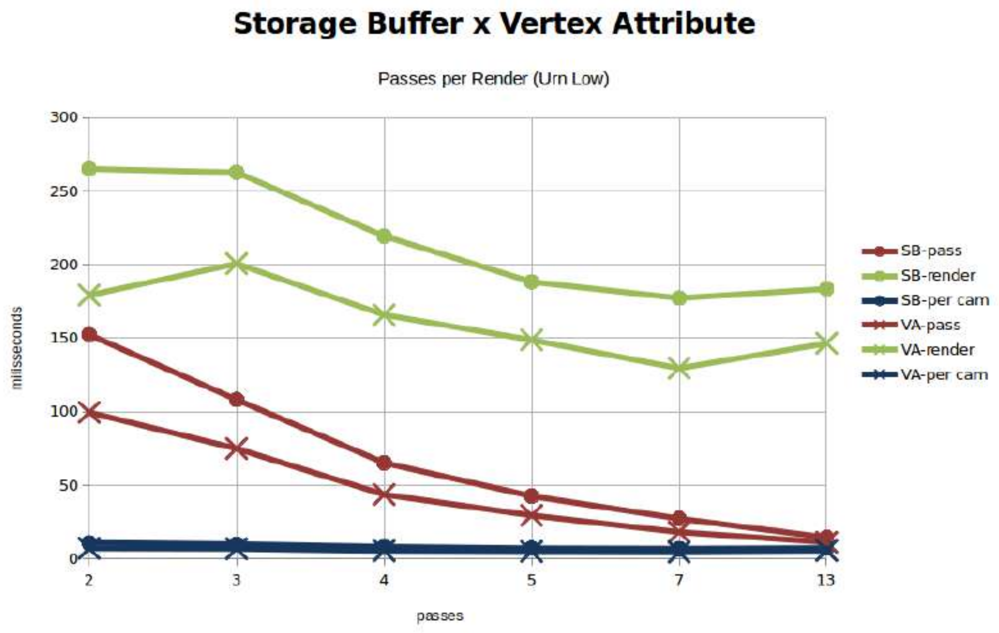


(b) Varying number of passes.

Figure 5.14: Urn high resolution dataset. Note that the dataset only has 10 photos, so it is possible to render everything in one pass.



(a) Varying number of maximum photos per render pass.



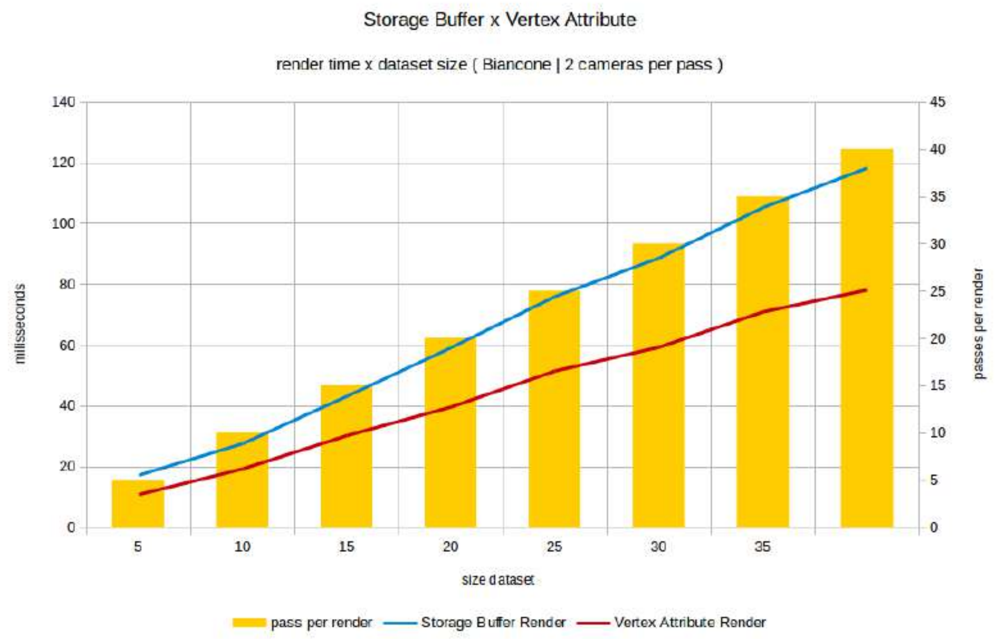
(b) Varying number of passes.

Figure 5.15: Urn low resolution dataset. Note that the dataset only has 10 photos, so it is possible to render everything in one pass.

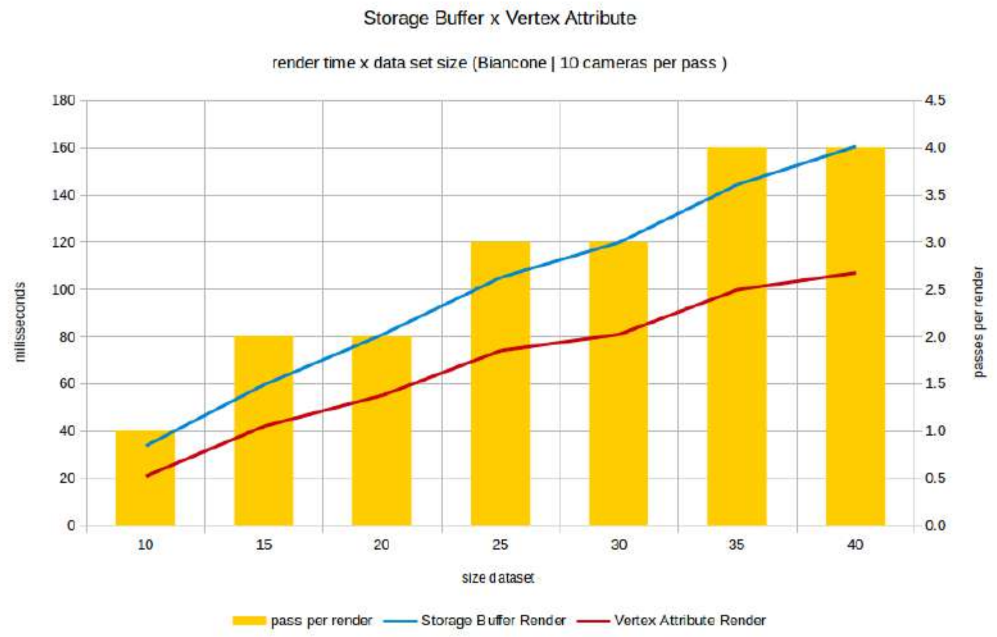
5.2 Photo Set Reduction

The next experiments were realized by reducing the total number of photos in each dataset. This allows to observe the behavior of having less photos without changing the geometry, particularly the number of vertices. We show total timings results for the Biancone dataset in Figure 5.16 and for the Duomo dataset in Figure 5.17, since they had the largest set of photos. In Figures 5.18 and 5.19 average timings per pass are illustrated.

We can observe a practically linear behavior in regards to the total number of photos, and the average time to complete each pass does not vary significantly in regards to the total number of necessary passes. Note that the maximum variation when analyzing the average timings was around 15ms, which is much less than the variations when varying the number of cameras per pass as discussed previously in Section 5.1. This was the expected behavior since only the total number of passes is changing between the original and the reduced sets.

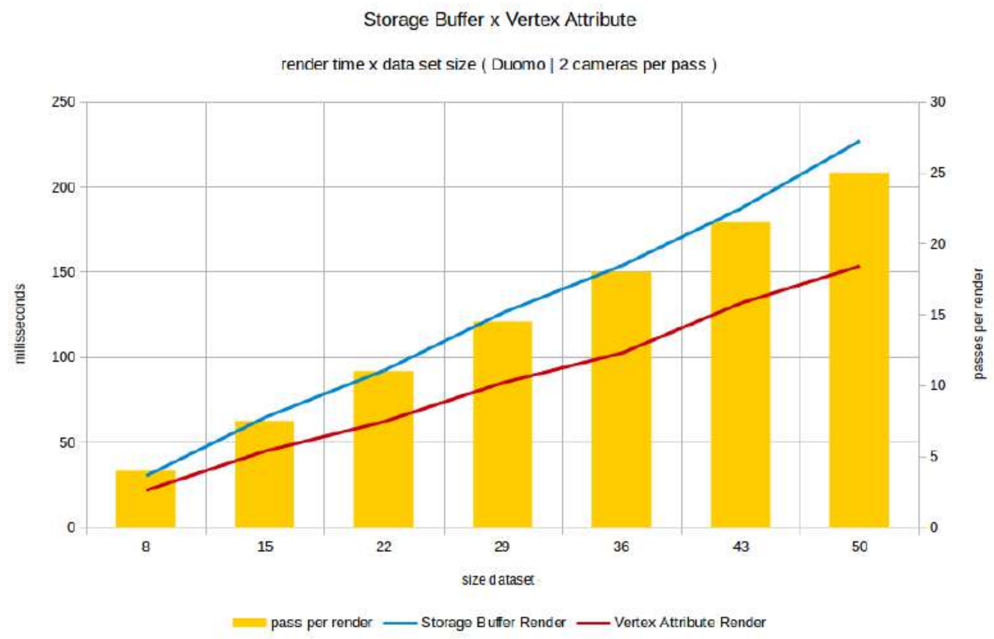


(a) Maximum two cameras per pass.

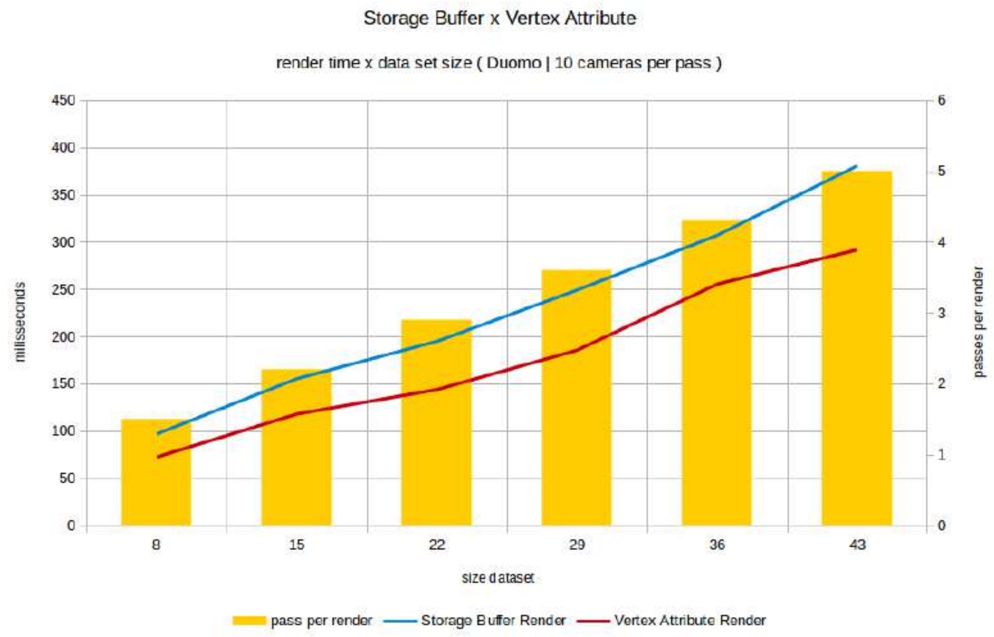


(b) Maximum ten cameras per pass.

Figure 5.16: Biancone model reducing the total number of photos in the dataset.

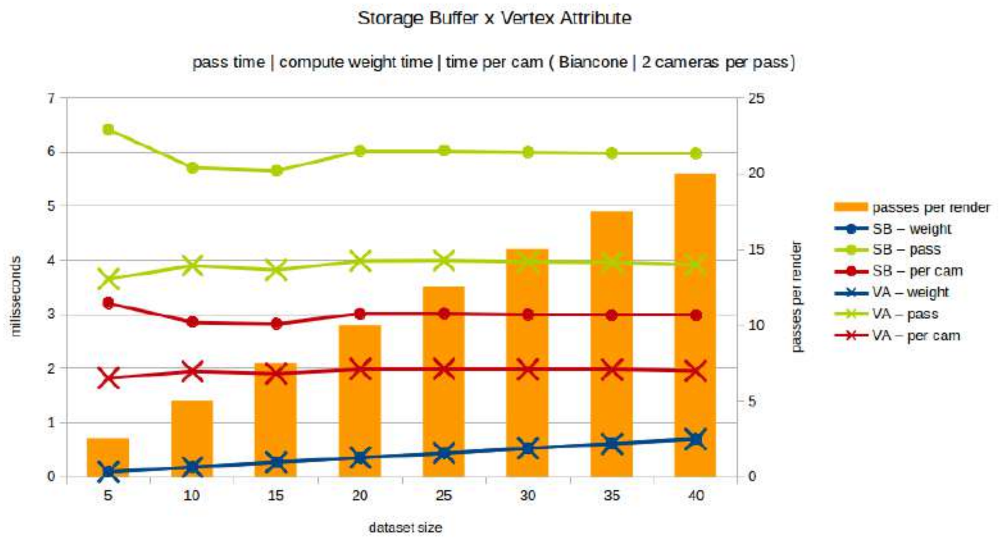


(a) Maximum two cameras per pass.

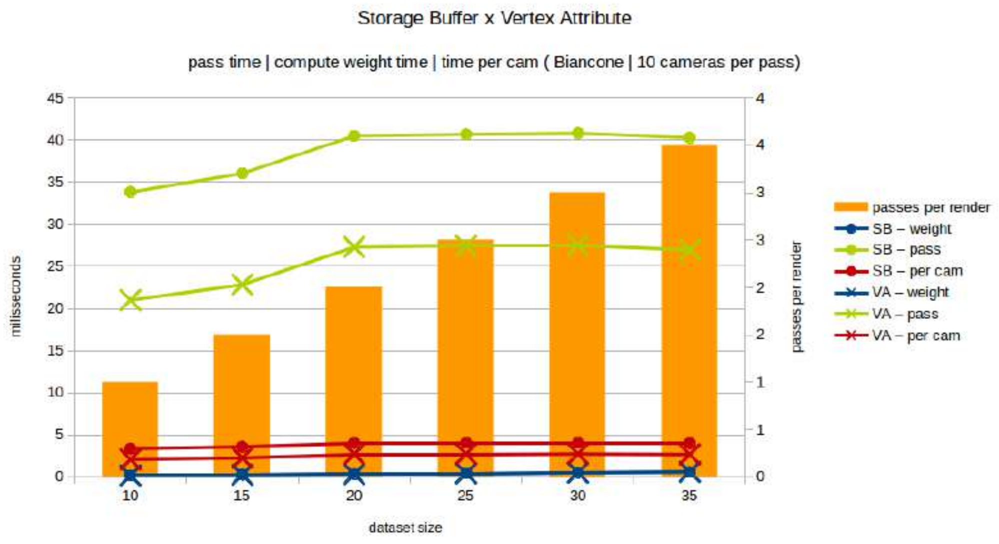


(b) Maximum ten cameras per pass.

Figure 5.17: Duomo model reducing the total number of photos in the dataset.

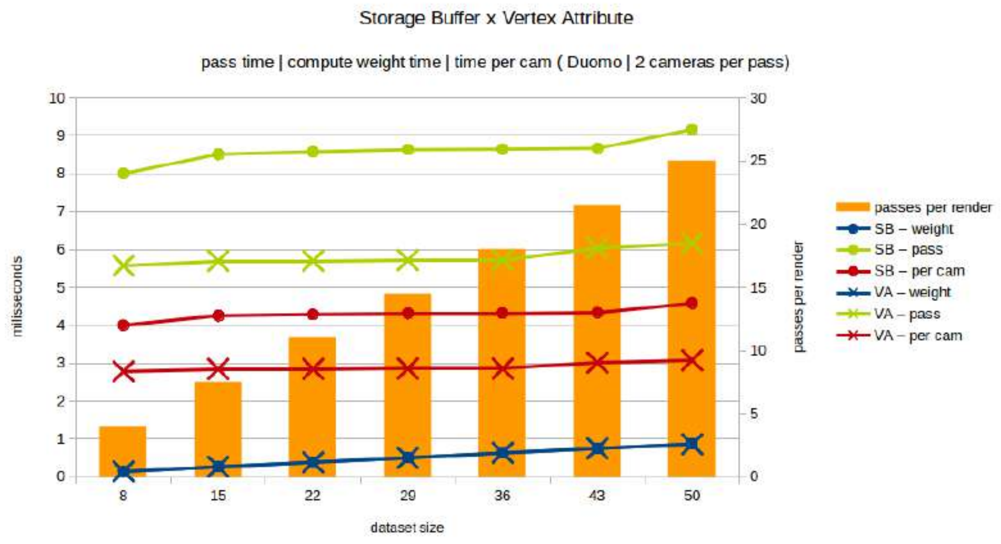


(a) Maximum two cameras per pass.

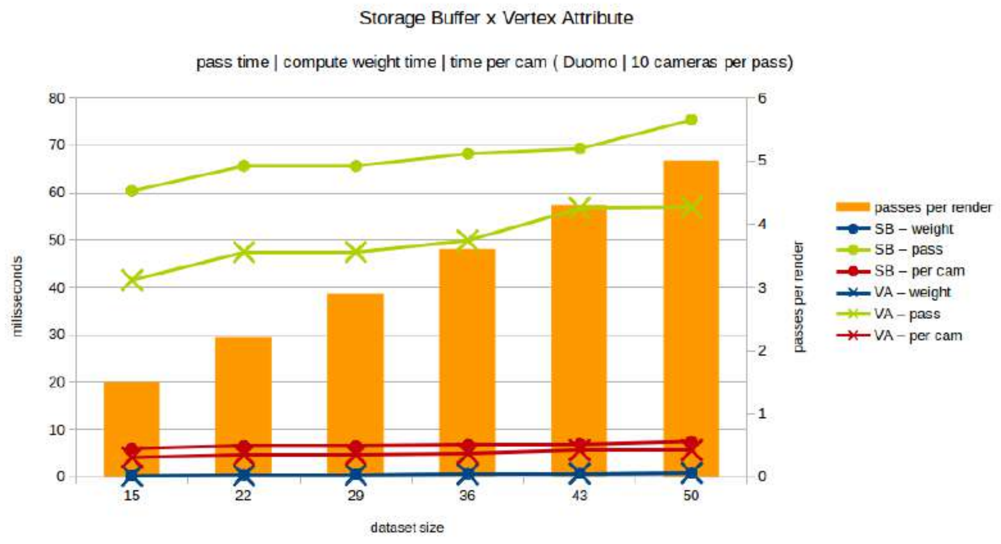


(b) Maximum ten cameras per pass.

Figure 5.18: Biancone model, average timings per pass when reducing the number of photos in the dataset.



(a) Maximum two cameras per pass.



(b) Maximum ten cameras per pass.

Figure 5.19: Duomo model, average timings per pass when reducing the number of photos in the dataset.

Chapter 6

Conclusion

Realistic virtual representations of 3d objects is a common goal in many areas. A popular way to represent reflectance, or color, information for 3d models is through textures. However, when generating a texture from a set of photos, the loss of information may be significant in some cases.

In this dissertation, we propose a real-time rendering system that aims at using at its maximum the original resolution of the photos. We join ideas from texture generation methods and real-time visualization techniques to achieve our goal. Our method blends precomputed and view dependent weights to prioritize the best pixels of a photo, and the best photos given a view position and direction. To deal with large datasets we propose a multi-pass rendering strategy.

We have analyzed our method with different real datasets varying in object scale and number of photos. Through our experiments we have found optimal parameters for the number of photos per pass, and noticed a linear relation between the render time and the number of photos in the dataset.

Albeit the encouraging results, there is still much room for improvement. Our system is very conservative in discarding an entire photo, since we do not have coverage information, i.e., discarding a photo with low weight may also discard the only photo that covers a part of the mesh, and thus leaving the 3d object partially untextured. In fact, an approach to include minimum coverage information may drastically reduce the number of required photos in a give moment. In addition, a minimum coverage strategy it may also discard photos that have low weights and may be only causing blurring artifacts during the computed average.

Most blurring artifacts come from misalignments during the image-to-geometry registration phase. We have not covered this stage in our work, receiving as input to our system the already calibrated and registered photos. Nevertheless, some means to treat, at least partially, the misalignment problem would greatly improve the results. A method such as the one proposed by DELLEPIANE *et al.* [13] to correct misalignments using precomputed optical-flow may be incorporated at the cost of passing extra textures along the images, since the flow between each overlapping pair of photos must be computed.

Another natural direction for future work is to employ methods to correct brightness variations between images. In our datasets all photos were taken during a single session, but if illumination variations can be handled, more generic datasets could be used. For example, a set of photos from the web taken from different cameras at varying illumination scenarios, such as those used in the Photo Tourism system proposed by [11].

Bibliography

[1] ORTIN, D., REMONDINO, F. "Occlusion-free Image Generation for Realistic Texture Mapping". In: editor (Ed.), *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, v. XXXVI, 2005.

[2] PREVITALI, M., BARAZZETTI, L., SCAIONI, M. "An automated and accurate procedure for texture mapping from images". In: *Proceedings of the 2012 18th International Conference on Virtual Systems and Multimedia, VSMM 2012: Virtual Systems in the Information Society*, pp. 591–594, 2012. ISBN: 9781467325653. doi: 10.1109/VSMM.2012.6365984.

[3] BAUMBERG, A. "Blending Images for Texturing 3D Models." *Bmvc*, pp. 404–413, 2002. doi: 10.5244/C.16.38. Disponível em: <`http://pdf.aminer.org/000/067/136/blending{_}images{_}for{_}texturing{_}d{_}models.pdf`>.

[4] BERNARDINI, F., MARTIN, I. M., RUSHMEIER, H. "High-Quality Texture Synthesis from Multiple Scans", *IEEE Transactions on Visualization and Computer Graphics*, v. 7, n. 4, pp. 318–332, 2001.

[5] LEMPITSKY, V., IVANOV, D. "Seamless mosaicing of image-based texture maps". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2007. ISBN: 1424411807. doi: 10.1109/CVPR.2007.383078.

[6] GOLDLUECKE, B., CREMERS, D. "Superresolution texture maps for multiview reconstruction". In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1677–1684, 2009. ISBN: 9781424444205. doi: 10.1109/ICCV.2009.5459378.

[7] OISHI, S., KURAZUME, R., IWASHITA, Y., et al. "Colorization of 3D geometric model utilizing laser reflectivity". In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2319–2326, 2013. ISBN: 9781467356411. doi: 10.1109/ICRA.2013.6630891.

[8] CALLIERI, M., CIGNONI, P., CORSINI, M., et al. "Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3D models", *Computers and Graphics (Pergamon)*, v. 32, n. 4, pp. 464–473, 2008. ISSN: 00978493. doi: 10.1016/j.cag.2008.05.004.

[9] BRIVIO, P., BENEDETTI, L., TARINI, M., et al. "PhotoCloud: Interactive remote exploration of joint 2D and 3D datasets", *IEEE Computer Graphics and Applications*, v. 33, n. 2, pp. 86–97, 2013. ISSN: 02721716. doi: 10.1109/MCG.2012.92.

[10] BURT, P. J., ADELSON, E. H. "A Multiresolution Spline with Application to Image Mosaics", *ACM Trans. Graph.*, v. 2, n. 4, pp. 217–236, out. 1983. ISSN: 0730-0301. doi: 10.1145/245.247. Disponível em: <`http://doi.acm.org/10.1145/245.247`>.

[11] SNAVELY, N., SEITZ, S. M., SZELISKI, R. "Photo tourism", *ACM Transactions on Graphics*, v. 25, n. 3, pp. 835, 2006. ISSN: 07300301. doi: 10.1145/1141911.1141964.

[12] RONG, G., TAN, T.-S. "Utilizing Jump Flooding in Image-based Soft Shadows", *Building*, pp. 173–180, 2006. doi: 10.1145/1180495.1180531. Disponível em: <`http://doi.acm.org/10.1145/1180495.1180531{%}5Cnhttp://dl.acm.org/ft{_}gateway.cfm?id=1180531{&}type=pdf`>.

[13] DELLEPIANE, M., MARROQUIM, R., CALLIERI, M., et al. "Flow-based local optimization for image-to-geometry projection", *IEEE Transactions on Visualization and Computer Graphics*, v. 18, n. 3, pp. 463–474, 2012. ISSN: 10772626. doi: 10.1109/TVCG.2011.75.