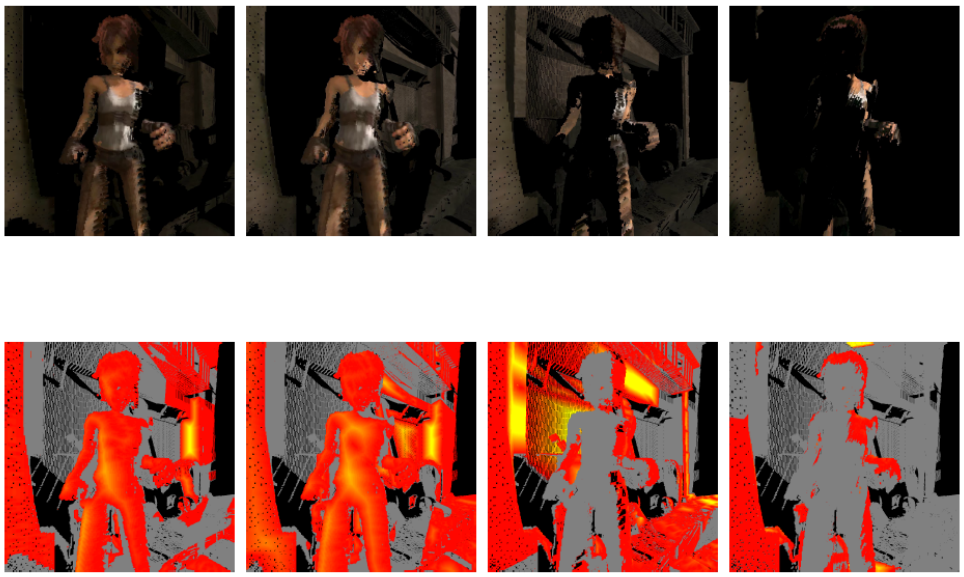


Seam Elimination in Free-Viewpoint Video

Master of Science Thesis in Computer Science - Algorithms, Languages, and Logic

KARL BRISTAV

Department of Computer Science & Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Seam Elimination in Free-Viewpoint Video
KARL BRISTAV

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Cover: Individual camera contributions to scene geometry (top) and their respective weight maps (bottom) generated using a method described in this paper. The scene geometry and images are taken from the Sintel movie project, © copyright Blender Foundation — www.sintel.org

Department of Computer Science and Engineering
Gothenburg, Sweden 2016

Abstract

Free-viewpoint video (FVV) is a term for any system that lets users freely adjust the viewpoint in a video to novel positions, synthesized from source data in the video stream. Recent advances in the area by Kämpe et al. suggest encoding the geometric information separately as dynamic 3D voxel data, together with RGB-video streams from a set of standard RGB cameras capturing the scene from individual view points. During play back, the 3D voxel data is rendered and the color information from the RGB cameras are back-projected onto this geometry, to provide the final result per frame. In the setting of a voxelized geometry and a number of RGB camera streams, some graphical artifacts may occur if the images of the cameras are projected onto the geometry with equal weights. We refer to this as the naive method. To alleviate these issues, three methods are proposed, all building on calculating weights for use in a weighted averaging when calculating the final color of a piece of geometry. The most promising of the proposed methods is implemented and tested against an implementation of the naive method. The results show that our method reduces the number of graphical artifacts while keeping the processing time low.

Keywords: FVV, Free-Viewpoint Video, Computer Graphics, Voxels

Acknowledgements

First of all, I would like to thank Ulf Assarsson for suggesting this topic for my master's thesis.

I would like to hand a huge thanks to my supervisor, Erik Sintorn, for his great ideas, optimism, and support (above and beyond the call of duty).

I also thank Mika Segerström for always being there for me whenever progress was slow or i needed a break.

Finally, I want to thank Jakob Jarmar and Orvar Segerström for reading and commenting on my draft when I had stared myself blind.

Karl Bristav, Gothenburg, January 2016

Contents

Chapter 1

Introduction

Free-Viewpoint Video (FVV) is a system for viewing video in which the viewer can navigate freely inside the 3D-environment of the video and look at the filmed settings and actors from arbitrary view position and angle [1]. This could involve filming the scene with a number of cameras simultaneously with traditional color cameras as well as cameras capturing the distance to the objects in the scene, providing 3D-information from which the scene can be reconstructed.

One way of coloring the geometry of the reconstructed scene is to project the colors from the cameras onto it. This means that for every point on the geometry being colored, that point is projected onto the image plane of each visible camera in order to find the color that camera has for the projected point. When all visible cameras have been sampled, the final color of the given point is set to be the average of all the sampled colors. We refer to this as the naive method.

Some cameras may have different colors for the same point on the geometry, for example if their respective images have different effective resolution for that piece of geometry, or if the material of the geometry is view-dependent. A view-dependent material is a material that does not look the same from different angles. If two or more images then are projected onto a surface, since some of the cameras might not have recorded the same color for the same spot on the geometry, visual seams may occur on the borders of camera visibility.

Another type of visual artifact comes from the fact that both the voxelized geometry and the camera images are approximations of the original scene. Voxel geometry is likely to protrude outside the boundaries of the original geometry, causing a discrepancy between the two. When determining the color of a point on such a voxel protrusion, the projection of that point onto a camera might in some circumstances give the color of a point behind the

given point, as seen from the camera. Even if the geometry was perfect and no protrusions existed, every pixel of the camera image represents a small frustum rather than a single point. A pixel on the edge of an object will have color information from both that object and its background, making a projection of a point at the edge of an object still acquire color from the background.

The problem to solve is then how to project colors onto the surface in such a way that these artifacts are not visible.

This thesis considers the problem in the context of a voxelized scene. The scene consists of 100 frames of voxel geometry and four cameras, each of which has an associated file specifying position, rotation, field-of-view, and aspect ratio, as well as a video stream containing 100 frames of video corresponding to the geometry frames. The voxel geometry is reconstructed from a virtual depth cameras in Blender [2].

To limit the scope of the task, some aspects of the problem will not be considered in this thesis. Temporal consistency, i.e. how the result behaves when applied to consecutive frames and then played back, is an important characteristic of a possible solution, but too large to also consider in this thesis. The speed of the solution is also not wholly considered, as this thesis will focus on finding a solution; not optimizing it. A precomputation approach will be rejected only if it is obviously extremely slow.

Chapter 2

Previous Work

An early architecture for FVV is the *Multiple-Perspective Interactive Video* presented by Kelly et al. [3], which lays a foundation for constructing virtual views based on a number of fixed-position camera streams and simplified geometry representations. Another early system is the *Virtualized Reality*, presented by Rander et al. [1], in which a global mesh geometry is created from local meshes given by each camera and then textured. A system is considered specialized if it is not designed with general situations in mind, but can only be used for certain specific situations.

Examples of specialized systems include that of Horiuchi et al. [4], who present a system for rendering musicians from different viewpoints on different backgrounds. Koyama et al. [5] present a system for live FVV of a soccer stadium. Carranza et al. [6] present a model that utilizes a human body model and thus is unsuitable for subjects other than human beings. Hauswiesner et al. [7] also make use of a human body model when they present a system that allow users to try on clothes articles in a virtual environment. All of these make assumptions regarding the subjects and their properties.

A common property of many of the specialized systems mentioned earlier is that they make use of methods that only utilize the images provided by the viewpoints and information regarding the position and orientation of those; the geometry of the subject is not a part of the model. These systems are classified as image-based systems, whereas systems that utilize a computed geometry of the subject are called model-based systems [8].

General systems have the property that they are designed to work on any subject, and thus make very few assumptions regarding what kind of scene is portrayed. Examples of general systems include the unstructured lumigraph by Buehler et al. [9], which is dependent on a relatively high number of cameras. The systems presented by both Collet et al. [10] and Salvador et al.

[11] blends colors from cameras based on surface normals and the intrinsic interpolation of surface normals in a mesh geometry; the system of Collet et al. [10] additionally relies on a strictly controlled environment and a high number of cameras to achieve a good result. Ishikawa et al. [12] proposes an image-based general system where the user can walk through the scene, something that is normally not possible using image-based methods. The system, however is limited to views in a specific plane and also requires a high amount of specifically positioned cameras. Palomo and Gattass [13] propose an image-based algorithm that is capable of performing FVV using the images of conventional cameras and images by depth cameras. However, the purpose of the system is to be an efficient way of rendering FVV in real-time without need for precomputation, and it does not take into concern the visual artifacts that might be present..

Some systems fall somewhere between specialized and general. Wang et al. [14] consider a framework for synthesis of viewpoints between already existing viewpoints, as opposed to a system where the viewpoint can be freely moved; this fact does not make it specialized, but it might not be considered entirely general either.

FVV systems are possible to use in a wide variety of circumstances, and on different platforms. Systems have been presented for controlling FVV systems using a touch screen for possible use on mobile devices [15], and for streaming and cloud rendering of FVV data [16, 17].

Kämpe et al. [18] present an efficient method for storing time-varying voxel data in a directed acyclic graph (DAG). The voxel geometry this thesis uses is stored using this method.

An interesting technique is *relighting* [19], with which the scene lighting is undone so that the scene may be lit from an arbitrary source within the FVV system. One usage of relighting is when FVV content is mixed with other content, either other FVV content or some virtual environment. Relighting is then used so that all content can be lit the same way. Relighting could be used to eliminate the color differences between cameras so that seams will not occur. On the other hand, this would also erase any previous lighting information present in the scene. This thesis aims to facilitate the translation of already lit scenes from real world to FVV, preserving the lighting from the original scenes.

As for the texturing of the scene, Salvador et al. [11], Collet et al. [10], and Starck and Hilton [20] all employ weighted camera blending to achieve their results. The approaches of all of these, however, are not applicable to this thesis because they make use of normals in a generated 3D mesh to generate camera weights, something that is not directly applicable in this thesis as no information regarding the normals of the geometry is available.

An interesting method for FVV rendering without losing view-dependent information (such as reflections) is *view-dependent texture mapping (VDTM)*. VDTM was originally presented as a method for simulating geometric detail on basic geometric models [21], similar to how normal mapping [22] is used in modern computer graphics. Volino and Hilton [23] present a layered texture representation for VDTM with focus on FVV use that retains view-dependent information, improving realism.

Lempitsky and Ivanov [24] Describe a method for constructing a mosaic-like texture mapping from different cameras on triangle meshes. The method uses *seam levelling* in order to smooth seams between mosaic fragments. The seam levelling implementation presented makes use of the triangle mesh structure, making it unsuitable for use with a voxel grid.

Chapter 3

Method

Given a scene geometry and the color images from a number of cameras, this thesis considers how to assign colors to any point in space. The multiple frames of scene geometry consists of voxels specified by a single *Directed Acyclic Graph (DAG)* containing all geometry frames, as described by Kämpe et al. [25], and a number of static cameras, each with an associated position, orientation, field-of-view angle, and an image stream that correspond to the frames of the geometry.

One way of solving the problem of determining the color of a given point p is to project p onto the camera plane of each scene camera C_i. The colors Col_p^i of the pixels at the projected positions in the scene camera images $i = [1..N]$ are then averaged to calculate the final color Col_p:

$$Col_p = \frac{\sum_{i=1}^{N} Col_p^i}{N} \tag{3.1}$$

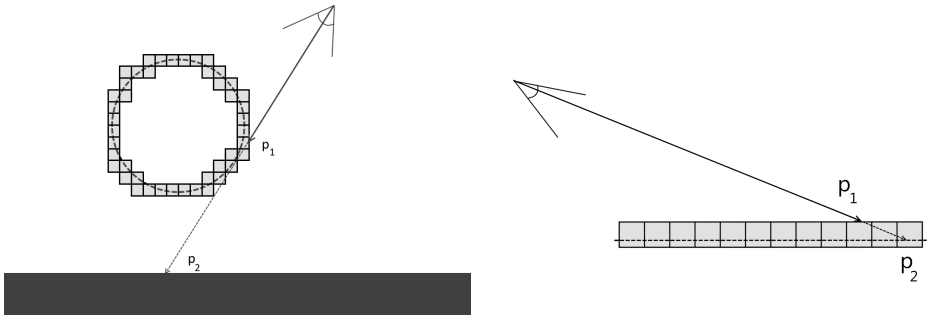
This method, however, disregards the geometry of the scene and may produce erroneous color values for points that are occluded by some geometry, as the color from the occluder will be used in the final value. We can determine whether a point is seen by a given camera or not by comparing the distance to the point from the scene camera and the value of the projected position on the depth map (an image from a scene camera's point of view representing distances to the geometry) of the scene camera [11], if such a depth map is available. If the value obtained from the depth map and the distance between the point and the scene camera are sufficiently close, the point is considered visible from that scene camera and the scene camera should be sampled for the final color; if the values differ, the point is occluded and that camera should not be sampled for the final color. Another way to test if a point is seen by a specific scene camera is to shoot a ray from the point towards the scene camera; if the ray hits geometry between the point and the camera,

the point is occluded. The final color is then obtained by what we call the naive projection method:

$$Col_p = \frac{\sum_{i \in C^v} Col_p^i}{|C^v|} \tag{3.2}$$

where C^v is the set of scene cameras visible from the point.

For a given point, the color given by projection on scene camera images may differ between cameras even if the point represents the same object for the scene cameras. There are various conceivable reasons for this. Firstly, the material of the object may be view-dependent, that is, it looks different from different directions. Any material that does not reflect light uniformly is view-dependent, as more light will be reflected in certain directions than others. Secondly, there might be a mismatch between the scene camera images and the scene geometry, as the voxelized geometry does not exactly correspond to the original scene geometry. Such a mismatch could cause an erroneous color sampling along silhouettes of objects; a point on the very edge of an object, as seen from a specific scene camera, could give the color value from the object behind it instead. The sampling mismatch is illustrated in Figure 3.1a. Another kind of mismatch can be seen is shown in Figure 3.1b, where the color value is obtained from the correct surface at the wrong position, thereby displacing the projection on that surface in the direction toward the camera.



(a) (b)

Figure 3.1: *Sampling mismatches occur. The dashed circle and line represents original geometry, and the squares represent voxels in the scene geometry. In both cases, coloring the point p_1 will give the color of p_2 from the camera, as the position of p_1 corresponds to p_2 in the original geometry*

These are the causes of the visual artifacts that may be present when the naive projection method is used. The problem then becomes how to,

correctly and without visual artifacts, determine the color of a given point of the geometry surface.

3.1 Choosing colors for view-dependent materials

As mentioned earlier, due to the nature of some materials, a point may give different colors from different scene cameras. The problem of choosing which of these colors to use, or how to combine them, for the final color is not trivial. When choosing colors for a view-dependent material, no solution is complete. Consider a mirror surface; every single view of the mirror will see a unique reflection in the mirror, thus we would need an infinite number of cameras to perfectly replicate the possible views of the mirror surface. Thus, everything described from this point on is to be considered as approximations, that may give plausible results for slightly view-dependent materials.

Simply choosing the color from the scene camera that might be considered to have the best view of a given position may give unsatisfactory results: If two neighboring points have different best scene cameras with different colors, the result will not look good. Similarly, blending colors from all visible scene cameras with equal weight may also give unsatisfactory results, as some scene cameras might have a very angled or otherwise compromised view of the given point, making them unsuitable for sampling.

There are a number of possible solutions to this problem. Instead of choosing the best scene camera to sample for a color, it is possible to choose a number of suitable scene cameras, and blend the color values acquired from them to produce the final color [11]. Apart from just choosing the cameras that see the given point, the normal of the surface on which the given point is located can be used to select which of these cameras can be used. The surface normal can also be used to determine the blending weights of these chosen cameras. The blending weight of each selected scene camera could be defined in such a way that the scene cameras for which the view vectors are the closest to being parallel to the surface normal of the given point are given more weight in the blending [11]. If blending the colors is not considered a viable option, levelling functions may be used. Levelling functions are commonly used in image stitching, and can be used to combine images without overlapping areas [26].

3.2 Eliminating occlusion seams and other artifacts

When a surface is partly visible by a camera, A, either because the surface is partly occluded or because it is partly outside the view of A, there may be a visible seam between the points where A is used and the points where it is not if the surface in question is view-dependent or if the cameras have different effective resolution for that surface. Figure 3.2 illustrates the circumstances under which seams may appear.

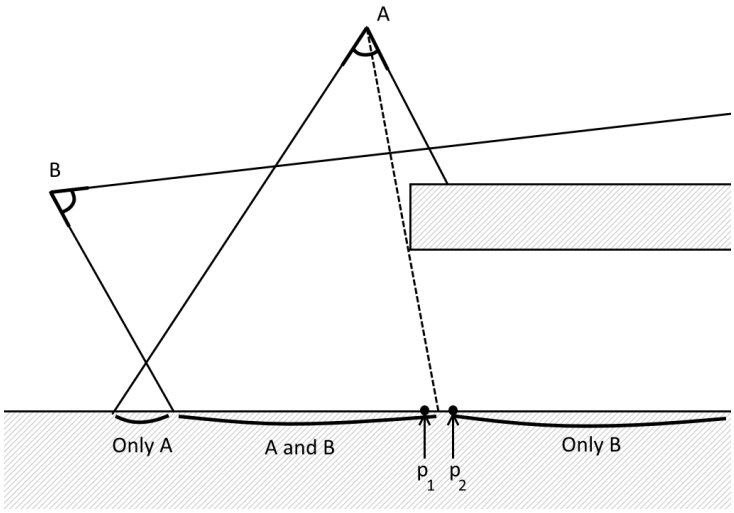


Figure 3.2: *point p_1 is in full view of scene cameras A and B, while its neighbor, p_2 is only in view by scene camera B. This could cause a seam between p_1 and p_2.*

The problem is how to, for any point in the scene, assign weights to scene cameras in order to present a plausible result without visible seams or other artifacts, such as sampling mismatches.

While there is not much previous work that deals explicitly with this kind of seams, some of the methods for handling view-dependent materials as described in section 3.1 could conceivably be used to handle this problem. A levelling function, for instance, could, combined with good selection of scene cameras to sample, provide a possible solution. That solution, however, makes use of a triangle mesh structure which we do not have access to.

Our solution to this problem is to assign weights to scene cameras based on how well they see a piece of geometry. To achieve this, the view of each scene camera is divided into *areas of visibility*. An area of visibility is a

geometrically continuous area which is visible to a certain scene camera. These areas of visibility are used to establish *weight maps*, a mapping of weights per scene camera onto either the geometry or the scene camera image, specifying the weight of each color in the scene camera image. The weights should be assigned in such a fashion that each area of visibility has a set weight, except near the edges, where the weights are lower the closer to the edge they are. This ensures that the center of the area of visibility is associated with full weight of that scene camera, while the scene camera's influence fades out near the edges, making for a seamless blending into other cameras' areas of visibility. Figure 3.3 illustrates two areas of visibility from two different cameras. This solution alleviates the problems with occlusion seams by making sure that if an area of visibility overlaps another one, the weights of both areas are lower closer to the edges than in the center, fading out their influence. This solution also alleviates the problem of sampling mismatches along edges of objects because the silhouette edge of an object is per definition also the edge of the area of visibility on that object; thus the colors that are likely to be erroneously sampled have a very low weight compared to those sampled from scene cameras for which the given geometry point is not along the object's edge.

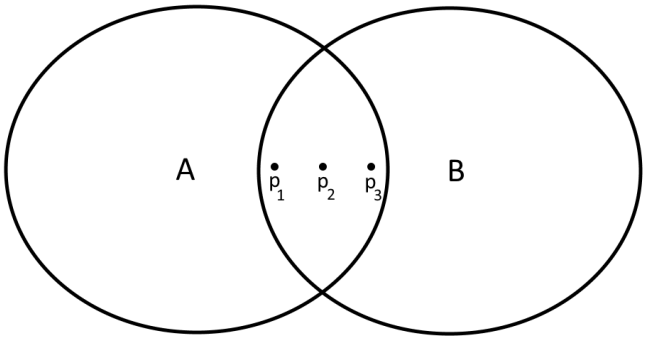


Figure 3.3: *A surface with areas visible to scene cameras A and B, respectively. In p_1, B has a low weight while A has a high weight. In p_2, A and B have the same weight. In p_3, B outweighs A.*

The presentation of three methods will follow, each taking a slightly different approach to the solution presented.

3.2.1 Identifying areas of visibility in voxel space

With this method, areas of visibility are built per scene camera by identifying which voxels are on the edge of such an area, followed by an iterative filling process that assigns weights to voxels based on how many voxel steps away the edge is.

A voxel is defined to be on the edge of an area of visibility if any of its neighbor voxels are not visible from the scene camera, which means that it is either occluded or outside the scene camera view. After the edge voxels have been identified, they are given the weight 1 and put in a list. For each voxel in this list, its unprocessed, non-occluded neighbors are given the weight 2 and put in a new list. This new list is processed in the same manner as the last, setting the weight of every unprocessed and non-occluded neighbor to 3, and putting them in a new list. This continues until a set max weight is reached, whereupon the rest of the unprocessed and non-occluded voxels are given the max weight. This ensures that the area of visibility has a uniform weight value in the center region and fading weights in the edge region. Figure 3.4 illustrates an area of visibility filled with weight values.

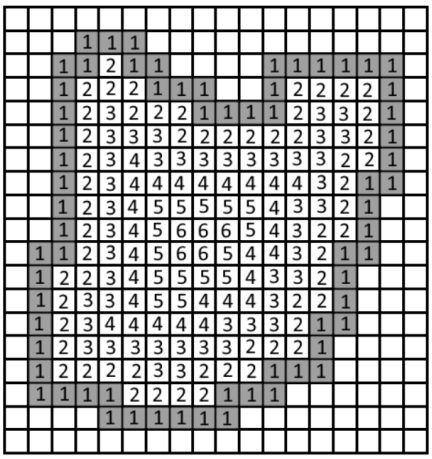


Figure 3.4: *Weights of a flat voxel grid, with edge voxels marked in gray. All voxels inside the edges have been marked with their respective distances to the outside of the area.*

However, there are some limitations with this method. Since the weights are assigned based purely on integer distance across the geometry to the nearest area edge, the distance of the camera to the geometry is not taken into consideration. It could be done by modifying the weights with some

function of the distance between the voxel in question and the camera, for example the inverse distance, but that might, for a very close camera, change the weight values so much that the requirement of edges with weights close to 0 is broken. The result is that if the weights are not (relatively) close to 0 in the edges of some regions, they might not blend seamlessly with overlapping areas, producing visual artifacts.

Another issue is that this method has a voxel-centric view of the problem, and that might cause artifacts in the finished result. For example, one voxel may correspond to more than one pixel in the camera image. This, combined with the fact that only one color per voxel is stored, risk making some parts of the scene (specifically, the part closest to the camera in question) undersampled. Another problem comes from the fact that we determine occlusion for some camera by shooting rays from the center of the voxel towards the camera. While this might intuitively not seem to be a problem, there exist situations in which the center of a voxel is occluded but not some other part (see Figure 3.5). Even if we were to shoot rays from all corners of the voxel, as well as its center, there might exist situations in which the center and all corners of a voxel are occluded, but not some other part. The result of this is that some voxels will be falsely flagged as completely occluded, making for a possibly inadequate solution.

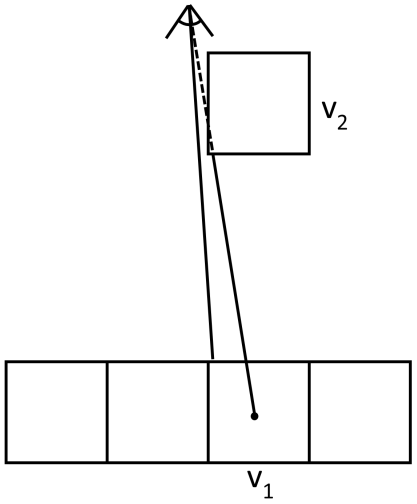


Figure 3.5: *The voxel v_1 is being identified as occluded by v_2 despite not being so, due to inadequate occlusion testing.*

3.2.2 Dynamically identifying areas of visibility in view-camera space

This method approaches the problem from the direction of the view camera, which is the camera from which the scene is viewed by the user of the system. The method dynamically establishes areas of visibility only for the points currently being rendered.

The scene camera weights for each pixel of the view are calculated by shooting rays through pixels in the vicinity of that pixel. The vicinity of a pixel is defined as an area surrounding it with some margin, defined in number of pixels, as seen in Figure 3.6. For each of the rays that hit anything, the hit position is checked for occlusion against all scene cameras that can see the original pixel. The weights are then calculated by summing the number of hit positions visible from each camera.

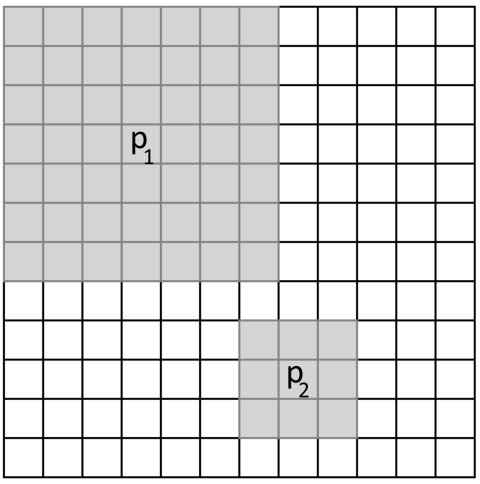


Figure 3.6: *Examples of square vicinities of pixels p_1 and p_2. These vicinities are 7 and 3 pixels wide, respectively.*

This method also ensures that the areas of visibility have a uniformly weighted center, as the max weight assigned is the area of the vicinity, which is a set size throughout the entire process. Figure 3.7 illustrates the general idea of this solution.

An advantage of this method is that since colors are not stored in the voxels, but rather retrieved on a per pixel basis, a voxel can have any number of colors, preventing undersampling of the scene cameras.

One big disadvantage of this method is that the weight calculation cannot

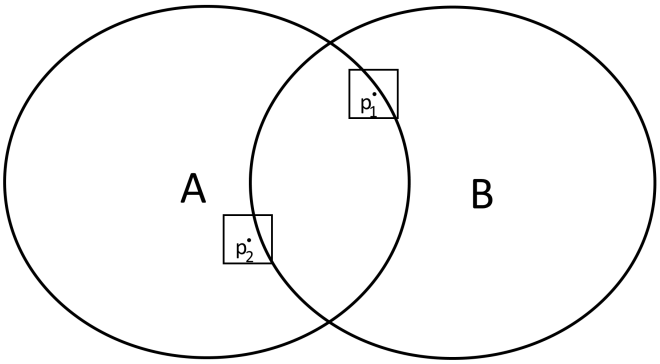


Figure 3.7: *The point p_1 has full weight from B and little more than half weight from A. p_2, on the other hand, has full weight from A and no weight from B, since p_2 is not visible by B.*

be precomputed since it depends on the location and rotation of the view. This makes this solution fairly computationally heavy in real-time.

Another drawback is that if the pixel is near the edge of an object, some part of the vicinity of the pixel might correspond to a piece of background geometry. If it does, the visibility of the camera in the background may be used in the weight calculation of the pixel, as illustrated in Figure 3.8. This may cause some areas to have more weight for a certain camera than they should, and would affect the final result.

This algorithm does not provide a complete solution to the given problem, as the weight of a point close to the edge of an area of visibility will not be close to zero, as is desirable. This will reduce the effectiveness of the method. A slightly modified version of the method can be used to satisfy the edge weight condition: The weight of each camera visible at the given point is set to be the closest pixel distance to a point that is not within one of that camera's areas of visibility. This will ensure that the points close to an area of visibility edge is given low weight for that camera.

Given the modified weight assigning algorithm, the method does still not completely solve the problem. The center of an area of visibility could be at the edge as seen from the view, giving it low weights where it should have high. The reverse is also possible, and will cause sampling mismatches to still be visible in some cases.

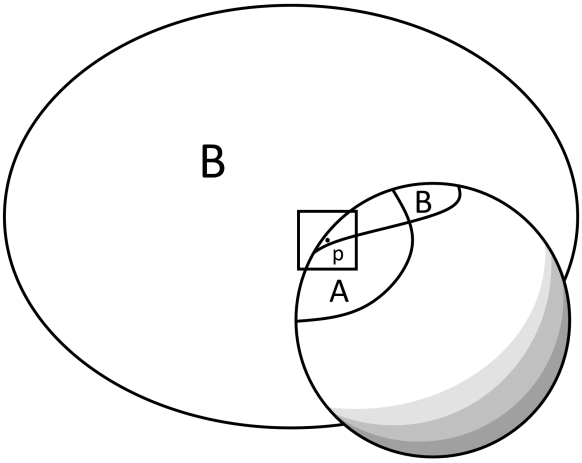


Figure 3.8: *The point p should only have a small weight for B, but since the background of the object is also seen by B, it gets taken into consideration, giving B a larger weight than it should have.*

3.2.3 Identifying areas of visibility in scene camera space

This method combines a screen space approach with the ability to precompute weights, so that instead of calculating the weights every time they are needed, they are precomputed and stored on disk. The areas of visibility are calculated in scene camera space, and the weight of a pixel is represented by the distance to the nearest local discontinuity in geometry. The smallest weight this method will assign a pixel is 1, as that is the smallest distance from one pixel to another.

The precomputation is done by first shooting one ray per image pixel for each scene camera and store the resulting collision distances in a depth map. These distances are then used to, for each image pixel of the scene camera, calculate the distance to the nearest local discontinuity. A local discontinuity is defined as when the recorded depth of two neighboring pixels of the scene camera image differ by more than a certain threshold value (See Figure 3.9). The choice of threshold value is important, as a too low value could make the solution interpret surfaces that face the camera as discontinuous with themselves, giving too low weights. The discontinuity is found by traversing the depth map around the hit pixel in an outward fashion until a local discontinuity is found. This distance is then stored as the weight for that pixel in that camera. It is worth noting at this point that

this algorithm is not optimized for performance in any way, and is thus not suitable for real-world usage.

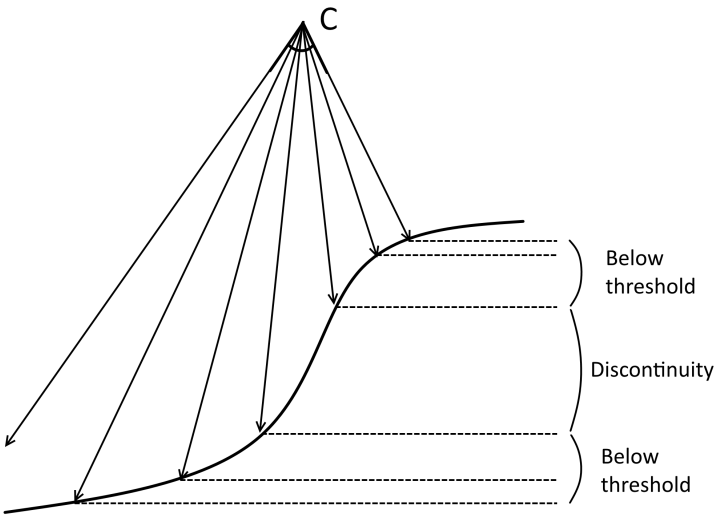


Figure 3.9: *A local discontinuity is detected when the difference between two neighboring distances is more than a certain threshold.*

When a ray is shot from the view camera into the scene to calculate the color for the pixel corresponding to the ray, the colors and the stored weights for the hit position is looked up for each camera. The final color is then calculated using a weighted average of the colors from the cameras.

An advantage of this method is that weights for a piece of geometry far away will be lower than weights for an identical piece of geometry that is closer (see Figure 3.10), making distance an integral part of weight calculation. This is advantageous because as the camera has a lower effective resolution on an object that is far away, we want the camera to have lower weights on that object.

The outward search for a local discontinuity for the calculation of each weight could become very time-consuming, as the search area is larger than the square of the search distance. Therefore, a max distance value is introduced, where the search will stop as soon as it has reached a certain distance from the pixel of origin. If this max value is reached, the weight of the pixel is set to the max distance. This will result in weight plateaus in the middle of areas of visibility that are large enough.

The fact that the weights actually are the pixel distance to the closest discontinuity might give unexpected results if the scene camera images are of different resolutions. Some sort of normalization is likely necessary in order

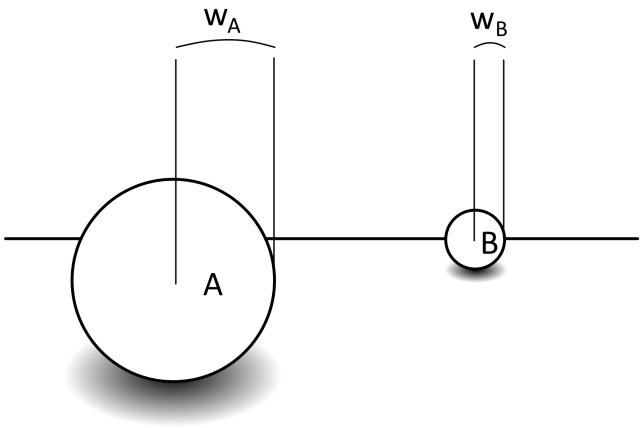


Figure 3.10: *Despite being of equal size, objects A and B have different weights in the center, as A appears much bigger in the scene camera space.*

to achieve even results in such cases.

If the geometry resolution is higher than the scene camera image resolution, there might be cases where discontinuities in the geometry are missed because the resolution of the depth map is too low to discover them. This could conceivably be a problem. The resolution of the depth map could of course be increased to avoid this, but then it would also take longer to calculate.

Chapter 4

Results

Given a scene consisting of voxel geometry and a number of camera positions with associated video streams, we want to determine the color of a given position on the geometry. The naive method does this by checking which cameras are in view of the given position, and averaging the color values from those cameras at that position, as described in Equation 3.2. This might create a number of different artifacts, and our method is designed to eliminate these. Our method uses a weighted averaging, and seeks to assign different weights to the different areas of the camera projections, based on how good of a view the camera has of the corresponding area in the geometry.

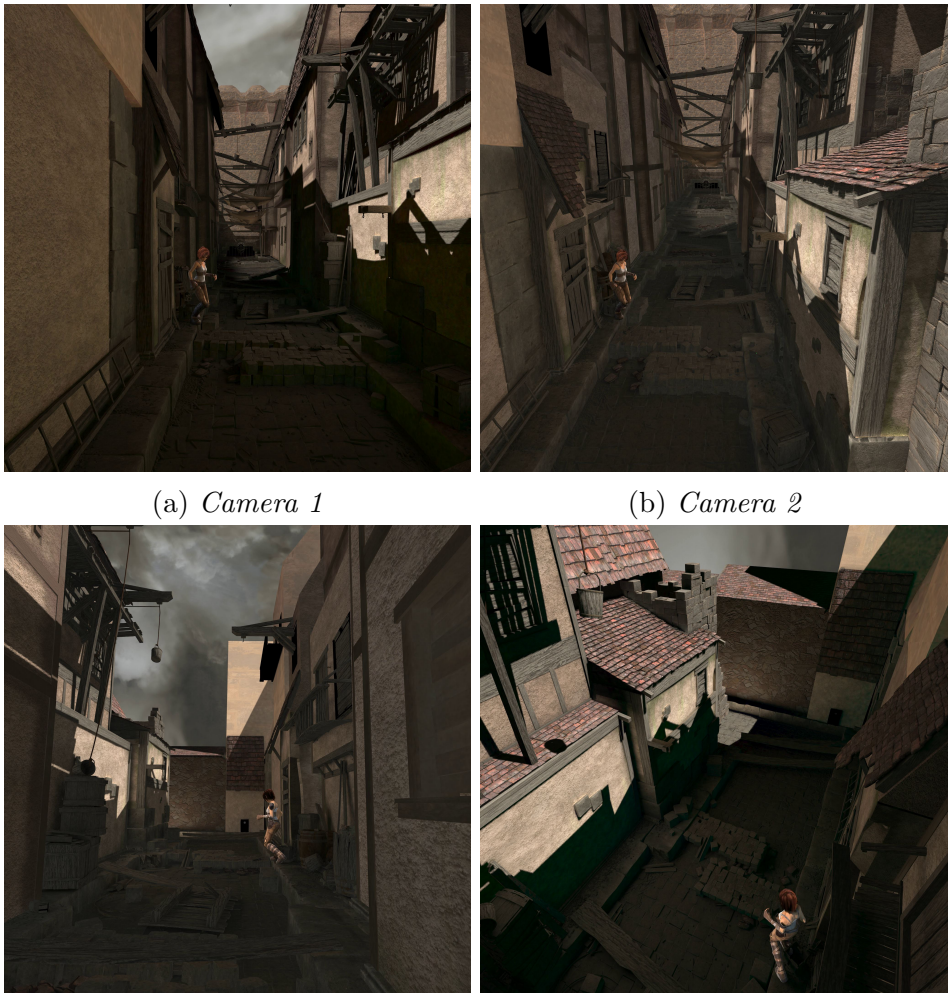
The method was tested by implementing both the naive method and the algorithm described in section 3.2.3, and then comparing the visual quality of a number of views rendered using them both. Our implemented method is an algorithm that assigns weights in camera space of the scene cameras. The weight of a pixel in the scene camera image is based on the pixel distance to the nearest local discontinuity in the geometry. Max distance is, in the context of our method, the maximum distance at which the local discontinuities are looked for, and thus also the maximum weight given to a pixel in a scene camera image. The implementation of our method uses a max distance value of 32 throughout most of this chapter (specifically, sections 4.1.1, 4.1.2, 4.1.3, and 4.1.4). The choice of implemented method was made based on the limited timeframe of the master thesis only giving adequate room to implement a single method, combined with the desirable properties possessed by the chosen method. The implementations were made in `C#` using the OpenTK toolkit [27] for rendering and vector math.

A number of *views* (virtual camera positions and orientations) were selected for comparison by identifying parts that displayed prominent occlusion seams and/or other graphical artifacts when rendered using the naive method. The graphical artifacts produced by the naive method include oc-

clusion seams, sampling mismatches, and projection displacement. Occlusion seams occur on edges along the occlusion shadow of an object (see section 3.2). When cameras have different color values for the same position, the transition from an area both cameras can see to an area that only one camera can see will be a transition from two different colors averaged to just one of the colors. Sampling mismatches occur because the voxelized scene geometry does not always match exactly with the original scene geometry (see Figure 3.1a). If a voxel extends outside the surface of the original object, a ray that would pass by the object in the original scene might hit the voxel, creating a mismatch between the actual and believed hit positions of the ray. This mismatch may cause the color of the background of the object to be projected onto the object itself. Projection displacement also occurs when the voxelized geometry does not closely correspond to the original geometry (see Figure 3.1b). A ray shot toward an object could hit the voxelized surface somewhat earlier than it would hit the surface in the original geometry. If the ray is shot at an angle, this might cause the projection on that surface to be displaced in the direction toward the camera.

4.1 Test results

Four camera views exhibiting occlusion seams and other graphical artifacts will be presented, and figures illustrating the difference between the two rendering methods will be shown. Figure 4.1 shows, for reference, the first frame of each scene camera's video stream. These four frames comprise the entire image input for the algorithm.



(a) *Camera 1* (b) *Camera 2*

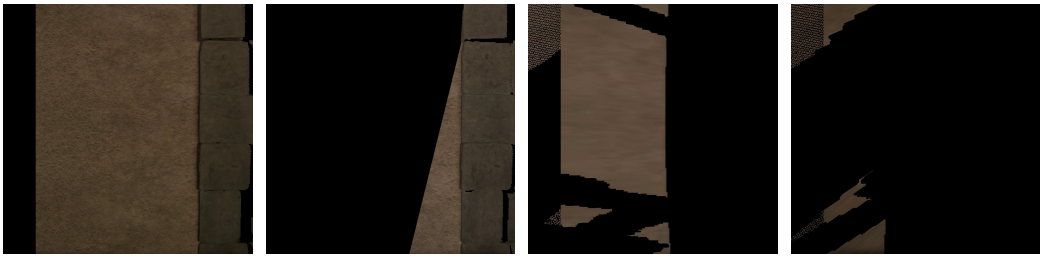
(c) *Camera 3* (d) *Camera 4*

Figure 4.1: *The views of each camera in the scene.*

4.1.1 View 1: Wall

The *Wall* view corresponds to a geometrically simple part of the scene depicting a flat wall. It features a number of seams and serves a good example of the chosen method. Figure 4.4a shows the view rendered using the naive method.

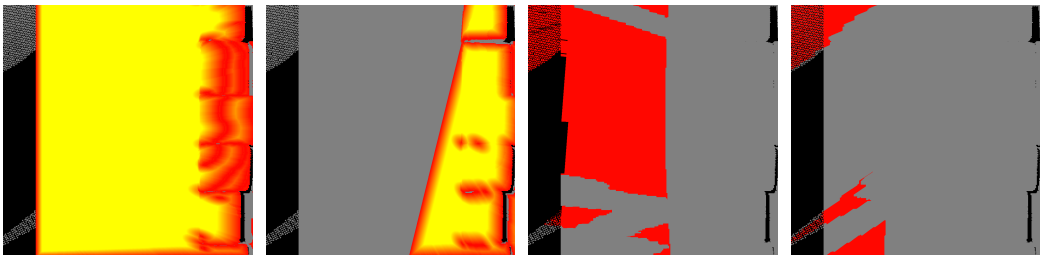
Each camera contributes equally to the result in Figure 4.4a, and the contribution of each camera can be seen in Figure 4.2. Cameras 3 and 4 see the wall at an angle, and so have effectively lower resolution than the other two, making the colors in their contributions seem smeared out.



(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.2: *Individual camera contributions of the* Wall *view rendered using the naive method. Black corresponds to no contribution from that camera in that view.*

The weights of the contributing cameras, for use in our method, are found in Figure 4.3. For this view, Camera 1 is contributing greatly to the final result; Camera 2 is also contributing, but not in as large area as Camera 1, and cameras 3 and 4 are almost not contributing at all.

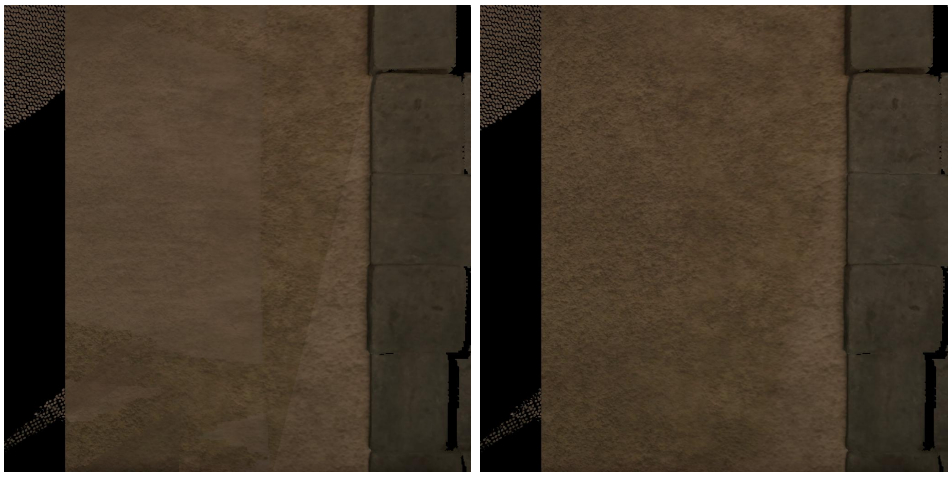


(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.3: *Individual camera weight maps of the* Wall *view in our method. Red corresponds to a weight of* 1*, yellow corresponds to a weight of* 32*.*

The complete rendering with our method can be seen in figure 4.4b. Figure 4.4 shows the comparison of the *Wall* view rendered with the naive

method and our method, respectively. As is evident, our method produces a result without visible seams, without adding additional graphical artifacts.



(a) Wall *view rendered using the naive method.*

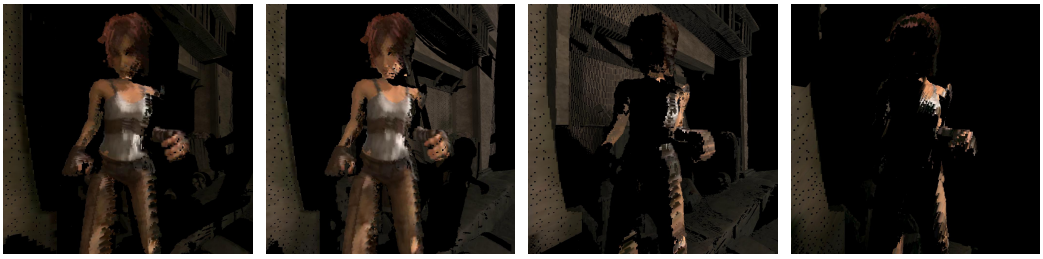
(b) Wall *view rendered using our method.*

Figure 4.4: *Comparison of the naive rendering and thesis rendering for the* Wall *view*

4.1.2 View 2: Woman

The *Woman* view depicts the woman in the scene. Figure 4.7a shows the view rendered using the naive method. Using the naive method, this view does not display a lot of seams, but is fraught with other visual artifacts. These artifacts are created by sampling mismatches and the fact that none of the cameras in the scene can see the woman in her entirety, so there are few overlapping areas of visibility, especially on her left side.

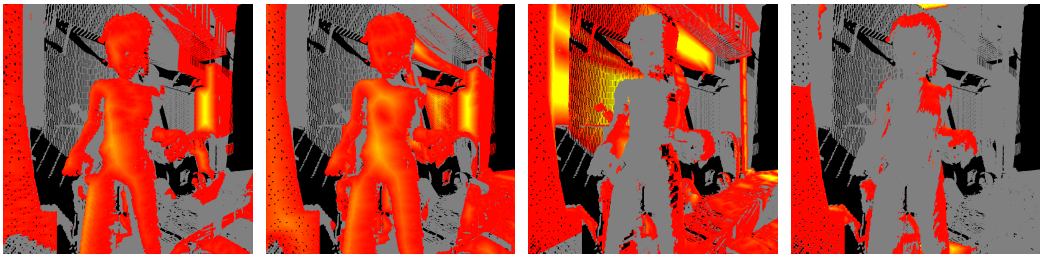
Figure 4.5 breaks down the individual contributions to the result using the naive method shown in Figure 4.7a. From this figure, we can see that Camera 3 and Camera 1 are responsible for a lot of the artifacts on the woman's left side because of sampling mismatches. We can see that none of the cameras has an adequate view of the left side of the woman at all, making all parts of the her facing in that direction suffer from graphical artifacts.



(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.5: *Individual camera contributions of the* Woman *view rendered using the naive method. Black corresponds to no contribution from that camera in that view.*

Using our method for weight maps (seen in Figure 4.6) gives good weights of cameras 1 and 2 for the upper body and front of the legs of the woman. All cameras have very low weights along her left side, meaning that none of them will contribute significantly more than any other camera.



(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.6: *Individual camera weight maps of the Woman view in our method. Red corresponds to a weight of* 1*, yellow corresponds to a weight of* 32*.*

The complete rendering with our method can be seen in figure 4.7b. Figure 4.7 shows the comparison of the Woman view rendered with the naive method and our method, respectively. Our method alleviates some of the problems seen when using the naive method, but retains some problems as well. The big artifacts along the woman's left leg and torso are slightly smaller, and the woman's torso is also clearer.

(a) Woman *view rendered using the naive method.* (b) Woman *view rendered using our method.*

Figure 4.7: *Comparison of the naive rendering and thesis rendering for the* Woman *view*

4.1.3 View 3: Roof

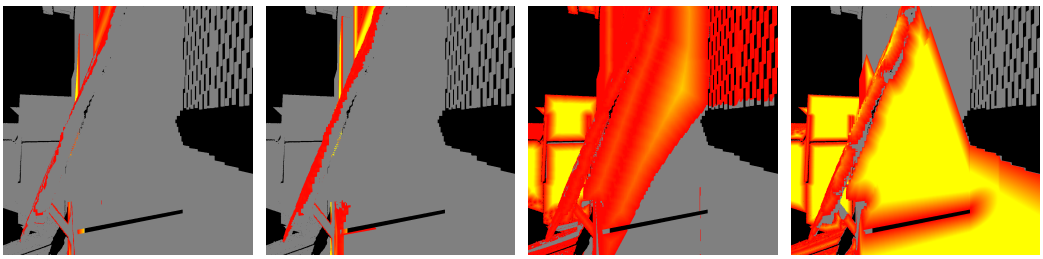
The *Roof* view is focused on a piece of wall and roof just above the woman in the scene. The *Roof* view displays, when rendered using the naive method, two large seams crossing each other. One of the seams arises from occlusion of another object, whereas the other seam is the consequence of the field of view of a camera ending. Figure 4.10a shows the view rendered using the naive method.

As can be seen in Figure 4.8, the contributions to the naive method result (Figure 4.10a) are done almost exclusively by only two of the four cameras. Camera 3 has a view of the geometry from the ground, whereas Camera 4 has a closer view, from a higher position. Each camera contributes equally to the result in Figure 4.10a.

(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.8: *Individual camera contributions of the* Roof *view rendered using the naive method. Black corresponds to no contribution from that camera in that view.*

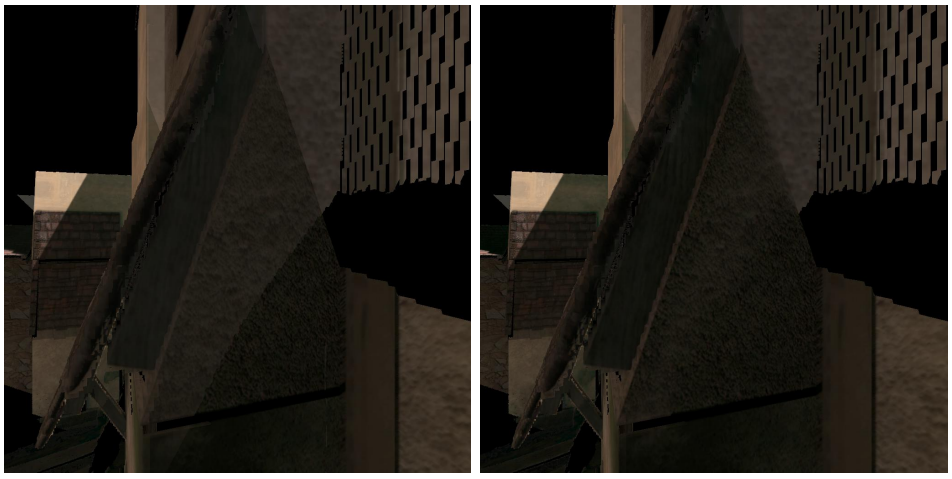
The weights of the contributing cameras, for use in our method, are found in Figure 4.9. Here, Camera 4 is awarded higher weights than Camera 3, because of its proximity to the geometry.



(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.9: *Individual camera weight maps of the* Roof *view in our method. Red corresponds to a weight of* 1*, yellow corresponds to a weight of* 32*.*

The complete rendering with our method can be seen in figure 4.10b. Figure 4.10 shows the comparison of the *Roof* view rendered with the naive method and our method, respectively. As can be seen, Camera 4 has a large influence on the final result. Our method produces a result that has eliminated the two prominent seams.



(a) Roof *view rendered using the naive method.*

(b) Roof *view rendered using our method.*

Figure 4.10: *Comparison of the naive rendering and thesis rendering for the* Roof *view*

4.1.4 View 4: Bucket

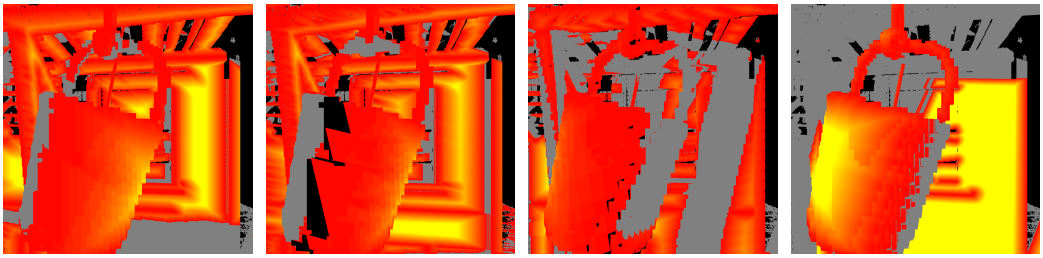
The *Bucket* view shows a bucket hanging in a piece of rope in front of a building façade. A naive rendering (see Figure 4.13a) of this view displays a high amount of sampling mismatches along the edges of the bucket, as well as some blurriness caused by projection displacement on the façade.

Each camera contributes equally to the result in Figure 4.13a, and the contribution of each camera can be seen in Figure 4.11. We can see that cameras 1 and 4 produce significant amounts of sampling mismatches; from a wall in the case of Camera 1, and from a tile roof in the case of Camera 4. The source of the projection displacement is mainly Camera 2, which views the façade at an angle.

(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.11: *Individual camera contributions of the* Bucket *view rendered using the naive method. Black corresponds to no contribution from that camera in that view. We can observe sampling mismatches on the left side of the bucket in Camera 1, as well as on the right side of the bucket in Camera 4.*

The weights of the contributing cameras, for use in our method, are found in Figure 4.12. The problematic areas of sampling mismatches in cameras 1 and 4 are assigned very low weight values. Likewise, the areas subjected to projection displacement of the façade are assigned low weights compared to Camera 4, which has the most straight-on view.



(a) *Camera 1* (b) *Camera 2* (c) *Camera 3* (d) *Camera 4*

Figure 4.12: *Individual camera weight maps of the* Bucket *view in our method. Red corresponds to a weight of* 1*, yellow corresponds to a weight of* 32*.*

The complete rendering with our method can be seen in figure 4.13b. Figure 4.13 shows the comparison of the *Bucket* view rendered with the naive method and our method, respectively. In the rendering using our method, the sampling mismatches are almost not visible at all, and the blurriness caused by projection displacement has nearly been eliminated. The texture on the handle of the bucket is also more distinguishable, as opposed to the naive rendering; the handle itself still looks bad because the voxels of the scene geometry are larger than the handle in the original geometry, in effect undersampling it.

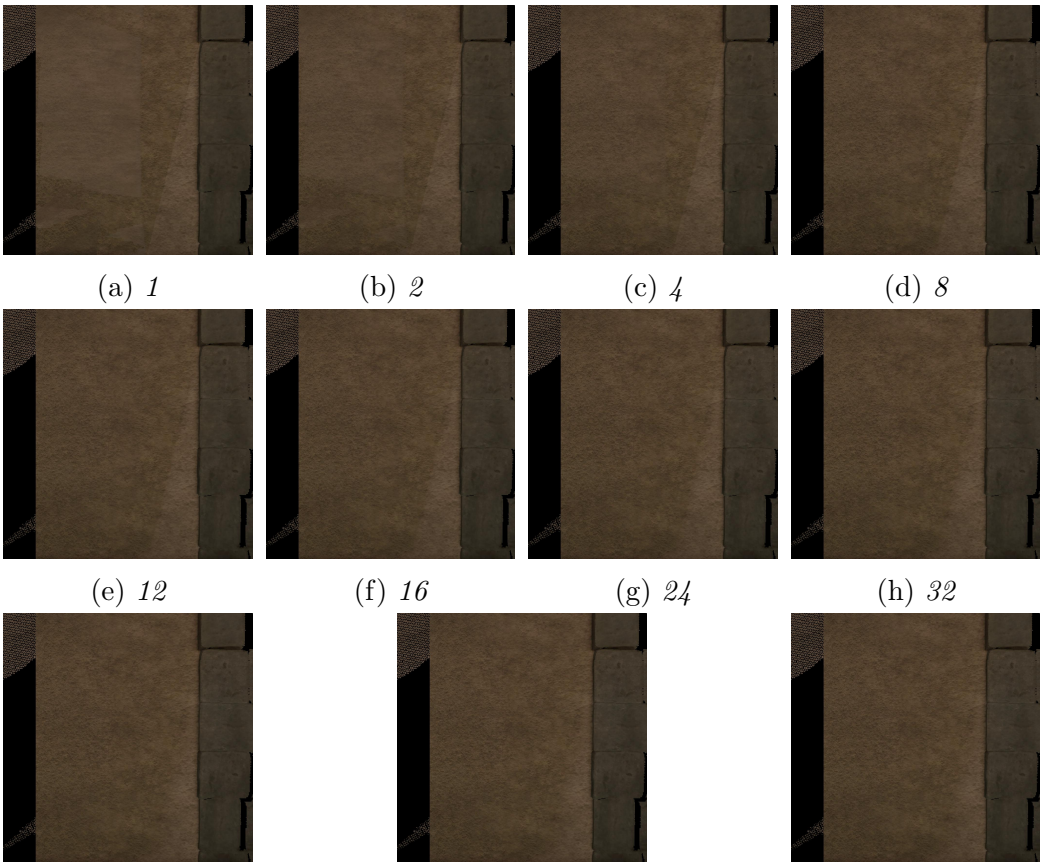
(a) Bucket *view rendered using the naive method.* (b) Bucket *view rendered using our method.*

Figure 4.13: *Comparison of the naive rendering and thesis rendering for the* Bucket *view*

4.1.5 Max distance size comparison

The chosen max distance affects the end result of our method. A smaller max distance confers a narrower gradient between the edge of an area of visibility and the part of the area that is considered in full view.

Figure 4.14 shows the *Wall* view rendered using our method with a max distance of 1, 2, 4, 8, 12, 16, 24, 32, 128, and ∞, respectively. With a max distance of 1, all areas of visibility will have a uniform weight of 1, and thus, our method will become identical to the naive method. The difference from max distance 1 to 32 is noticeable, but higher values than 32 contribute little to the end result, and from 64 and upwards, the difference is almost zero.



(a) *1* (b) *2* (c) *4* (d) *8*

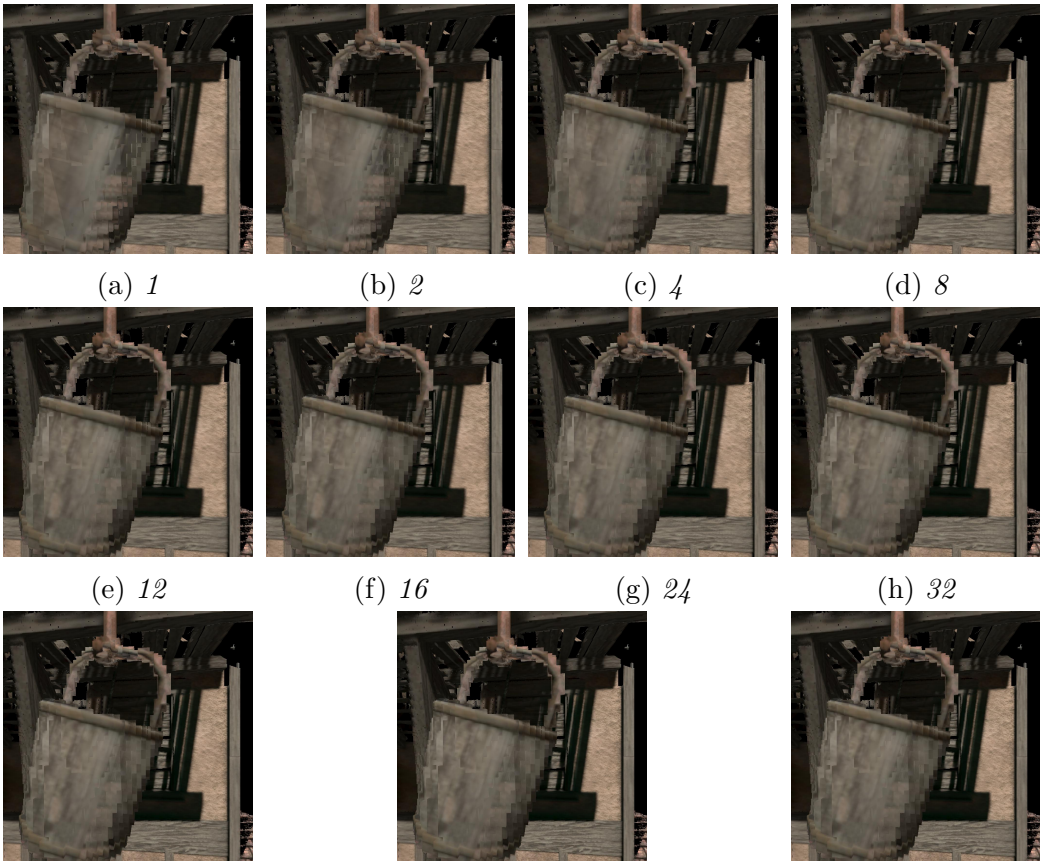
(e) *12* (f) *16* (g) *24* (h) *32*

(i) *64* (j) *128* (k) ∞

Figure 4.14: *Comparison of different values of max distance for the* Wall *view*

Another example of varying max distance is found in Figure 4.15, showing the *Bucket* view with the same max distance values as in Figure 4.14. The

sampling mismatch artifacts on the bucket itself does not vanish more after max distance 16, and the only thing changing at higher values than that is the blurriness caused by projection displacement, which sees no further improvement above max distance 64.



(a) *1* (b) *2* (c) *4* (d) *8*

(e) *12* (f) *16* (g) *24* (h) *32*

(i) *64* (j) *128* (k) ∞

Figure 4.15: *Comparison of different values of max distance for the* Bucket *view*

4.2 Performance

Apart from precomputing time, of which the naive method does not require any, the rendering time is slightly longer with our method. We can assign relative speed values for the methods by comparing the render times. Table 4.1 details the total rendering times of six different views (four of which comprise the views presented in Section 4.1) for the naive method and our method, respectively. Each view was rendered 50 times to provide some statistical

soundness. The rendering was performed on a Windows 7 desktop computer with an Intel i7-4770K CPU, 16 GB RAM, and an NVIDIA GeForce GTX 770 GPU. The implementation itself is a CPU-only raycaster with very few optimizations. As Table 4.1 shows, our method has a relative speed value at lower resolutions around 0.8, indicating that it runs at 80% of the speed of the naive method, but the difference becomes smaller as the resolution increases. The fact that our method is performing very slightly better than the naive method at higher resolutions can be attributed to runtime uncertainties.

Rendering size	128^2	256^2	512^2	768^2
Rendering time (Naive)	00:07:33	00:34:07	02:54:29	06:16:14
Rendering time (Our)	00:09:24	00:39:55	02:48:20	06:09:02
Relative speed (Our)	0.803	0.854	1.037	1.02

Table 4.1: *Rendering times and relative speeds for the naive method and thesis method, respectively.*

The preprocessing time varies depending on many variables, such as the scene, number of cameras, and max distance value. Table 4.2 shows how the preprocessing time correlates to varying values of max distance. As can be seen, a low max distance value confers faster preprocessing. This is because the algorithm can quit early if no discontinuities are found within the max distance. Since the preprocessing algorithm is not optimized for speed, the preprocessing times could possibly be improved significantly with optimizations and GPU acceleration.

Max Distance	**Running Time (hh:mm:ss)**
1	00:02:22
2	00:02:27
4	00:02:48
8	00:04:06
12	00:06:00
16	00:08:52
24	00:14:20
32	00:20:21
64	00:44:25
128	01:18:18
∞	01:41:07

Table 4.2: *Preprocessing times for a single frame using different values of max distance.*

Chapter 5

Discussion

While all tested views show significant improvement with our implemented method over the naive method, some show less improvement than others. In particular, the *Woman* view does not display the same improvement as the other ones, because of the fact that the geometry of the woman is relatively thin, combined with the unfavorable positions of the cameras in the scene. This shows that we might benefit from cameras dedicated to the actors in the scene, as people are more prone to catch visual errors in other humans than in walls and trees.

Renderings made using our implemented method looks much better than when using the naive method, with only minimal performance loss. One downside to our method is that the weights need to be saved alongside the video streams, increasing the disk size of the processed FVV sequence. The weights could be encoded as single-channel video streams to reduce size, but the aesthetical consequences of that remains to be tested. If a faster algorithm is developed, it would also be possible to calculate the weights in real-time, alleviating the need for saved weights altogether.

Our implemented method might need to be tweaked somewhat when used with other scenes and setups: Scene size, number of cameras, voxel resolution, and camera resolution all change the conditions under which the algorithm operates. One variable that needs to be changed according to the situation is the discontinuity threshold, which is the lowest value by which the distance of two neighboring pixels must differ for them to be considered discontinuous. With changes in scene size and voxel resolution, the discontinuity threshold will probably have to be tweaked to obtain a good compromise between not registering separate objects as the same, despite them being very close to each other, and recognizing an angled wall to be a single object. Another variable that will need tweaking is the max distance, which determines the maximum weight a pixel can be assigned to have to the nearest discontinuity.

The max distance will have to be selected in such a way as to balance visual quality and preprocessing time. A good lowest value for the max distance for this scene can be estimated by comparing the different figures in section 4.1.5; 32 seems to be the value at which most of the seams and other artifacts vanish enough to be indistinguishable. At 64, almost no differences with max distance ∞ can be seen. Between 32 and 64 seems to be a value for max distance; maintaining high quality while still keeping the processing times relatively short, as seen in table 4.2.

Given more time, we would have evaluated some methods for finding approximate surface normals from the voxel geometry. This would have made it easier to incorporate some methods from the previous work into this thesis, such as calculating weights based on how the scene camera's view direction compares to surface normals in the scene.

Chapter 6

Conclusion

The main question in this thesis was how to find color values for the surfaces of a voxel geometry in such a way that minimizes seams and other graphical artifacts.

Given the constraints of a scene of pure voxel data, without any information of surface normals, and four cameras with video streams, we proposed three slightly different methods. Each method builds on the common concept of blending color information from different cameras based on specific weights. The weights are stored in a weight map, which is a broad term for weights saved in a way that corresponds to the geometry, the current view being rendered, or the views of the cameras in the scene. The weights are generated in such a way that influence from a camera fades out near the edge of the areas of visibility (geometrically continuous areas completely in view of a certain camera) that camera has on the geometry.

The result is a method that succeeds in minimizing graphical artifacts within the possibilities of the given scene; it cannot create novel data where there was none.

Chapter 7

Future Work

An obvious contender for future improvement is optimization. There is a lot to be done in this regard, as almost none of it has been done in this thesis. Speeding up intersection tests, raycasting on the GPU, and using more efficient data structures, to name a few. A more efficient and/or approximative weight generation algorithm, for example, could help the method described in section 3.2.3 get rid of its precomputation needs, calculating all weights in real-time.

One interesting future approach is to evaluate the possibilities of implementing view-dependent surfaces in the context of this thesis. This would aid the realism of the rendering.

In the current suggested methods, all cameras that can be used for contributing to a point does so, even if it is a minuscule amount. Investigating the possibility of only having a certain number of cameras provide color information to a surface point could prove beneficial, as some areas are assigned very similar weights for all cameras because of their small size. The weighted averaging on these areas will then take bad cameras into consideration more than they would have if the surface area was larger.

Testing the suggested method's temporal consistency is a necessary step toward using it in real FVV applications.

Bibliography

[1] Peter Rander et al. "Virtualized Reality: Constructing Time-varying Virtual Worlds from Real World Events". In: *Proceedings of the 8th Conference on Visualization '97*. VIS '97. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, 277–ff. ISBN: 978-1-58113-011-9. URL: http://dl.acm.org/citation.cfm?id=266989.267081.

[2] *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. blender.org. URL: https://www.blender.org (visited on 12/06/2015).

[3] Patrick H. Kelly et al. "An Architecture for Multiple Perspective Interactive Video". In: *Proceedings of the Third ACM International Conference on Multimedia*. MULTIMEDIA '95. New York, NY, USA: ACM, 1995, pp. 201–212. ISBN: 978-0-89791-751-3. DOI: 10.1145/217279.215267. URL: http://doi.acm.org/10.1145/217279.215267.

[4] Toshiharu Horiuchi et al. "Interactive Music Video Application for Smartphones Based on Free-viewpoint Video and Audio Rendering". In: *Proceedings of the 20th ACM International Conference on Multimedia*. MM '12. New York, NY, USA: ACM, 2012, pp. 1293–1294. ISBN: 978-1-4503-1089-5. DOI: 10.1145/2393347.2396449. URL: http://doi.acm.org/10.1145/2393347.2396449.

[5] Takayoshi Koyama et al. "Live 3D Video in Soccer Stadium". In: *ACM SIGGRAPH 2003 Sketches & Applications*. SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 1–1. DOI: 10.1145/965400.965463. URL: http://doi.acm.org/10.1145/965400.965463.

[6] Joel Carranza et al. "Free-viewpoint Video of Human Actors". In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 569–577. ISBN: 1-58113-709-5. DOI: 10.1145/1201775.882309. URL: http://doi.acm.org/10.1145/1201775.882309.

[7] Stefan Hauswiesner et al. “Free Viewpoint Virtual Try-on with Commodity Depth Cameras”. In: *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*. VRCAI ’11. New York, NY, USA: ACM, 2011, pp. 23–30. ISBN: 978-1-4503-1060-4. DOI: 10.1145/2087756.2087759. URL: http://doi.acm.org/10.1145/2087756.2087759.

[8] Saied Moezzi et al. “Virtual View Generation for 3D Digital Video”. In: *IEEE MultiMedia* 4.1 (Jan. 1997), pp. 18–26. ISSN: 1070-986X. DOI: 10.1109/93.580392. URL: http://dx.doi.org/10.1109/93.580392.

[9] Chris Buehler et al. “Unstructured Lumigraph Rendering”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: ACM, 2001, pp. 425–432. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383309. URL: http://doi.acm.org/10.1145/383259.383309.

[10] Alvaro Collet et al. “High-quality Streamable Free-viewpoint Video”. In: *ACM Trans. Graph.* 34.4 (July 2015), 69:1–69:13. ISSN: 0730-0301. DOI: 10.1145/2766945. URL: http://doi.acm.org/10.1145/2766945.

[11] Jordi Salvador et al. “From Silhouettes to 3D Points to Mesh: Towards Free Viewpoint Video”. In: *Proceedings of the 1st International Workshop on 3D Video Processing*. 3DVP ’10. New York, NY, USA: ACM, 2010, pp. 19–24. ISBN: 978-1-4503-0159-6. DOI: 10.1145/1877791.1877797. URL: http://doi.acm.org/10.1145/1877791.1877797.

[12] Akio Ishikawa et al. “Free Viewpoint Video Generation for Walk-through Experience Using Image-based Rendering”. In: *Proceedings of the 16th ACM International Conference on Multimedia*. MM ’08. New York, NY, USA: ACM, 2008, pp. 1007–1008. ISBN: 978-1-60558-303-7. DOI: 10.1145/1459359.1459553. URL: http://doi.acm.org/10.1145/1459359.1459553.

[13] Cesar Palomo and Marcelo Gattass. “An Efficient Algorithm for Depth Image Rendering”. In: *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry*. VRCAI ’10. New York, NY, USA: ACM, 2010, pp. 271–276. ISBN: 978-1-4503-0459-7. DOI: 10.1145/1900179.1900236. URL: http://doi.acm.org/10.1145/1900179.1900236.

[14] Peiming Wang et al. “View Synthesis Framework for Real-time Navigation of Free Viewpoint Videos”. In: *Proceedings of International Conference on Internet Multimedia Computing and Service*. ICIMCS ’14.

New York, NY, USA: ACM, 2014, 427:427–427:430. ISBN: 978-1-4503-2810-4. DOI: 10.1145/2632856.2632943. URL: http://doi.acm.org/10.1145/2632856.2632943.

[15] Junya Kashiwakuma et al. "A Virtual Camera Controlling Method Using Multi-touch Gestures for Capturing Free-viewpoint Video". In: *Proceedings of the 11th European Conference on Interactive TV and Video*. EuroITV '13. New York, NY, USA: ACM, 2013, pp. 67–74. ISBN: 978-1-4503-1951-5. DOI: 10.1145/2465958.2465961. URL: http://doi.acm.org/10.1145/2465958.2465961.

[16] Ahmed Hamza and Mohamed Hefeeda. "A DASH-based Free Viewpoint Video Streaming System". In: *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. NOSSDAV '14. New York, NY, USA: ACM, 2014, 55:55–55:60. ISBN: 978-1-4503-2706-0. DOI: 10.1145/2578260.2578276. URL: http://doi.acm.org/10.1145/2578260.2578276.

[17] Dan Miao et al. "Resource Allocation for Cloud-based Free Viewpoint Video Rendering for Mobile Phones". In: *Proceedings of the 19th ACM International Conference on Multimedia*. MM '11. New York, NY, USA: ACM, 2011, pp. 1237–1240. ISBN: 978-1-4503-0616-4. DOI: 10.1145/2072298.2071983. URL: http://doi.acm.org/10.1145/2072298.2071983.

[18] Viktor Kämpe et al. "Exploiting Coherence in Time-varying Voxel Data". In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '16. New York, NY, USA: ACM, 2016, pp. 15–21. ISBN: 978-1-4503-4043-4. DOI: 10.1145/2856400.2856413. URL: http://doi.acm.org/10.1145/2856400.2856413.

[19] James Imber et al. "Free-viewpoint Video Rendering for Mobile Devices". In: *Proceedings of the 6th International Conference on Computer Vision / Computer Graphics Collaboration Techniques and Applications*. MIRAGE '13. New York, NY, USA: ACM, 2013, 11:1–11:8. ISBN: 978-1-4503-2023-8. DOI: 10.1145/2466715.2466726. URL: http://doi.acm.org/10.1145/2466715.2466726.

[20] Jonathan Starck and Adrian Hilton. "Model-Based Multiple View Reconstruction of People". In: *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*. ICCV '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 915–. ISBN: 978-0-7695-1950-0. URL: http://dl.acm.org/citation.cfm?id=946247.946683.

[21] Paul E. Debevec et al. “Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-based Approach”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 11–20. ISBN: 978-0-89791-746-9. DOI: 10.1145/237170.237191. URL: http://doi.acm.org/10.1145/237170.237191.

[22] Jonathan Cohen et al. “Appearance-preserving Simplification”. In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’98. New York, NY, USA: ACM, 1998, pp. 115–122. ISBN: 978-0-89791-999-9. DOI: 10.1145/280814.280832. URL: http://doi.acm.org/10.1145/280814.280832.

[23] Marco Volino and Adrian Hilton. “Layered View-dependent Texture Maps”. In: *Proceedings of the 10th European Conference on Visual Media Production*. CVMP ’13. New York, NY, USA: ACM, 2013, 16:1–16:8. ISBN: 978-1-4503-2589-9. DOI: 10.1145/2534008.2534022. URL: http://doi.acm.org/10.1145/2534008.2534022.

[24] V. Lempitsky and D. Ivanov. “Seamless Mosaicing of Image-Based Texture Maps”. In: *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR ’07.* IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR ’07. June 2007, pp. 1–6. DOI: 10.1109/CVPR.2007.383078.

[25] Viktor Kämpe et al. “High Resolution Sparse Voxel DAGs”. In: *ACM Trans. Graph.* 32.4 (July 2013), 101:1–101:13. ISSN: 0730-0301. DOI: 10.1145/2461912.2462024. URL: http://doi.acm.org/10.1145/2461912.2462024.

[26] Anat Levin et al. “Seamless Image Stitching in the Gradient Domain”. In: *Computer Vision - ECCV 2004*. Ed. by Tomás Pajdla and Jiří Matas. Lecture Notes in Computer Science 3024. DOI: 10.1007/978-3-540-24673-2_31. Springer Berlin Heidelberg, May 11, 2004, pp. 377–389. ISBN: 978-3-540-21981-1 978-3-540-24673-2. URL: http://link.springer.com/chapter/10.1007/978-3-540-24673-2_31.

[27] *The Open Toolkit Library — OpenTK*. URL: http://www.opentk.com/ (visited on 12/06/2015).