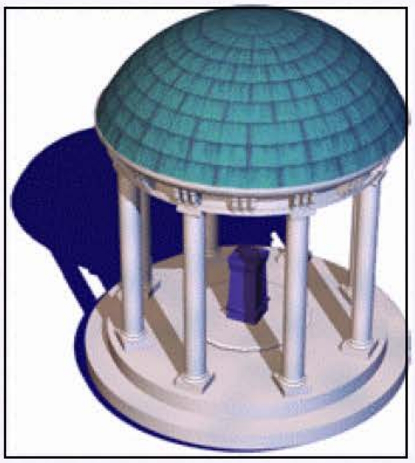
Forward Rasterization: A Reconstruction Algorithm for Image-Based Rendering

TR01-019
May 29, 2001

Voicu Popescu



Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

FORWARD RASTERIZATION:

A RECONSTRUCTION ALGORITHM FOR IMAGE-BASED RENDERING

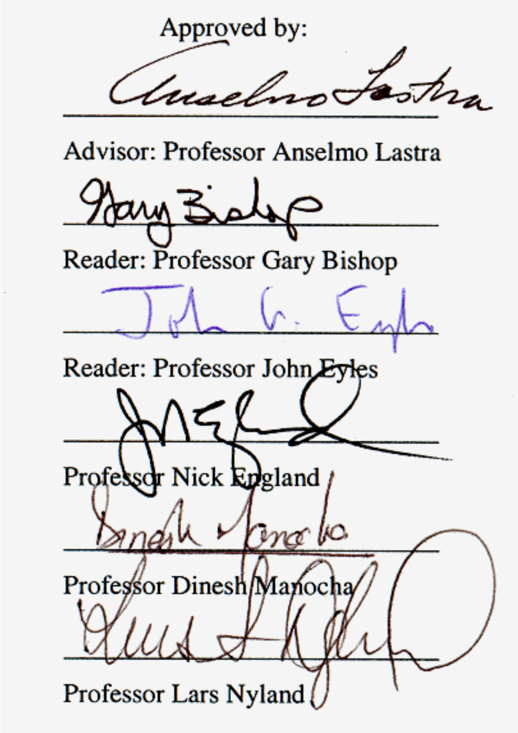
by

Voicu Sebastian Popescu

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2001



Approved by:

Anselmo Lastra

Advisor: Professor Anselmo Lastra

Gary Bishop

Reader: Professor Gary Bishop

John G. Eyles

Reader: Professor John Eyles

Professor Nick England

Dinesh Manocha

Professor Dinesh Manocha

Professor Lars Nyland

ABSTRACT

Voicu Sebastian Popescu

Forward Rasterization: a Reconstruction Algorithm for Image-Based Rendering

(Under the direction of Professor Anselmo Lastra)

In a recent alternative research path for interactive 3D graphics the scene to be rendered is described with images. Image-based rendering (IBR) is appealing since natural scenes, which are very difficult to model conventionally, can be acquired semi-automatically using cameras and other devices. Also IBR has the potential to create high-quality output images if the rendering stage can preserve the quality of the reference images (photographs). One promising IBR approach enhances the images with per-pixel depth. This allows reprojecting or *warping* the color-and-depth samples from the reference image to the desired image. Image-based rendering by warping (IBRW) is the focus of this dissertation.

In IBRW, simply warping the samples does not suffice for high-quality results because one must reconstruct the final image from the warped samples. This dissertation introduces a new reconstruction algorithm for IBRW. This new approach overcomes some of the major disadvantages of previous reconstruction methods. Unlike the *splatting methods*, it guarantees surface continuity and it also controls aliasing using a novel filtering method adapted to forward mapping. Unlike the *triangle-mesh method*, it requires considerably less computation per input sample, reducing the rasterization-setup cost by a factor of four.

The algorithm can be efficiently implemented in hardware and this dissertation presents the *WarpEngine*, a hardware architecture for IBRW. The WarpEngine consists of several identical nodes that render the frame in a sort-first fashion. Sub-regions of the depth images are processed in parallel in SIMD fashion. The WarpEngine architecture promises to produce high-quality high-resolution images at interactive rates.

This dissertation also analyzes the suitability of our approach for polygon rendering and proposes a possible algorithm for triangles. Finding the samples that must be warped to generate the current frame is another very challenging problem in IBRW. The dissertation introduces the *vacuum-buffer* sample-selection method, designed to ensure that the set of selected samples covers every visible surface.

To Andreea and to our daughter

ACKNOWLEDGEMENTS

For their inspired and knowledgeable help, their generous patience and their essential encouragement, many thanks to Anselmo Lastra, John Eyles, Gary Bishop, Lars Nyland, Nick England, John Poulton, Joshua Steinhurst, David McAllister, Daniel Aliaga, Leonard McMillan, Dinesh Manocha, Mary Whitton, Jeannie Walsh, David Plaisted, and John Thomas. This work would not have been possible without support from DOE, DARPA, NSF, UNC CS Alumni Association, Link Foundation, Intel and Microsoft.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Thesis

This chapter will first introduce the reader to the problem addressed by the present dissertation. The thesis of the dissertation is stated next followed by a thesis demonstration summary.

1.1 Motivation

1.1.1 The problem of interactive 3D computer graphics

The first great achievement in the field of interactive 3D computer graphics consisted of successfully *suggesting* a 3D scene to the user. Although the scene was quite simple and the interaction between matter and light was crudely approximated, the fundamental laws of physics such as perspective projection and occlusion-visibility were respected. The user, heavily relying on his imagination and prior life experience, correctly associated cuboid-and-prism objects to be houses, *red* little spheres to be apples and so on. In order to reach this stage fabulous inventions were required like the framebuffer, fast geometry processors, the z-buffer, fast rasterizers.

The suggestion stage enabled numerous interactive 3D graphics applications in visualization, training, and entertainment. However, it was obvious that increased realism of the images would greatly benefit to all these applications. The special look-and-feel of computer generated imagery does not have any intrinsic value. It is an approximation, and, as with all approximations, one desires to reduce the differences when compared to the original as much as possible. The ironic expression "it looks like computer graphics" was coined to indicate the lack of realism of computer generated images due to all the drastically simplifying assumptions used during rendering. The interactive 3D graphics researchers were unanimous: the next level of the art is achieving the perfect *illusion* for an exploring user. This level has not been reached yet. Generating images, at interactive rates, that can be mistaken for photographs is still an open problem.

Some researchers have sacrificed interactivity and focused just on obtaining photorealistic results. The physical phenomena associated with image generation are fairly well understood and, with no upper bound on the amount of per-frame computation, images that look like photographs are successfully generated. Most 3D graphics applications require interactivity; thus numerous efforts were made to improve the quality of rendered images while maintaining high refresh rates.

At the beginning, graphics hardware could render only simple geometric models that only vaguely resembled the scene they modeled. Thus, much effort was concentrated on increasing the rendering power. Bottlenecks like fill-rate (the time required to set the color of pixels inside a surface to the color of the surface), transformation (computing the new screen positions of the objects to give the user the illusion of moving), and primitive sorting (communication costs between various components of the graphics hardware) were alleviated in turn. As more geometric primitives could be processed each second, new obstacles in achieving high-quality interactive rendering were encountered:

- the manual creation of the scene model is a very laborious task that requires minute description of object shapes, materials and lights;
- the cost of representing all surface details with geometric primitives is prohibitive.

Also rendering hardware will probably never be powerful enough to render the entire model of a complex scene at each frame. A more practical approach is to find and adapt the subsets of the model that are necessary for the current view. Unfortunately, finding and adapting the model subsets is a very challenging problem.

Images can help alleviate these problems. They were first used in 3D computer graphics for rendering flat-surface detail in the classic technique of texture mapping. In texture mapping, photographs or previously rendered images are pasted onto polygons. For example the label of a bottle is easily rendered by texture mapping without resorting to numerous polygons (triangles) to model the drawings and text of the label.

In the last few years the usage of images in 3D graphics has been extended. Numerous modeling and rendering methods have been developed that rely on images, creating the sub-field of image-based rendering (IBR). One of the great appeals of IBR is the potential to alleviate the modeling bottleneck as images (photographs) can be easily acquired. Early IBR results enabled quick modeling and then visualization (albeit from select locations) of natural scenes that could not have even been modeled before, and all this with modest hardware.

IBR techniques that allowed full range of motion followed. A radical method relies on building a database of color samples (rays) that is queried at rendering time. One of the advantages of the method is that a (static) scene can be acquired with just a calibrated camera. Another advantage is the simplicity of the rendering algorithm, the rays of the desired image are readily available in the database. However the method is impractical since the database grows too large for interesting scenes. Storage and bandwidth requirements are, at least for now, orders of magnitude too high.

McMillan and Bishop propose enhancing images with per pixel depth [McMillan95]. A color and depth sample can be easily reprojected onto the desired image. The reprojection of the samples, called 3D warping, is essentially a mechanism of reusing the original rays in many desired images, achieving a significant compression when compared to the ray-database method. Image-based rendering by warping (IBRW) is discussed in more detail in the next subsection.

1.1.2 The problem of image-based rendering by warping

If enough photographs of a scene are taken, every surface that can ever become visible to an exploring user is captured in at least one of the photographs. Photographs are obviously very high-quality renderings of the surfaces, thus, assuming that the color of a same surface does not change too much from the photograph (reference image) to the desired image, it seems that all the information required for high-quality renderings is available. An IBRW application must solve three problems:

1. find samples in the reference images for all surfaces visible in the desired image
2. project them in the desired image
3. reconstruct the desired image

McMillan and Bishop presented an elegant solution for the reprojection problem, which I will review in the next subsection. The first and third problems had no or unsatisfactory solutions. The present dissertation focuses on the desired-image-reconstruction problem but it also addresses the sample-finding problem. The algorithm developed for reconstruction can be efficiently implemented in hardware and the dissertation presents the WarpEngine, a hardware architecture for image-based rendering that relies on the algorithm developed.

1.1.2.1 3D image warping

A necessary and sufficient condition for reprojecting the samples is to know the reference- and desired-image camera views and the depth of each sample. The reference image camera parameters define the 3D ray along which the sample lies, the depth marks the position of the sample along the ray and the camera parameters of the desired view allow projection from 3D to the desired image plane. These transformations are elegantly condensed in the 3D warping equations shown in **Equation 1.1**.

$$u_2 = \frac{w_{11} + w_{12} \cdot u_1 + w_{13} \cdot v_1 + w_{14} \cdot d(u_1, v_1)}{w_{31} + w_{32} \cdot u_1 + w_{33} \cdot v_1 + w_{34} \cdot d(u_1, v_1)}$$

$$v_2 = \frac{w_{21} + w_{22} \cdot u_1 + w_{23} \cdot v_1 + w_{24} \cdot d(u_1, v_1)}{w_{31} + w_{32} \cdot u_1 + w_{33} \cdot v_1 + w_{34} \cdot d(u_1, v_1)}$$

Equation 1.1 3D warping equations.

- u_2, v_2 are the desired image coordinates,
- u_1, v_1 the original (reference) image coordinates,
- the w's are transformation constants obtained from the reference and desired image camera parameters,
- $d(u_1, v_1)$ is the generalized disparity at sample (u_1, v_1), which is defined as the ratio between the distance to the reference image plane and $z_{\text{eye}}(u_1, v_1)$.

The warping equation is equivalent to the vertex transformation operations commonly used in computer graphics, but allows one to take advantage of the regular structure of images to perform incremental transformation. The warped coordinates of a sample can be computed with six adds, five multiplies[1], and one divide.

Reconstructing by simply setting a desired image pixel to the color of the sample that warps within its boundary results in holes thus is not acceptable. Also, more than one visible sample can warp to

[1] If w_{34} is non-zero (there is a non-zero translation from the reference position) one can save a multiply by dividing all w's by w_{34}.

the same pixel, and simply discarding all but one sample produces aliasing. Reconstruction is a challenging task when warping images with depth.

1.1.2.2 Reconstruction

The 3D warping equation is a forward mapping that takes samples from the reference domain and maps them to the destination domain. This is fundamentally different from conventional texturing where an inverse mapping, from the desired image to the texture, is used to establish the color at a particular location in the desired image. An inverse mapping would be ideal for reconstruction since one could easily look up *all* desired image pixels *and* ensure proper filtering. Unfortunately there is no analytically computable inverse for 3D warping since the warped coordinates depend also on the depth of the input samples, which is a free variable. One could search for the reference sample that warps to a certain desired-image pixel by trying all possible depth values as described in [McMillan97], but the procedure is rather costly.

Reconstruction for IBRW has been done in one of two ways [Mark97, McMillan97]: with splats and with a polygonal mesh.

A splat [Westover90] is a representation of the projected shape of the reference sample. The original use for splats was to render transparency for volume rendering; thus the splats were blended in front-to-back order. For IBRW, we do not want to blend samples that are at different depths. Rather we want to overwrite samples that should be hidden and only blend samples that represent the same surface. This is very difficult to do because in IBRW we have no information about surfaces. To prevent samples of hidden surfaces from showing through, the sizes of the splats are overestimated [Shade98]. However, overlapping splats can incorrectly erase visible samples, resulting in aliasing. Such a case occurs when a splat covers completely a neighboring splat that originates from a sample that should be visible.

Good reconstruction can be obtained by connecting the samples of the reference image into a polygonal mesh. Not all samples should be connected, of course, and one has to first find where the mesh must be disconnected; a simple but robust solution to this problem is presented later in this chapter and then in more detail in Chapter 4. With meshing, continuity of the surfaces is maintained where desired, and hardware acceleration increases performance. Although the quality of the resulting images is good, the mesh method is not satisfactory. One reason is performance; another is efficiency.

Assume that 1280 by 1024 is the targeted resolution and that on average we warp twice the reference samples as the desired resolution. Two triangles must be rendered for every warped sample. The average number of triangles per second to sustain a frame rate of 30 Hz is given in **Equation 1.2**.

$$N \approx 1280 \times 1024 \times 2 \times 2 \times 30 \approx 157 Mtris / s$$

Equation 1.2 Estimate of the number of triangles that must be rendered every second when the mesh reconstruction method is used to generate XVGA output.

Neither high-end systems like PixelFlow [Molnar92] or InfiniteReality [Montrym97], nor the rapidly improving PC 3D graphics accelerators, can produce the necessary performance. Moreover, we speculate that it will take years for them to reach this sustained level of performance. Even then, we contend that it will take more hardware than for a machine optimized for IBRW.

The main motivation of this dissertation is the belief that there are reconstruction algorithms for IBRW with efficient hardware implementation that take advantage of the regularity of image-based primitives and of the small screen-size of the warped samples.

More must be said to explain the number of reference-image samples required at each frame, used in **Equation 1.2**. This number depends on the scene, on how it is modeled, and also on the performance of the algorithm used to find the necessary samples. One must process (on average) more than one reference image sample per desired image location because:

- there are surfaces that are redundantly captured in more than one reference image;
- there are surfaces captured in the reference images that are not visible in the desired image (depth complexity is greater than one);
- there are surfaces that were better sampled in the reference image than in the desired image, which leads to more than one visible sample per desired image pixel.

Two input samples per output pixel is a reasonable lower bound; in practice, I have found that it is difficult to use fewer. Chapter 6 describes an algorithm for finding the reference image samples needed for the desired image. With real-time depth-image updates (immediate mode), the number of samples will be determined by the number, resolution and update rates of the cameras.

1.2 Thesis Statement

The mesh method generates high-quality images because the triangles of the mesh are precisely rendered employing well-established antialiasing methods that guarantee sub-pixel accuracy. One would like to achieve the same precision in reconstructing the final image from the warped samples without paying the high cost of rasterizing two micro-triangles per sample.

Traditionally, triangle rasterization implies the following steps:

1. Setup:

   1.1. Compute desired-image plane discrete derivatives of rasterization parameters (color channels, depth for z-buffering, etc.).

   1.2. Compute expressions used to determine which pixels are inside the triangle.

2. Process pixels inside the triangle (scan-conversion):

   2.1. Increment previous pixel parameters to obtain current pixel parameters.

   2.2. If not visible, continue.

   2.3. If visible, set and continue.

In the case of IBRW, the triangles are typically very small in screen-space (one pixel or less), so the setup cost cannot be amortized over a large number of interior pixels at step 2. Setup is expensive since it is in essence equivalent to solving an equation, or an inverse computation, in order to answer the question "what should the difference be between the *z* of two consecutive pixels?" The depth *z* in the previous question can be replaced by any of the other parameters such as color channels, texture coordinates. When the triangle is large, the effort spent answering these questions pays off since the result is used over and over again for all pixels interior to the triangle.

For clarity, traditional rasterization was described above omitting antialiasing. In the case of antialiasing using *k x k* regular super-sampling the parameter variations (derivatives) computed at step 1.1 are for *1 / k* desired image pixels; in the case of the widely adopted jittered supersampling method, the *j* sub-samples are located on a high-resolution *K x K regular* grid, and at step 1.1 the variations computed are for *1 / K* desired-image pixels. At step 2, all the sub-samples interior to the triangle are processed. Although the sub-samples are not located in the center of the pixel anymore, they are fixed with respect to the pixel boundaries. Rasterization is still done backward, from the destination domain (discrete desired-image plane locations) to the source domain (parameter space); for differentiation with the method I propose, I will call the traditional rasterization described above *backward-mapped rasterization*.

The *forward rasterization* method I introduce processes an entire quad directly and does not require splitting the quad into two triangles. It generates samples for the quad to be rasterized that are independent of the pixel grid of the desired image by simply bilinearly interpolating in parameter space. The interpolation can take place either before or after the perspective projection of the quad, which in the IBRW context means before or after the 3D warp. Chapter 7 analyses the suitability of forward rasterization to polygon rendering; for the remainder of this chapter the forward rasterization discussion will be in the context of IBRW, which is the main focus of this dissertation.

For further clarification and to begin to convince the reader that the proposed method works, here are succinct answers to important questions.

In the case of backward rasterization the entire desired-image area covered by the triangle projection is visibility tested and all hidden samples are overwritten. For forward rasterization, how can one ensure that continuity of surfaces is maintained and that back-surface samples do not show through? To prevent the problem described, in the case of forward rasterization enough samples have to be generated by interpolation. Choosing appropriate interpolation factors is described later but for now note that interpolation factors cannot be overestimated by much since it results in inefficiency.

The method described saves on the rasterization cost, but does it achieve the same high-quality, sub-pixel accurate reconstruction? Proper antialiasing requires several samples per desired-image pixel, thus the first step in insuring proper reconstruction is to warp into an intermediate buffer (*warpbuffer*) that is of higher resolution than the final image. Remember that interpolation is done in parameter space so the samples do not map exactly in the center of the warpbuffer locations. The higher the resolution of the warpbuffer, the smaller the warpbuffer locations and the smaller the truncation errors are. In order to avoid the large cost of a very dense warpbuffer a pair of *offsets* is used to more precisely record the point where the sample warped. After visibility is solved, the offsets are used at reconstruction as described later; two two-bit offsets and *2 x 2* supersampled warpbuffer produce very good reconstruction results.

The two samples that are compared in the z-buffer visibility test are not exactly coincident in the desired-image plane, that is the 3D samples do not belong to the exact same ray. Does this cause visibility artifacts? In rare cases, the problem can cause a farther sample to be incorrectly ruled as visible to the detriment of a closer sample. For such a case to occur, the two 3D surfaces that generate the pair of incorrectly z-buffered samples must be close to each other and at an angle with the desired-image plane. The samples for such surfaces are typically taken from images that sample the surfaces better than the desired-image thus several samples of each surface will land at the same warpbuffer location. This minimizes the chances of a visibility artifact since it is unlikely that *all* samples of the front surface are farther than the closest sample of the back surface. In practice I did not find this to be a problem and neither have researchers who have used splatting for reconstruction, which is also prone to the artifact described (even more so since no higher resolution warpbuffer was used).

Forward rasterization is a simple algorithm, consisting only of bilinear interpolation and a slightly modified convolution (that uses the offsets), showing potential for efficient hardware implementation.

My thesis:

> ***Image-based rendering by warping using forward rasterization, which means parameter-space interpolation and offset reconstruction, produces high-quality images and can be efficiently implemented in hardware.***

The next section presents a summary of the demonstration detailed in the other chapters of the dissertation.

1.3 Summary of Demonstration

The thesis makes two direct claims:

1. High-quality images of a 3D scene modeled with depth-images can be obtained using forward rasterization.
2. The method can be accelerated by efficient hardware.

The proof of the first claim begins with a detailed description of the rendering algorithm, which shows that the output image is computed according to principles that were proved correct in the literature and are generally accepted by practitioners as a guarantee for high-quality results. Then empirical evidence is presented, which consists of images generated with the proposed method for a variety of scenes; the high complexity of the scenes suggests that the method is general. The image quality is illustrated by the image itself; no quality metrics are used[2].

[2] No robust image-quality metrics exist for computer generated images; moreover, image-based rendered images exhibit different artifacts than the geometry-based rendered images, which makes the existing quality-metrics unusable.

The second claim will be sustained by presenting a hardware architecture that implements the algorithm described. Efficiency is demonstrated by showing the reduced number of arithmetical operations compared to the backward rasterization. Also the architecture is a single-chip design. Configurations with several identical nodes are capable of sustaining interactive framerates even in the case of high output resolutions.

It is important to note that the thesis makes an indirect third claim. The reader will remember that image reconstruction is just the third step of IBRW. A complete IBRW system also must solve the problem of finding the depth-and-color samples required for the current desired view, which, as stated earlier, is an open problem. Although not strictly required for the soundness of the thesis demonstration, I do address the problem of sample finding for completeness and in order to give the devised architecture practical usefulness.

The next three subsections elaborate demonstrations of the three claims.

1.3.1 Rendering Algorithm

One has to treat the depth image as connected (as in the mesh approach) in order to prevent samples of hidden surfaces from showing through. The triangles resulting from the mesh method are very small in screen space, thus scan-conversion time is dominated by setup. Instead of conventional, backward rasterization, I propose simply bilinearly interpolating between connected samples in the reference image domain, reducing *per-sample* setup.

The algorithm is:

```
For all adjacent, connected samples
    Bilinearly interpolate color and depth to obtain subsamples
    Warp resulting subsamples to desired image space
    Z-composite warped subsamples into the warpbuffer
```

In order to reduce aliasing, the warping is done into a sub-pixel resolution warpbuffer (usually 2x2), which is then filtered to produce the final image in the frame buffer. The subsamples are z-buffered. The interpolation factor (number of subsamples created in each of the *x* and *y* directions) is critical in order to ensure that (1) back surfaces do not show through, and (2) not too many subsamples are generated[3]. The computation of the interpolation factor is described in more detail in Chapter 4.

Recall that the reference-image depth information is stored as generalized disparity that is proportional to $1/z_{eye}$, which is linear in image space. Consequently, if the four neighboring samples are planar, the sub-samples resulting from the interpolation are correctly located on the same plane. If the samples are not coplanar, the sub-samples define a general bilinear patch. Adjacent patches exhibit C^0 continuity.

[3] Out of efficiency considerations, one must generate the smallest number of subsamples that meets condition (1).

An alternative rendering algorithm, to save the cost of warping the sub-samples (dominated by the inverse computation, see **Equation 1.1**) is to *first* warp the reference-image samples and *then* interpolate. This still avoids the triangle setup costs. Just as when interpolation is done in reference-image space, if the original samples lie on a plane, the sub-samples are also on the plane. However the similarity between the two methods ends when the four original samples are not coplanar and the resulting screen-space quad might be concave (resulting in a "bow tie", see **Figure 1.1**). The simulations show that this is a very infrequent case, which usually occurs between silhouette samples that were marked as disconnected anyway.

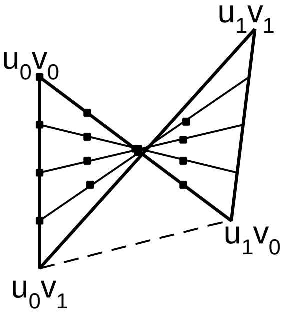


Figure 1.1 Four neighboring samples forming a concave quad when warped. Interpolating in desired-image space produces the sub-samples shown with little squares, which is different from the projection of the surface on the image plane. When interpolating in reference-image space and then warping, the surface projection is approximated better, but at higher computational cost.

1.3.1.1 Determining Connectivity

I developed an inexpensive way of detecting which samples should not be connected by interpolation, based on the surface curvature. At every reference image sample an approximation of the second derivative[4] of the generalized disparity is computed along four directions: E-W, SE-NW, N-S, and SW-NE. If the surface sampled is planar the second derivative is exactly zero. If it exceeds a threshold (which is unique per scene) the samples are marked as disconnected (see **Figure 1.2**).

1.3.1.2 Reconstruction Using Offsets

The goal is to render high-quality, antialiased images. Conventional jittered supersampling is not an option because of the *forward-mapping* nature of the warping process; warping produces sub-samples that do not correspond exactly with the centers of warpbuffer locations. Even with a 2x2 warpbuffer, aliasing is noticeable.

The proposed alternative is to compute the *(x, y)* location of the warp to a precision higher than that of the warpbuffer, and store that more precise location as an offset from the corner of the warpbuffer location. The *offsets* are used during reconstruction to obtain better filtering and a higher-quality final

[4] difference of neighboring differences

image. I have found that a 2-bit offset in each of *x* and *y* directions (total of 4 bits per warpbuffer location) provides good results. This, combined with a 2x2 warpbuffer, locates the warped subsamples to within one-eight of a pixel (see **Figure 1.3**). The results are illustrated in **Figure 1.4**.

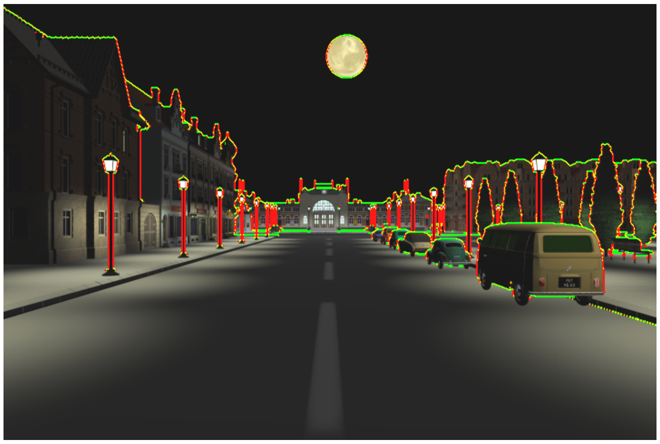


Figure 1.2 Depth-discontinuity detection in the reference image. The vertical and horizontal depth discontinuities are marked in green and red, respectively. On black and white copies the depth discontinuities appear as lighter pixels at silhouettes. The detection algorithm works well in spite of the great range of distances in the image.

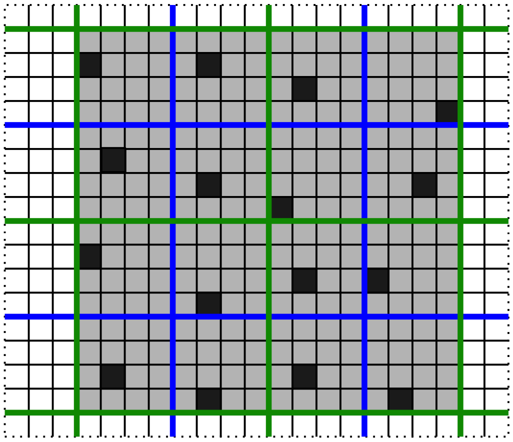


Figure 1.3 Efficient reduction of the truncation errors using offsets. In the warpbuffer fragment shown, the thick green (lighter) lines define the warpbuffer locations. The thick blue (darker) lines delimit the output pixels, which span four warpbuffer locations each. The fine black lines show the virtual subdivision of the warpbuffer locations corresponding to the two 2-bit offset values. The locations at which these samples warped are shown by black squares and are recorded to a precision of 1/8th of a pixel. There is exactly one sample per warpbuffer location. The output pixel is reconstructed using a two-pixel wide kernel, with a half pixel (one warpbuffer location) overlap. The kernel is shown in gray. The 16 color samples are weighted according to their position inside the warpbuffer location, as modified by the offsets. The reconstruction is equivalent to reconstructing from an 8x8 supersampled buffer that is sparsely populated, without having to explicitly allocate the dense buffer or to search for the locations that are populated.

Offsets are, of course, not entirely equivalent to a higher warpbuffer resolution. Although its location is recorded more precisely, only one sample is stored at each warpbuffer location. In the expected case, when the sampling resolution of the desired image is within a factor of two of that of the reference images, the 2x2 warpbuffer with 4x4 offsets provides a good reconstruction. Outside that interval other reference images should be used. One could increase the resolution of the warpbuffer to accommodate even bigger sampling mismatches, but this comes at a substantial additional cost, not only in memory, but also in warping since more reference image samples must be used.

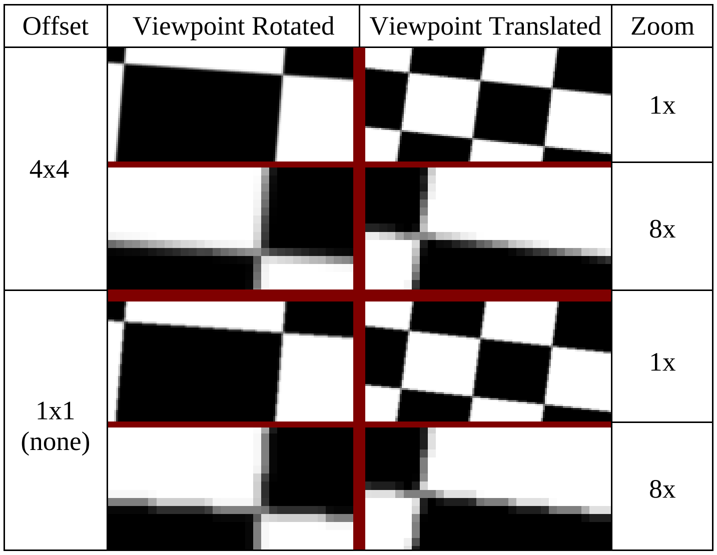


Figure 1.4 Antialiasing using offsets. These images were rendered from a depth image of a checkerboard. The left column is just rotated, the right also translated. We show both original and *8x* zoomed versions. All images used a *2 x 2* warpbuffer. The upper set was rendered with two bits for each of x and y offset. The lower set used no offsets and exhibits more aliasing.

The offset reconstruction also has good temporal antialiasing properties. Antialiasing by jittered supersampling or coverage-mask-based methods suffer from the problem of collinear sampling locations within a pixel. No matter how the sampling locations are chosen, at least two of them are collinear in jittered supersampling and *k* are collinear when *k x k* regular subpixel masks are computed. If from one frame to the next all collinear sampling locations move from one side to the other of a slowly moving edge, the change in color of the output pixel is too abrupt. Using 2x2 pixel kernels with 2x2 warpbuffer and 4x4 offsets guarantees 16 intermediate levels (when an edge moves slowly enough).

Figure 1.5, **Figure 1.6**, and **Figure 1.7** show images of various test scenes rendered with the method described.

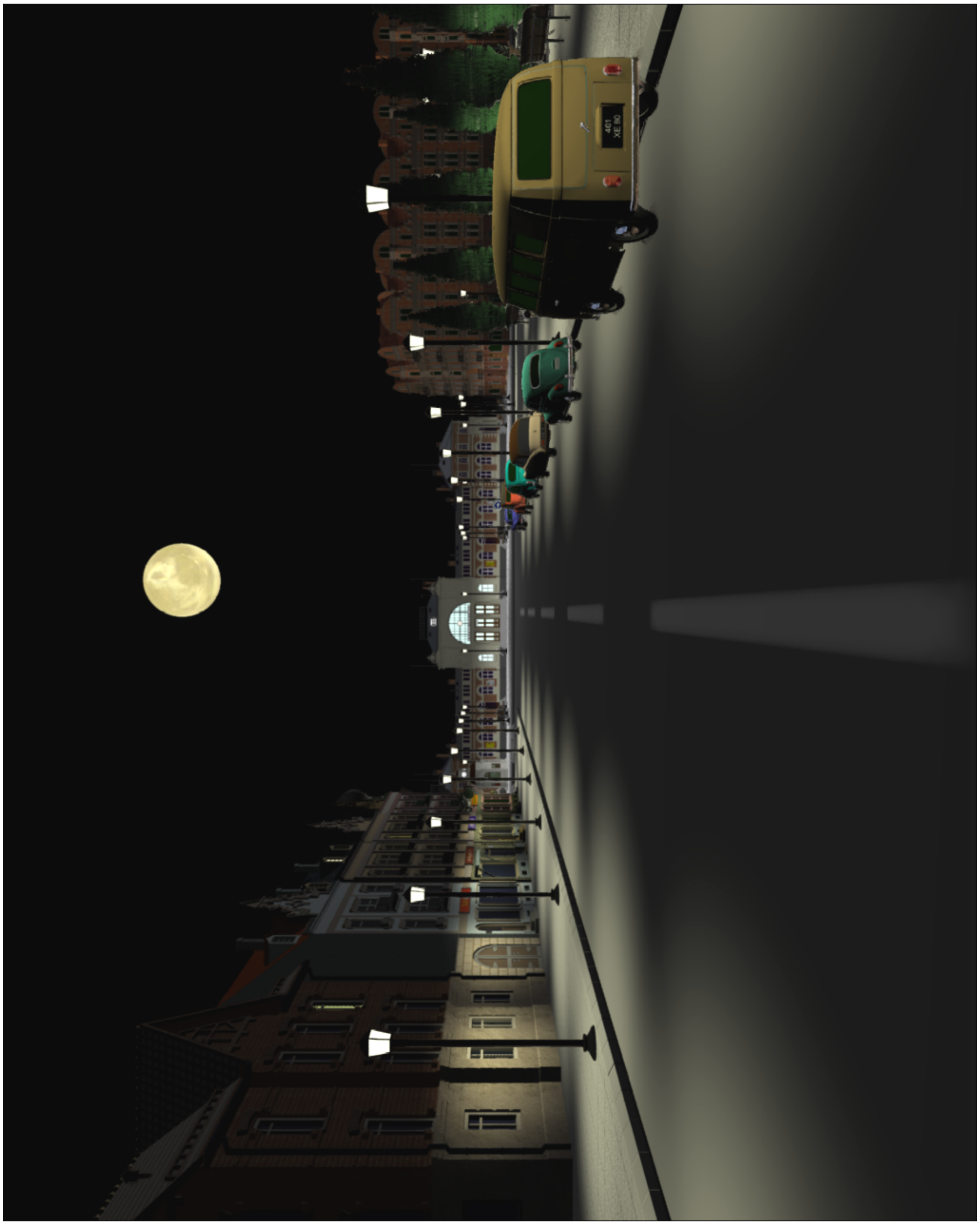


Figure 1.5 Image generated with the rendering method proposed. The complex town scene (*Eurotown*) is modeled with reference depth images placed on a regular grid and only the reference images of the current grid-cell were used to render each frame. The scene is synthetic: a geometric model was available and it was used to render the reference images.

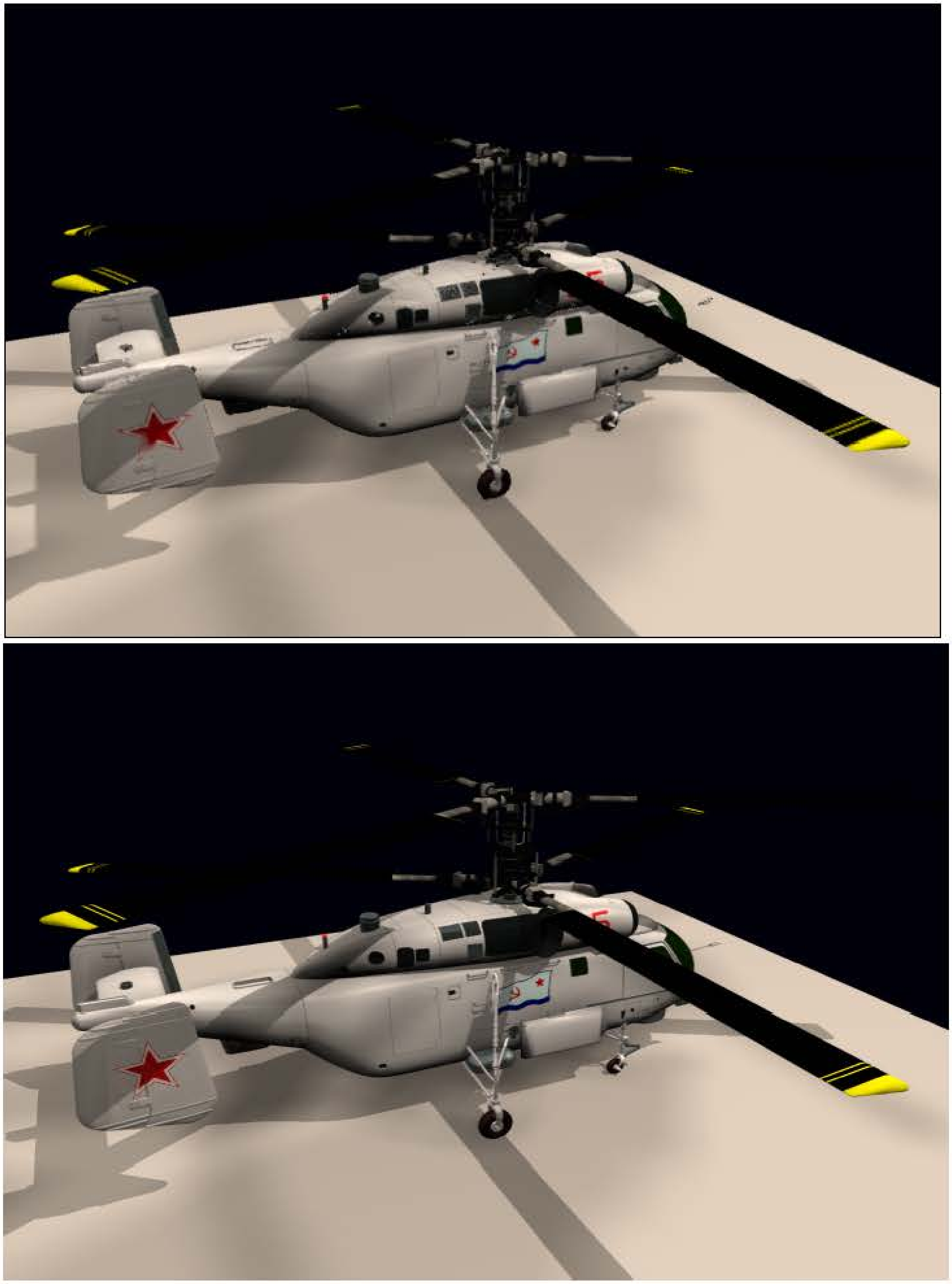


Figure 1.6 Example of image rendered using the forward rasterization method (top). For comparison, the bottom image was rendered directly from the geometric model.

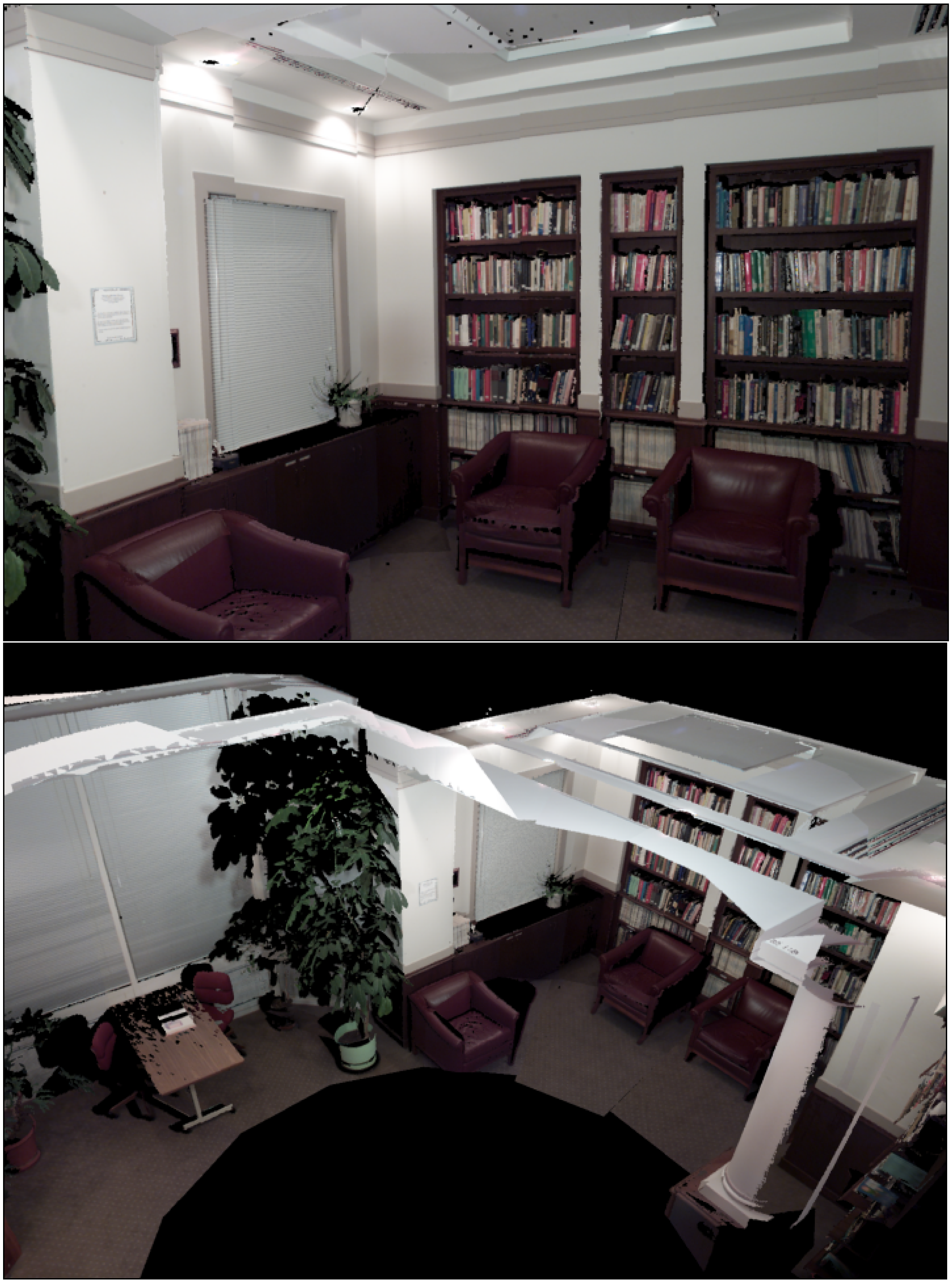


Figure 1.7 Rendering method tested on a natural scene. The reference depth-images of the reading room were created by registering color images with the range information acquired by our in-house laser rangefinder. The rangefinder captured data from two positions in the center of the room. We are missing some data, on the ceiling for example, but the surfaces for which data was available are rendered realistically.

1.3.2 The WarpEngine

The previous section shows that forward rasterization applied to IBRW produces high-quality results. This section describes the WarpEngine, which is a hardware architecture that implements this rendering algorithm. Devising the WarpEngine architecture was by no means my individual effort, but rather a collective accomplishment. John Eyles' major contributions stand out. Anselmo Lastra, Joshua Steinhurst, Nick England and Gary Bishop also contributed. Thus I will use the *plural* of the first person when describing the WarpEngine architecture throughout this dissertation.

1.3.2.1 Overview

The hardware architecture must provide sufficient warping power for all required reference-image samples and sufficient bandwidth to the warpbuffer. We decided to partition the reference images into 16x16-sample *tiles* (with a 1 sample overlap yielding a 15x15 payload) and to use these as the basic rendering primitive. Tiles provide several important advantages:

- we can selectively use portions of reference images as needed for adequate sampling and coverage of visible surfaces;
- one can easily estimate the screen area onto which a tile transforms, enabling efficient high-level parallelism;
- tiles are small enough that the same interpolation factor can be used for all samples, enabling SIMD low-level parallelism.

Warping and Interpolation

All the samples of a tile can be warped and interpolated with the same set of instructions so a SIMD implementation is, we believe, the most efficient. We opted for an array of simple byte-wide processors, similar to the one used in PixelFlow [Molnar92]. For a computation that can be efficiently mapped, a SIMD array provides efficient use of silicon, since control is factored out over all the processors. A large array of simple processors is more easily programmable than a complex pipelined processor. The programmability is necessary for use of the WarpEngine as a research tool.

A SIMD array equal in size to the reference-image-tile maps very efficiently, since the warping calculation is the same for every pixel, with minimal branching required. Nearest neighbor Processing Element (PE) connectivity provides each PE with access to the three other samples needed for interpolation.

Warpbuffer

The biggest design concern was providing sufficient warpbuffer bandwidth. We assume the maximum resolution to be HDTV (approximately 2K x 1K pixels) and 60 Hz update rate; we assume again that one must use at least two reference-image samples per output pixel. This implies that at least 240 million reference samples per second must be warped. In our simulations, a 2x2 warpbuffer resolution required in some cases an average interpolation factor of 4x4. Thus, for each warped reference-image pixel, 16 warped samples are generated, and the warpbuffer must process approximately 4 billion warped samples per second. Each sample is about 12 bytes in size (4 bytes RGB; 4 bytes Z-buffer; 4 bytes X and Y values, including offsets). Assuming a depth complexity of two, and that 50% of the hidden samples initially pass the Z-comparison test, an average of 10 byte accesses is required per warped sample. Thus total warpbuffer bandwidth is about 40 GigaBytes/sec.

To achieve this enormous warpbuffer bandwidth, a very large number of commodity DRAMs would be required (well over 100); similarly, the warping/interpolation processors would require hundreds of pins dedicated to interfacing with the warpbuffer. By placing the warpbuffer on-chip, that is, on the same ASIC as the processors that generate the warped samples, very wide and fast memory interfaces can be used.

Region-Based Rendering

With current technology, a single ASIC can provide neither sufficient processing power nor sufficient warpbuffer memory[5]. Thus multiple ASICs are required, and some form of high-level parallelism must be employed. Partitioning the warpbuffer into contiguous screen regions with each region assigned to an ASIC (screen-space subdivision) is appealing, because the typical 16x16-sample tile intersects only one screen region and therefore must be processed by a single ASIC (tiles that overlap region boundaries are assigned to multiple regions). By contrast, with interleaving, each tile would need to be processed by many or all of the ASICs.

For partitioning by screen-space subdivision, primitives must be sorted by screen region[6]. Using tiles as the rendering primitive means that sorting is performed on 256 samples at a time; the number of tiles per frame ranges from a few thousand to a maximum of a few tens of thousands (depending on screen resolution) so the computational and memory burden of sorting is considerably less than for the general polygon-rendering case. By assigning multiple screen regions to each ASIC, a smaller number of ASICs is sufficient; however this requires sorting into buckets corresponding to screen regions [Ellsworth97], because an ASIC must process all primitives in a given region before moving to its next assigned region.

The sort first, sort middle, sort last taxonomy developed to describe object-parallel polygon-rendering architectures [Molnar94], can also be applied to IBR architectures. Sorting by reference-image tiles is sort first from the point of view of reference-image samples, since after a tile has been assigned to a screen region, it is known a priori that its sub-samples will warp to the desired screen region (those that do not can be discarded, since the tile will be assigned to all pertinent regions). In polygon rendering, sort first

[5] As silicon technology improves, a full-sized warpbuffer becomes feasible (on an embedded-DRAM process).

[6] For tiles, this is efficiently done by warping the 4 corners, using both the tile’s minimum and maximum disparity values; the resulting 8 points define the tile’s screen-space bounding box.

[Mueller95] is prone to load-balancing difficulties; this is not be a problem for IBR, since reference-image tiles and interpolation factors are chosen to sample the destination image uniformly. We believe that sort first is an attractive approach for the WarpEngine, because it makes scaling of the system relatively painless. Performance is increased by adding additional ASICs, and assigning fewer screen regions to each ASIC. Screen-space subdivision requires a central processor, perhaps the host, which can perform the tile sorting, or a way of distributing these tasks across the multiple ASICs.

Processing Warped Sub-Samples

It is straightforward to build a region-sized on-chip warpbuffer with very high performance. Since each warped sample maps to only one location in the warpbuffer, the warpbuffer can be partitioned, with a *sample processor* assigned to each partition. Very high numbers of samples can be processed by instantiating more sample processors, processing simultaneous streams of warped samples. Load-balancing can be achieved by sub-pixel interleaving the partitions and providing input FIFOs for the sample processors. The region size is determined by the silicon budget for the warpbuffer, independently of the number of partitions.

The sample processors are very simple: they combine a new warped sample with the previous contents of the warpbuffer location, using a z-compare operation. Since the sample processors' memory interface does not cross chip boundaries, it can be very wide and very fast; thus the sample processors are not bandwidth limited.

1.3.2.2 WarpEngine Implementation

Our architecture, the *WarpEngine*, consists of one or more identical *Nodes* (typically 4 to 32); each Node consists of an ASIC and a Tile Cache. The ASIC contains:

- a <u>16x16 SIMD Warp Array</u>, for warping and interpolating reference-image samples;
- a <u>Region Accumulator</u>, which includes a double-buffered warp buffer for a 128x128 screen region and 4 sample processors for resolving visibility;
- a <u>Reconstruction Buffer</u>, for computing final pixel values;
- a <u>Network Interface</u>, which connects the Nodes together into a high-bandwidth ring, and provides a connection to the host, a connection to each of the Warp Arrays, and a connection to the <u>Tile Cache</u>.

The Tile Cache is a commodity DRAM device; it is used for caching both reference-image tiles and instructions. A double-buffered Frame Buffer receives the final pixel values from the Nodes for display.

The basic operation of the system is as follows (see **Figure 1.8**):

- The host determines which reference-image tiles are to be used to compute the destination image, and computes the screen-space bounding box for each of these tiles. For each screen region, the host maintains a *bin*; each bin contains pointers to the tiles whose bounding boxes intersect that screen region.

- For each screen region, the host assigns a Node to be responsible for that screen region. The host sends each tile in the region's bin to the Node. (Tiles are cached in each node's Tile Cache. If a tile is resident in one of the caches, the host instructs the Network Interface to forward it to the appropriate Node. If not, the host must send the tile data to the Node).
- Each tile received by each Node is loaded into the Warp Array, which performs the warping and interpolation calculations for the tile, and forwards the warped samples to the Region Accumulator.
- The Region Accumulator collects the warped samples into its sub-pixel resolution warp buffer.
- After all tiles in the region's bin have been processed, the Region Accumulator swaps its buffers and initializes the visibility buffer, in preparation for processing the next screen region.
- Concurrently with processing the next screen region, the Region Accumulator sends the previous region's data to the Reconstruction Buffer. The Reconstruction Buffer computes the final pixel values for the region and forwards them to the Frame Buffer.
- After all regions have been processed and the final pixel values calculated and forwarded to the Frame Buffer, the Frame Buffer swaps buffers.

The system can function in *retained* mode, in which there is a fixed set of reference images describing an environment, or *immediate* mode, in which new reference images are being received "on the fly".

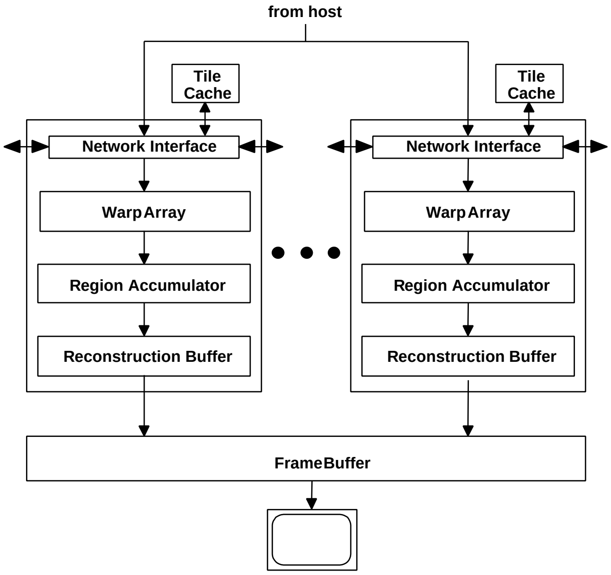


Figure 1.8. Block-diagram of the WarpEngine

1.3.2.3 Host and Software

The host is responsible for determining which reference-image tiles will be used to compute the current destination image, for sorting the tiles according to screen region, and for sending the tiles to the WarpEngine Nodes. The host must also determine the interpolation factor for each chosen tile and send instructions to control the warping and interpolation, but these instructions are cached in the Tile Caches and should not represent a significant computational or bandwidth burden for the host.

Retained Mode

Since all reference images are available beforehand, and since the depth discontinuities in the reference images do not depend on the desired view, surface connectivity is estimated as a pre-process. This frees the Warp Array of an additional task at the price of a few additional connectivity bits per reference-image sample.

Determining which tiles are needed for the current image is a laborious process and is described in subsection 1.3.3. One also must determine the interpolation factor for each chosen tile. The ideal interpolation factor is the minimum value for which surface continuity is preserved. Chapter 4 introduces a method for computing the interpolation factors that *guarantees* that all warpbuffer locations covered by a quad have at least one sample. The method implies per-sample computation and is consequently too expensive to executed on the host. We have two solutions to the problem:

1. Compute approximate interpolation factors using per tile information which can be computed as a preprocess since in retained mode the reference images are available beforehand.
2. Let the WarpEngine accurately establish the interpolation factors.

For the first solution, the per-tile maximum-changes in disparity along each direction are used to estimate the maximum screen-space distance between two neighboring samples. The maximum one-pixel disparity variation is computed as a pre-process, taking into consideration depth discontinuities.

The second solution is attractive because it is accurate and because the WarpEngine must compute the information needed anyway for warping purposes. In the warp and then interpolate scenario, after all the samples are warped, the interpolation factors can be easily computed by the SIMD array with a max function that uses the neighboring PEs' intercommunication paths. The only difficulty is that the appropriate SIMD interpolation code is known only after the interpolation factors are established. The solution is two-stage-pipelined processing of tiles: warp and compute interpolation factors for tile *t* while waiting for the interpolation code for tile *t-1* and then interpolate between warped samples of tile *t-1*.

In retained mode, frame-to-frame coherence can be exploited to minimize bandwidth requirements by storing each rendered tile in the Tile Cache of the WarpEngine Node that rendered it[7]. A large percentage of these tiles can then be used in rendering the same region for the next frame, and many of the remainder can be re-distributed using the Network Interface and used by other WarpEngine Nodes for other regions. Only a relatively small percentage of the tiles will need to be sent from the host; in fact, with a

[7] Using a 256-Mb SDRAM chip as the Tile Cache, each WarpEngine node can cache up to 16K reference-image tiles (each tile contains, 256 pixels, each with 4 bytes of color and connectivity, and 4 bytes of disparity).

modest number of reference images, it should be possible to cache all the reference-image tiles. A PC's AGP interface should provide plenty of bandwidth for sending missing tiles and pointers to cached tiles[8].

Immediate Mode

For immediate-mode, frame-to-frame coherence cannot be utilized as effectively, since users may wander into areas of the environment that have not been previously sampled in reference images, and the environment itself may indeed be in flux (persons moving, for example). This means that bandwidth requirements from the host will be much higher. In the worst case, it may be necessary on each frame to send every tile from the host to the WarpEngine, and to render every tile.

Within the next few years, we do not expect real-time depth-image acquisition at better than VGA resolution. We can build an immediate-mode system with 20 WarpEngine Nodes that contains a full-screen-sized warp buffer; this means that bucket sorting is not required. Similarly, the low-resolution yields a manageable amount of data. If one data stream provides 640*480 pixels at 30Hz, this is 36,000 tiles/sec or 72 megabytes/sec. For an immediate mode system with four such data streams, a single high-end PC host with an AGP 4X Interface could handle routing tiles to the WarpEngine Nodes. Silicon technology (for the WarpEngine ASIC) and interface technology (for data bandwidth) should scale as depth-image acquisition scales. We are also investigating the possibility of decompressing tiles within the Warp Array.

Another difference is that the PEs will have to compute connectivity information. This is not a serious performance loss since the computation required is simple enough: two adds and a compare for each of the four directions along which connectivity is estimated; a PE can easily get the disparities of the neighboring samples through the closest-neighbor communication paths. Also the host cannot approximate the interpolation factors and they will have to be estimated on the WarpEngine, as described above.

1.3.2.4 Performance Considerations

The performance estimates are based on our WarpEngine functional-block-level software simulator. The Warp Array performance was measured with a cycle-accurate simulator. The Warp Engine system has two basic performance limits: the number of tiles per second that can be warped and interpolated, and the number of regions per second for which final pixel values can be reconstructed and forwarded to the Frame Buffer. The first defines the maximum achievable rendering rate, while the second defines the maximum achievable update rate for a given screen-size.

Tile Warping/Interpolation Performance

We have found that the Warp Array will require 1878 cycles to perform a 3D image warp for all samples in a tile, using fixed-point arithmetic as described in [Mark99].

[8] AGP 2X presently supports peak data transfer rates of 533 MBytes/sec, with a future 4X extension to 1066 MBytes/sec planned. Actual usable throughputs are 50-80% of the peak rate.

If interpolation is done after warping, the interpolated samples will be computed and output to the Region Accumulator one sample at a time (over the entire tile). Outputting one sample for the entire tile requires 256 clock cycles (one cycle per PE). The time to actually compute each interpolated sample from the warped samples will be significantly less. We found that, on average, interpolation generates about 8 sub-samples, so about 8 * 256 = 2K cycles are required to interpolate and output the warped samples.

The Region Accumulator can process up to two samples per clock cycle, assuming decent load-balancing, so it is very likely that the one sample per cycle peak output rate of the Warp Array can be sustained. The total time per tile is therefore about 4K cycles, or about 75K tiles per second, per Node. In our simulations a typical overlap factor is less than 1.5, so the net performance will be 50K tiles per second per Node. Again, according to our simulations, for VGA output resolutions 5K tiles are typically required to render a scene; these numbers extrapolate almost linearly to higher resolutions. Using these assumptions, we computed the following performance numbers for some typical system configurations (see **Table 1.1**).

Screen size	**Tiles per frame**	**Nodes**	**Subsamples per sec.**	**Update rate**
640x480	5K	3	307 M	30
1280x1024	20K	16	1.6 G	40
2048x1024	32K	32	3.2 G	50

Table 1.1 WarpEngine projected performance for various system configurations.

These numbers show that 4 Nodes can easily handle VGA output resolution loads and that 32 Nodes make a quite powerful system capable of high update rates at HDTV resolution.

If interpolation is done before warping (which we do not think is necessary), it takes on average 8200 cycles to interpolate and warp the same average number of 8 sub-samples. However, there is enough time for the warped sub-samples to be forwarded to the Region Accumulator so no additional cycles are needed. This indicates that interpolation and then warping is feasible but it requires on average twice as many Nodes for the same performance.

Reconstruction Performance

The Reconstruction Buffer operates on the back buffer. It requires 64K clock cycles to compute final pixel values for a region, which is pipelined with the time to render another region. Only if the next region is assigned fewer than 16 tiles (less than it takes to *cover* the region) will the reconstruction time affect performance.

1.3.3 Finding the depth-and-color samples necessary for the current view

Since the reference images are subdivided into tiles, finding the *samples* for the current view is equivalent to determining which *tiles* are needed. Choosing the tiles for the current image begins with finding the tiles that are visible. This is done efficiently by subdividing each reference image down to 16x16 tiles in quad-tree fashion and recursively testing whether rectangular sub-images of the image are visible. The visibility test itself is identical to the bucket sorting of tiles: the 4 corners of the sub-image are warped with the minimum and then the maximum disparity of the sub-image. The bounding box of the 8 resulting points is a conservative estimate of the screen area covered by the sub-image. If the sub-image is a tile (a leaf in the quad-tree) it is also assigned to the appropriate screen region bin(s).

Depending on the scene, a large number of tiles can be visible and warping all of them is inefficient. Not all visible tiles are needed for the current frame since some tiles sample the same surfaces. Choosing among the visible tiles is not a trivial task. First one must determine which tiles sample the same surfaces and then choose among the several candidate tiles according to a quality metric.

The algorithm I use approximates each visible tile by two triangles. The triangles are transformed, projected and scan-converted according to the desired view. The z-buffer test is fuzzy and when two samples are close, the one that belongs to a better tile wins. A better tile is a tile whose approximating triangles have a desired-image size closer to 16x16, which implies a reference-image sampling close to the destination-image sampling. After all visible tiles are processed, the chosen tiles are the tiles that have at least one sample left in the tile-choosing buffer.

For tiles that have depth discontinuities the triangle approximation breaks. Such tiles are segmented in depth-discontinuity-free subsets, whenever possible. The segments are then processed like regular tiles. Another important problem with the tile choosing algorithm presented above is that it does not guarantee that it selects samples for *all* visible surfaces. To address this problem I developed the *vacuum buffer* tile-selection method. Given a set of tiles and a desired view, the vacuum buffer estimates conservatively whether surfaces visible in the desired view could have been missed by the set of tiles. The vacuum buffer is essentially a generalized z-buffer that records the sub-volumes of the current view-frustum that have not yet been determined by the set of tiles considered so far. The method is described in detail in Chapter 6.

Chapter 2 Background

2.1 Why Image-Based Rendering?

The last twenty-five years have brought remarkable progress in the field of 3D computer graphics. The rapid pace at which the field evolved should be attributed to the very nature of the problem addressed. The process of converting the model of a 3D scene into desired images is a task well suited for a computer. The machine is shielded from the incredible complexity of the real world, and although it does not produce *perfect* results, it produces *some* results (see **Figure 2.1**).

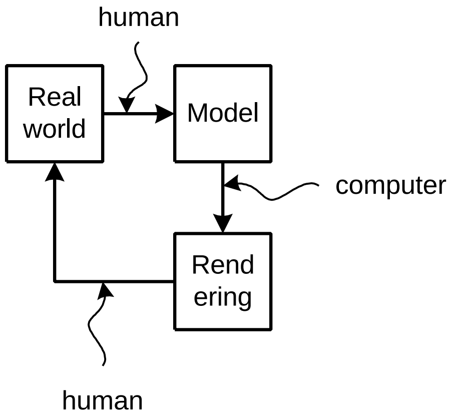


Figure 2.1 The problem of 3D graphics. Human contributions are needed in order to create the model. Rendering from the model is automatic, but, since the results are not perfect, one relies on the user to convert them (mentally) into snapshots of the real world.

Although the rendered images only approximately represent the scene, the earlier results were incrementally improved at a rapid pace. As a comparison, improvement was more difficult in the case of computer vision applications that by definition have to deal with the complexity of the real world.

At the beginning, the obvious bottleneck in interactive 3D graphics was the performance of the rendering hardware. The scene model consisted of a collection of polygons (usually triangles) that represented surfaces, of a list of light sources and of a list of material descriptions. The quality of the images was affected by the limited number of polygons that could be rendered at each frame, by the coarse descriptions of lights and materials, and by the simplicity of the expressions that approximated the results of the light / matter interaction.

Increasing the number of triangles that can be processed at each frame increases the quality of the rendered images by allowing an increase in the complexity of the scene model. After a triangle is transformed and projected to the desired image plane, the surface it represents is rendered by setting all the pixels that are inside the frame of the triangle to the color of the surface. The cost of this *rasterization* operation is proportional to the area of the projected triangle, and was the bottleneck in early systems. The fill-rate, defined as the number of pixels that can be set per second, was improved using parallelism. Ingenious architectures like the one based on Scan Line Access Memories [Demetrescu85] and Pixel-Planes [Fuchs89] allocate many processing units to each projected triangle and several projected triangles are processed in parallel.

The transformation and projection cost is proportional to the number of triangles and, as more and more triangles could be rasterized each second, the cost increased accordingly. On the other hand, complex models are composed of smaller triangles in order to capture minute detail. This implies triangles of smaller screen size, which are easier to rasterize. Consequently the per-triangle (per-vertex) computation replaces the old per-pixel computation bottleneck. Graphics-optimized floating-point units addressed this problem [Clark82].

Fast transformation[9] and fast rasterization allowed for a dramatic increase in the number of triangles processed per frame. Examples of such powerful graphics architectures include (but are not limited to): SGI's RealityEngine [Akeley93], SGI's InfiniteReality [Montrym97], UNC's PixelFlow ([Molnar92], [Eyles97]), 3DFX's Vodoo, NVIDIA's GeForce. Simultaneously, lights and surface materials were described using more and more parameters that were used in more and more complicated lighting equations. The result was a substantial increase in the quality of the rendered images. However, the images were still easily distinguishable from photographs, and continued to require the viewer's contribution in order to be associated to snapshots of a real scene. The problems that needed to be solved this time were fundamental limitations of the classic modeling and interactive rendering techniques.

First, the human effort required to produce the complex geometric models of 3D scenes is too costly. Second, the rendering quality will probably not be entirely satisfactory without considering more than just the reflection of a light ray off the *first* object encountered. When object inter-reflections are considered, complex scenes composed of numerous surfaces imply very high or prohibitive rendering costs. The need of automating the modeling process while trying to lower the cost of realistic rendering prompted 3D graphics researchers to try using images. There are numerous rendering techniques that use images; the most significant ones are reviewed in the next section.

2.2 Image-Based Techniques

I will categorize the image-based rendering methods as complete methods and incomplete methods. The incomplete methods either do not rely exclusively on images or they do not produce *correct* 3D views from *arbitrary* locations. The term incomplete is by not pejorative and the methods in this class

[9] The term is used here generically implying all per-vertex computations

offer good, practical solutions to specific applications. They also have contributed to the development of complete techniques.

2.2.1 Incomplete image-based rendering methods

2.2.1.1 Texture mapping

Texture-mapping [Blinn76] was the first use of images in 3D computer graphics. Instead of specifying the color of a triangle by specifying the color of its vertices, the color is specified by an image (texture map). Hard to model materials such as cloth, marble and skin can be captured effortlessly using a camera. Triangles do not need to model flat-surface color detail anymore and are used solely to model the 3D appearance of the objects in the scene. At rendering time, the color of a pixel interior to the polygon is looked up in the texture-map. This is relatively inexpensive and certainly less expensive than achieving the same result by rendering from triangles. For example text can be pasted onto objects using an image of the text that was rendered offline. Another example is pre-computing radiosity solutions as texture-maps, which can be inexpensively reused (assuming that lighting conditions are constant) [Bastos97].

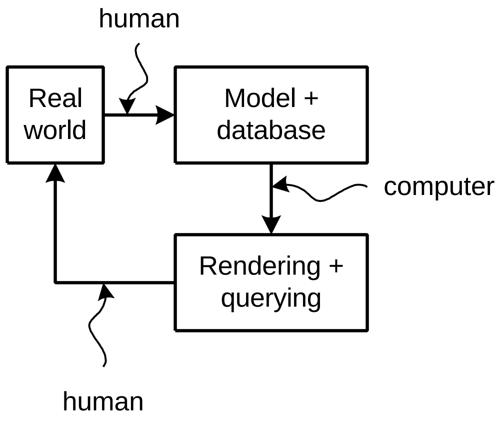


Figure 2.2 Rendering using both a geometric model and images (database of color samples). The samples alleviate the modeling bottleneck and increase the realism of renderings. At rendering time the sample database is queried for inexpensive rendering of surface detail.

Texture mapping introduces the idea of supporting rendering with a database of color samples that is queried at render time (**Figure 2.2**). One can request a sample along a ray that is not included in the database, case in which the approximate answer is composed of neighboring samples by interpolation. Texture mapping achieves for the first time a reuse of some of the rendering effort / quality of previously rendered (acquired) images. The success of texture mapping shows that for some surfaces *recording and reusing* works better than pure *synthesis*. An analogy I heard from Paul Debevec: the electronic pianos that play back recorded sounds produced by mechanical pianos are always preferred to the pianos that synthesize the sounds.

2.2.1.2 More textures, coarser geometry

Although the modeling task is made easier by texture mapping, a detailed geometric model of the scene is still required. A way of reducing the modeling effort even more was demonstrated by the *Façade* system [Debevec96]. Façade is designed for outdoor architectural scenes. First high-resolution photographs of every façade of every building in the scene are acquired. Then the system detects vertical and horizontal edges in the photographs and assists the modeler in creating a crude geometric model of the scene consisting mainly of cuboids, prisms and pyramids for the buildings. The modeler sets the texture-map coordinates for the polygon vertices using a graphical interface, which considerably expedites the completion of this otherwise arduous task. The real façade details are of course not entirely flat[10], but approximating them with a flat texture-map provides a convincing visualization of a large outdoor scene.

2.2.1.3 Images only: *panoramas*

One of the better known early systems that drastically reduced the manual modeling effort was *Quicktime VR* [Chen95]. 360 degree panoramas are constructed from overlapping photographs taken with a camera that rotates around a center point. Stitching software automatically creates a full spherical or cylindrical panorama. Using a simple software renderer running on a personal computer, a user can interactively explore natural scenes. The system was simple enough to become a commercial product of some success. No viewpoint translation is possible. Several panorama points were connected with video corridors that maintained the continuity of the exploration experience. When, for example, the user decides to go in another museum room, he loses control over the view direction and a video sequence is played back showing the transition from the panorama center in the current room to the panorama center in the next room. Forward translation is effectively simulated by zooming in; some panoramas have several resolution levels and the quality of the image does not deteriorate. Low modeling effort and photorealistic renderings compensate for the lack of proper translation.

2.2.1.4 Approximate 3D: 2.xD

More freedom of movement is achieved by *view morphing* [Seitz96]. Given a pair of photographs, the user is free to move on the baseline between the two viewing points. The new intermediate images are created using the camera parameters of the two views and a set of corresponding points in the two images. This transformation is *shape preserving*[11] and does not require the pair of reference images to have parallel image planes. The method produces convincing new images from given images, although they are just an approximation of the correct 3D views. Key in achieving good results is the set of corresponding points.

[10] 3D detail is occasionally added using view-morphing

[11] A solid object appears to move rigidly (does not stretch or compress) in the sequence of intermediate images.

Since the camera parameters of the stereo pair are known, the corresponding points are equivalent to 3D information (one could infer the 3D location of the samples identified as the same in the two images).

Clearly, a complete 3D rendering method, capable of producing arbitrary *novel*[12] views cannot rely exclusively on images and their camera parameters. Correct novel views are equivalent to knowing where the samples lie in 3D space. The proof of this claim is trivial: if one knows where the samples are located in 3D space, then one will know where to project them in the desired image plane, creating novel views. If on the other hand one knows where a sample projects in the desired view, one can infer its 3D location from its position in the original image and the camera parameters of the two images.

However, finding the 3D location of the samples of an image is a non-trivial problem. It has been a central problem of computer vision for many decades. In the beginning of the chapter, I pointed out that initially 3D computer graphics progressed with large strides since it did not deal with the complexity of the real world as input, and a simplifying model was used. As building the models became a difficult task, in order to further improve the quality of the rendered images, computer graphics had to tackle the full complexity of the real world.

Because of the difficulty of inferring 3D structure from images, researchers continued to develop approximate methods. A popular idea is to use *sprites* ([Lengyel97], [Shade98]) in order to produce some limited motion parallax between objects (or group of objects) that are at different depth. The scene is modeled as a collection of billboards that move independently as the user wanders between them. Since the depth variation on the objects is limited and certainly considerably less important than the depth variation between different objects the approximation often produces satisfactory results. Modeling with sprites is relatively simple since objects can be extracted from the images using 2D operators. The more challenging part is locating and orienting the supporting planes for the billboards, but since typically there are only a few of them, it can be done manually.

2.2.1.5 IBR to accelerate geometry-based rendering

Image-based rendering ideas have also been used to accelerate conventional geometry-based rendering. The *Talisman* architecture [Torborg96] is designed around the idea of achieving high rendering performance with commodity hardware by making the best possible load-lowering approximations. The individually animated objects are rendered into separate image layers (similar to sprites) that are composited to create the final image. Spatial and temporal coherence are exploited and instead of re-rendering the objects at every frame, the new image is approximated by applying a 2D (affine) transformation to the previous full rendering of the object. Compression is broadly used to reduce required bandwidth and image storage capacity. Although the hardware was never built, Talisman introduced a novel approach to designing 3D rendering hardware.

[12] There are rendering systems that rely exclusively on images (section 2.2.2.1). However they do not produce novel views, in the sense that the rays for all the views ever generated already existed.

The idea of reusing rendering effort was used in [Schaufler96] where several images were maintained and reused from a three-dimensional cache. A slightly different idea is to simplify the geometric model before rendering as a pre-process. Replacing the subset of the geometric model corresponding to a distant object with an *impostor* [Maciel95], which is a rendered image of the object, reduces the number of polygons that have to be rendered at each frame. The images are only approximations of the object replaced and finding where to place the images is a difficult problem. Note that whether an object is rendered as an image or whether it is rendered from scratch depends on the current view position. This implies possible artifacts when one switches from the image representation to the geometry representation.

Good results are obtained in the more constrained case of architectural scenes. The portals (doors, windows) that interconnect the cells (rooms) of such a scene are a natural place for the impostors ([Aliaga97], [Rafferty98], [Popescu98]). The geometric model of the adjacent rooms is replaced with images, dramatically reducing the per frame polygon count, which is reduced to the model of the current cell. As the viewer moves from one room to the next, the images are replaced with the geometric model. Since the objects visible in the adjacent room are typically not close to the door, the transition from images to geometry is smooth as no nearby object is rendered from images (a case in which the rendering quality would be insufficient). Rendering the images as a pre-process has the advantage of a virtually unlimited rendering-time budget. If storage is a problem, the images can be rendered by a lower priority process that tries to anticipate what images are needed, which is a relatively simple problem in the portals and cells world.

Recently another effort was made to try to generalize the use of impostors. The geometric model of an arbitrary 3D scene is pre-processed and sets of geometric primitives are replaced with images such that the total number of primitives remains under a pre-specified maximum [Aliaga99]. What parts of the geometric model are replaced with images depends on the current view. Assuming that the rendering from images (use of the impostors) is of constant cost, the system guarantees a sustained frame-rate since the number of primitives is bounded at each frame. Since a view frustum that is enclosed by another view frustum is guaranteed to contain less geometry, it is sufficient to bound the primitive count for a finite subset of viewing locations. The pre-processing algorithm is automatic and starts by subdividing the viewing volume according to a regular grid. The solution is computed for the intersection points of the grid, for all viewing directions. In some cases no valid solution can be found and the grid must be locally refined. At the end all possible viewing locations are assigned a grid node, which limits the number of primitives that have to be rendered at run-time.

Post-rendering warping ([Mark97], [Mark99]) relies on path prediction techniques and rendering from images in order to increase the rendering frame-rate. The scene is rendered for the estimated future desired view. Another view, close to a past location of the user is also rendered. The two images are used to compute intermediate views by warping. A rendering capability of 4 fps for example can be enhanced to 12 fps if warping can be done 10 times a second. The two reference frames used to warp the intermediate images exhibit a large amount of coherence, which suggests that warping can be done efficiently and accurately. Image warping will be described in detail later in this chapter.

The next section presents two IBR methods that are complete, in the sense that they produce arbitrary views of a 3D scene, relying only on images.

2.2.2 Complete image-based rendering methods

2.2.2.1 Ray-database methods

The *Lumigraph* [Gortler96] and *Lightfield* [Levoy96] methods were published simultaneously and are remarkably similar. The central idea is to acquire all the color samples one might ever need while rendering a 3D scene. Since the camera parameters (intrinsic and extrinsic) also must be known in order to know how to use the color samples, the database is in fact a ray database.

The problem of rendering has been formulated as evaluating the *plenoptic* function [Adelson91], which associates a color value to each point of the space of all possible rays. The rays are half-lines and can be specified by five parameters (three for the cartesian coordinates of the origin and two for the angles specifying the direction). In the case of a dynamic scene a sixth parameter is needed to specify the time. The lumigraph-lightfield are explicit definitions of the plenoptic function.

Is the number of samples needed to model a 3D scene bounded? If the output resolution is finite and if the viewer cannot get infinitely close to an object, the total number of samples required is finite while ensuring an appropriate sample density for any desired view.

The next question is how many samples are needed. The Lumigraph and Lightfield both consider static scenes and simplify the 5-dimensional function with a 4-dimensional parameterization. The rays are defined by pairs of points on two planes. The points are specified with two plane-coordinates each. The rays originate from one of the planes, the back plane, and go through the front plane[13]. The two planes are discretized by grids, and a ray is defined for each pair of two grid points, one from the back- and one from the front-plane.

None of the 5 dimensions of the plenoptic function are redundant of course. Dropping one dimension is possible only by clamping the origins of all the rays to a plane, namely the back plane. While still providing the full range of ray orientations, the 4-parameter ray description does not work unless the space between the two sampling planes is free of occluders.

Even if there are no occluders, the 4-parameter definition of rays has implications on the quality of the rendering. At rendering time, when establishing the color for a certain pixel of the desired image, one is interested in the sample returned by a ray that passes through the center of the pixel. Due to the discrete nature of the database, it is unlikely that the precisely specified ray can be found. Interpolating between existing rays is difficult since there is no bundle of rays originating from the desired viewpoint, thus it is

[13] In this discussion, rays are oriented in the opposite direction of the light, like in ray-tracing.

impossible to determine which rays to interpolate. This is precisely the price to pay for defining the rays as lines and not as origin plus orientation, and it translates into image blurring. The authors of the Lumigraph use scene-depth information in order to improve the search for the appropriate ray, but of course, this re-introduces the difficult problem of depth extraction.

The Lumigraph - Lightfield methods are very important since they take the idea of IBR to the extreme. The model is completely replaced with a database, and the rendering is merely a querying of the ray-database (see **Figure 2.3**). However, the size of the database (for interesting scenes and viewing volumes) implies storage capacities and bandwidth beyond what current technology offers. In the original papers, the researchers have found that for modest (sub-VGA) output resolutions the database for a simple scene composed of a single object requires megabytes of storage after modest compression, while the viewer is restricted to a small viewing volume.

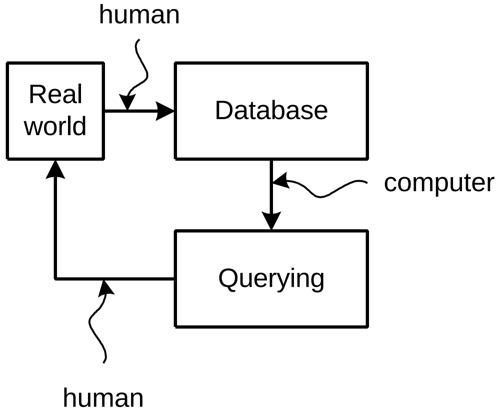


Figure 2.3. Rendering as database querying. The database of rays explicitly defines the plenoptic function. The human intervention at the modeling stage is greatly reduced and is limited to operating scene acquisition devices (calibrated cameras).

2.2.2.2 Image-based rendering by warping

Rendering methods can be classified on a continuum, according to how they define the plenoptic function, implicitly or explicitly (see **Figure 2.4**). Rendering from a geometric model is equivalent to specifying the plenoptic function as a collection of implicit functions to be used for continuous sub-domains. For example Gouraud-shading a triangle is equivalent to a function that associates a color sample to rays. The expression of the function uses parameters derived from the coordinates of the vertices, surface properties and light descriptions.

Texture mapping makes the first step towards the *explicit* end of the continuum: some rays are specified explicitly. The Façade system uses coarser geometry and more textures thus it is another step. The lumigraph-lightfield approaches are exclusively explicit descriptions of the plenoptic function.

A conclusion reached earlier in this chapter is that in order to build a *new* view from existing views of the scene one must know the 3D model-space location of the color samples. The lumigraph-lightfield approaches do not contradict the conclusion since all the needed rays are already available in the database, so the views are not *new*.

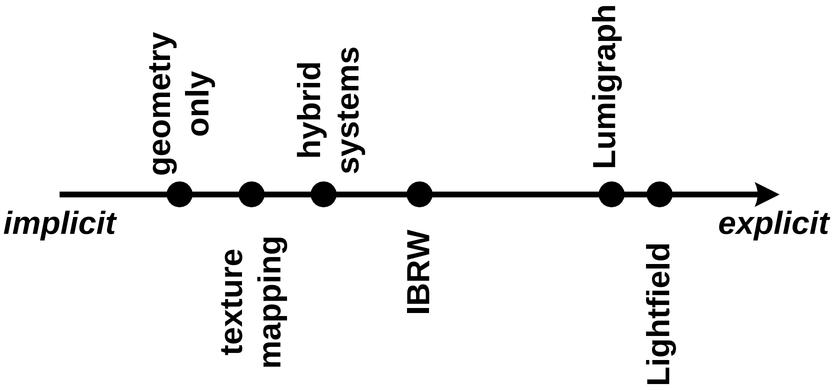


Figure 2.4. Classification of rendering methods according to the scene representation used, from implicit to explicit.

Depth images and the 3D warping equations

McMillan and Bishop describe an IBR method that uses images enhanced with per-pixel depth [McMillan95]. Since the camera parameters used to acquire the images are also known, the depth effectively records the position of the color sample in 3D space. The depth images are warped in order to create new 3D-correct views. The warping was described in section 1.1.2.1. The new location of a sample (in the desired view) is computed using the 3D-warping equations, which are a condensed formulation of the classic 3D transformations. The 3D-warping equations take the original-image sample coordinates and depth as input and provide the desired image coordinates (see **Figure 2.5**).

Compared to the ray-database methods, image-based rendering by warping (IBRW) reduces the number of samples that must be stored in the database since a sample can be reused to create novel views using the depth information. The depth information is analogous to a model since a continuous interval of new rays can be generated from the original ray. On the continuum used to classify IBR methods, IBRW is more implicit than the lumigraph-lightfield methods (see **Figure 2.4**).

The fundamental assumption on which 3D warping relies is that a surface looks the same when seen from a different view. This assumption is of course not always true. The only surfaces that have the same appearance when seen from different angles are diffuse surfaces, whose reflection model is perfectly Lambertian. Natural scenes are rich in surfaces that change appearance as the viewing angle changes. For correct rendering of non-diffuse surfaces one can still use the warping equations to determine the new location of the surface but the color sample must be modified according to the new viewing angle. The methods developed for geometry-based rendering still apply and one can use them for achieving the view-dependent effects.

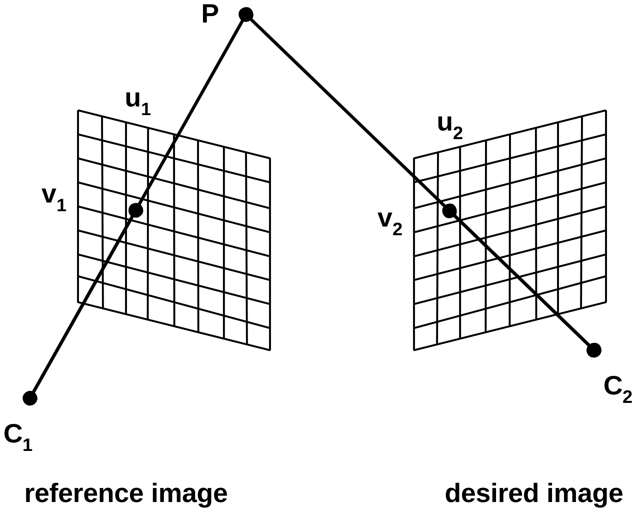


Figure 2.5. Illustration of 3D warping. The reference image sample (u_1, v_1) corresponds to the model-space point P. The warping equations re-project the sample in the desired image at location (u_2, v_2). C_1 and C_2 are the centers of projection of the reference- and desired images.

Visibility computation without depth

Besides the efficient incremental transformation given by the warping equations, McMillan and Bishop also contributed a visibility algorithm that ensures correct results without depth comparisons. The algorithm relies on an ordering of the samples that is *occlusion compatible*; in other words, if two samples re-project (warp) to the same desired image location, the sample that is closer to the viewer is warped last and overwrites the more distant sample. Moreover the ordering of the samples does not prevent incremental warping since the depth image is subdivided in 1, 2 or 4 rectangular subregions that are traversed in row- (or column-) major order.

Correct visibility without testing is possible because of the way the samples move when warped. A formal proof can be found in [McMillan96] and the argument that follows is meant to give just an intuitive explanation.

The samples can either move away from the epipole or move towards the epipole, where the epipole is the projection of the center of projection of the desired image onto the reference-image plane. Intuitively, the samples flow away from the epipole when the viewer starts at the reference location and then moves forward (see **Figure 2.6**). If the user moves backwards, the samples agglomerate at the epipole. The samples always flow on segments that originate at the epipole (epipolar segments).

Let's consider the case of samples flowing away. If two samples land at the same location, the sample that was closer to the epipole traveled more since it had to catch up with the sample that was farther from the epipole. The sample that travels more is always closer to the viewer, which is an intuition we have from observing the movement of objects in our field of view when we walk / travel. Consequently the sample that traveled more should remain visible thus it should be warped last (in order to overwrite the other sample and not to get overwritten).

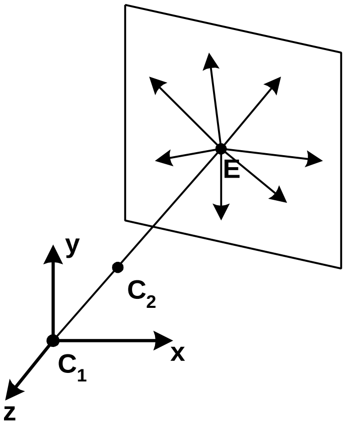


Figure 2.6. Illustration of epipolar geometry. C_1 is the center of projection of the reference image. As the view translates forward (to C_2), the samples in the image move away from the epipole E on radial epipolar segments.

An important observation is that not any arbitrary two samples can land at the same location. Samples flow along epipolar lines, so if two samples are to occlude each other they must belong to the same epipolar line. Thus visibility infers a partial ordering of the samples of the image that is defined only for samples on the same epipolar line.

Hence, in order to establish an occlusion-compatible ordering, it is sufficient to enforce the correct traversal order along the epipolar lines (towards or away from the epipole). An advantageous occlusion-compatible ordering can be obtained by dividing the reference image along the horizontal and vertical epipolar lines, resulting in 4, 2 or 1 rectangular sheets according to where the epipole is located with respect to the frame of the image (**Figure 2.7**).

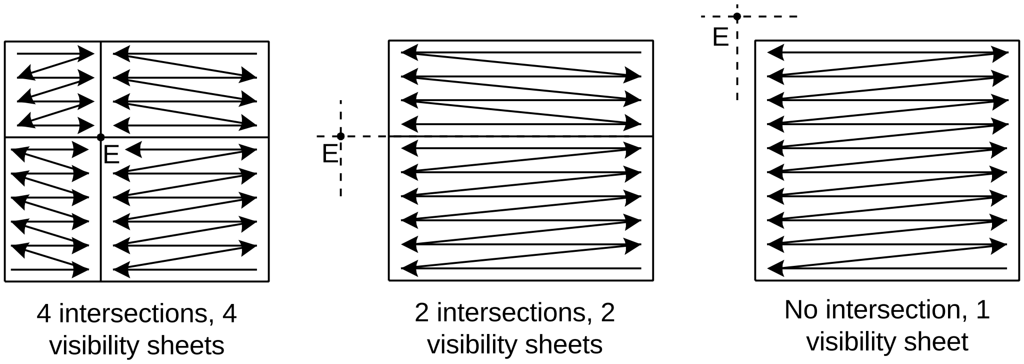

E
4 intersections, 4 visibility sheets
E
2 intersections, 2 visibility sheets
E
No intersection, 1 visibility sheet

Figure 2.7. Occlusion compatible ordering using epipolar geometry. The number of visibility sheets depends on the number of intersections between the horizontal and vertical epipolar lines and the frame of the reference image. The case shown corresponds to the outward moving samples; for correct visibility, the samples farthest from the epipole have to be warped first. The arrows show the occlusion compatible traversal of the visibility sheets.

Depth acquisition devices

Another reason for the appeal of IBRW are the recent important advances in rangefinding technology. Rangefinders ([K2T], [Beraldin92], [Cyra], [Nyland99]) and other range-acquisition equipment ([Kanade99], [Minolta]) are rapidly improving and appearing on the commercial market. Some instruments can even acquire range in real time. Such a "depth camera" will enable extraordinary applications: a spectator of a live event will not be confined to the view of the TV camera; he or she can choose any seat in the arena, and even venture onto the stage or court.

Before IBRW is used in systems able to realistically render natural scenes at interactive rates, several problems have to be solved. One of them is the reconstruction of the final image; the central contribution of the present dissertation is a new reconstruction technique, thus the problem of reconstruction for IBRW is analyzed in detail in the next chapter, along with a presentation of the previous solutions.

Chapter 3 Reconstruction for IBRW

The warping equation allows one to compute the desired image location of a depth-and-color sample. Unfortunately warping the samples to their new location is not enough to obtain a high-quality rendering from a different view.

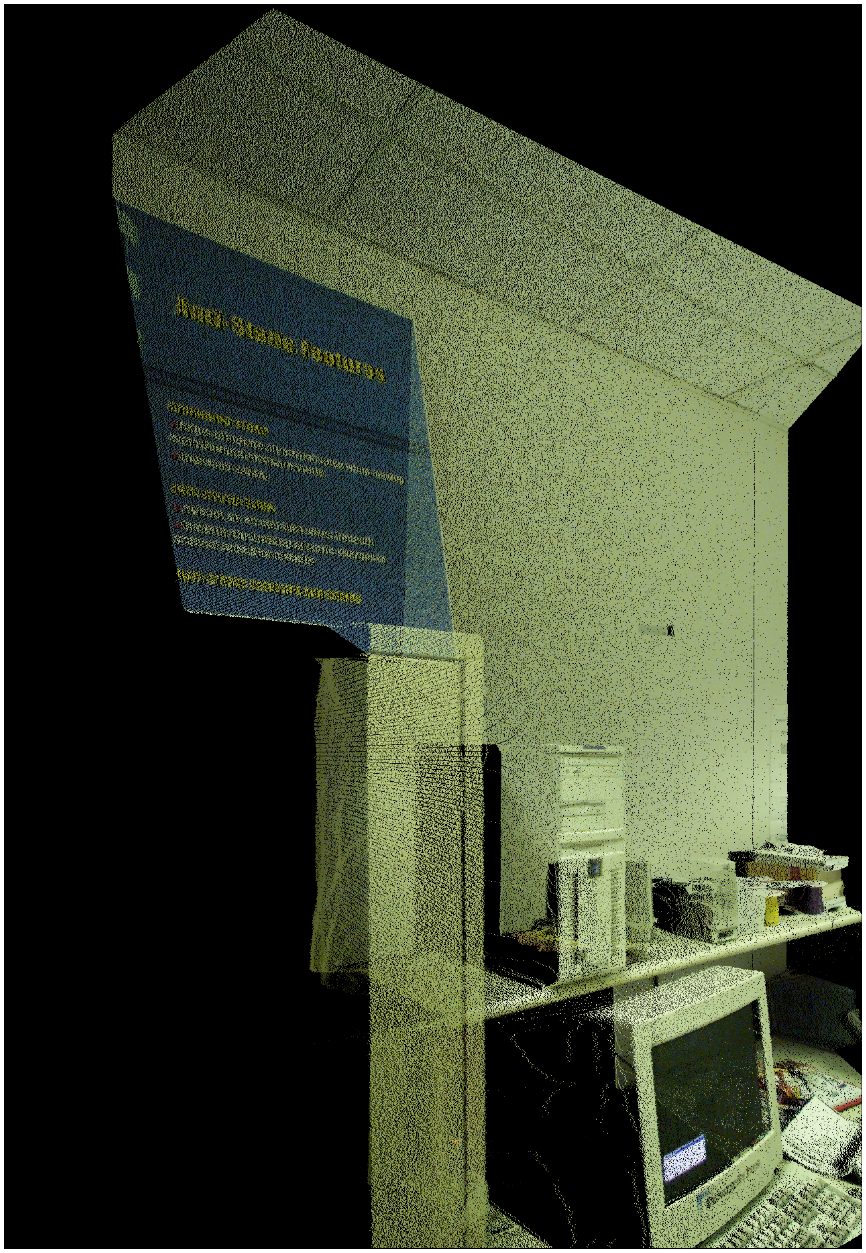


Figure 3.1. Desired view obtained by warping the samples of a depth image.

Just dropping the samples at their new locations does not produce acceptable results as can be seen in **Figure 3.1**. There are several types of error:

- when warped, the samples may move apart and unfilled background pixels or samples of a hidden surface may incorrectly show through; for example the front face of the cabinet appears to be semi-transparent and we can see the shelf and wall through it;
- samples of the same visible surface can land at the same location and simply keeping just one produces aliasing; the correct solution is a blend of the visible samples;
- some surfaces, visible in the new view, are not sampled by the reference image, thus they are missing in the rendered image. Such examples are the side faces of the computer case and monitor.

While the first two problems - incorrect visibility and aliasing - are clearly rendering problems imputable to improper use of the available surface samples, the missing samples are a modeling error. This chapter concentrates on the problem of rendering from images with depth, describing the previous solutions. Chapter 6 addresses the sample-finding problem, which implies finding good samples for all surfaces visible in the desired view.

One quickly recognizes that the reconstruction problems described above can be avoided if the forward mapping of samples is replaced with an inverse mapping. If one could easily compute the reference-image coordinates of a sample that warps to a given location in the desired image, one could traverse the desired image and query for an adequate sample for every single location. This would ensure surface continuity (correct visibility) and permit filtering to avoid aliasing in a fashion similar to the classic texture-mapping operation: a pixel interior to a textured polygon is indexed from a texture-map instead of forward mapping the samples of the texture (texels). The texture-map coordinates of the pixel are computed from the texture-map coordinates of the vertices of the polygon. Samples at fractional coordinates can be computed from neighboring samples by interpolation [Williams83], avoiding blockiness when sampling density decreases from the texture domain to the polygon domain. If the sample density increases, the width of the neighborhood of samples considered for interpolation offers control of aliasing[14].

There is no analytically computable inverse for 3D-warping. In other words, one cannot express the reference image coordinates as a function of the desired image coordinates. The term that prevents us from inverting the warping equations (see **Equation 1.1**) is the depth term.

One can find the sample of the reference image that maps to a certain location in the desired image only by searching along the reference image epipolar segment corresponding to the desired image location (see **Figure 3.2**). The search cannot be avoided as the depth of the samples in the scene cannot be expressed analytically and it is given by the samples. In my implementation, *60* steps along the epipolar segment were required on average for each desired image pixel, at a *256 x 256* resolution. Moreover, a

[14] Since convolution with large filters is expensive, the texture-maps are pre-filtered at several lower resolutions and the appropriate level of detail is readily available at rendering time (mip-mapping).

single reference image typically does not contain all the samples needed for the desired view, thus the search has to be performed in several images and the closest sample chosen. The searching procedure is described in more detail in [McMillan97].

Since the inverse mapping is not practical, alternative reconstruction solutions for IBRW had to be found.

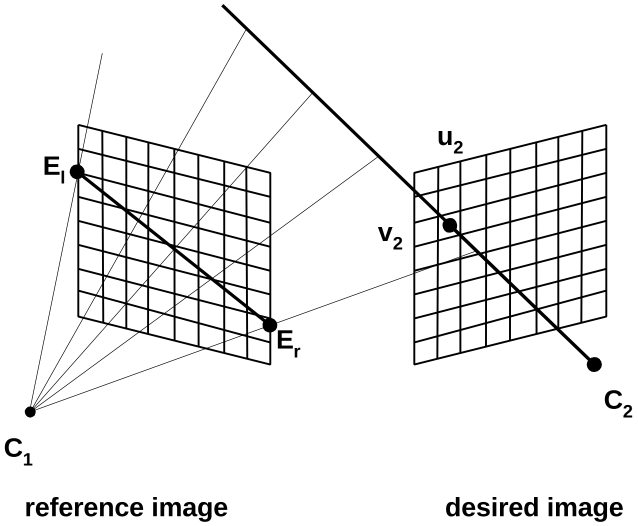


Figure 3.2. Inverse warping. The desired image location (u_2, v_2) defines a ray that projects on the reference image as the epipolar segment E_lE_r. Any of the samples along E_lE_r could map at (u_2, v_2), if they have the appropriate depth. Finding the sample is costly since it implies searching along E_lE_r.

3.1 Splatting

The first attempt to improve the reconstruction for image warping was *splatting* ([McMillan97], [Mark97], [Mark99]). The shape and size of the projected depth-and-color samples is approximated by splats, a term borrowed from volume rendering [Westover90]. Splatting was used in numerous IBRW applications ([Rafferty98], [Popescu98], [Shade98], [Aliaga99]), since it is relatively inexpensive. The quality of the reconstruction relies heavily on the precision with which the splats approximate the projected samples.

If the splats are too small, neighboring samples of the same surface will leave holes, and, again, back-surfaces or uninstantiated pixels show through. Consequently, the splats are typically approximated to excess. Splats that are too large can incorrectly erase visible samples, which produces artifacts. Also the precision of splats was limited to the size of their building blocks - final-image pixels, which prohibited the reconstruction of the final pixel value from more than one color sample.

Attempts have been made to improve the quality of splatting reconstruction by filtering the samples as they are warped [Shade98]. However, blending before the visible samples are selected and the invisible ones erased can result in artifacts due to incorrect blending of an invisible sample and a visible sample. To avoid this, the splats were chosen to be opaque at the center and then semitransparent at the edges, such that they blend in with adjacent visible samples. Computing the precise size and shape of the splat for a sample is very difficult (expensive) and it has to take into account the neighboring samples. Computing accurate splats that tile perfectly without overlap and without leaving holes is in fact equivalent to scan-converting the polygonal mesh obtained by interconnecting the neighboring samples of the reference image with polygons.

3.2 Polygon mesh

Another reconstruction technique developed for IBRW is the mesh method, which connects four neighboring samples of a depth image by two micro-triangles. The depth image becomes a 3D mesh, with the nodes defined by the depth samples. The mesh is interrupted when neighboring samples do not belong to the same surface. The mesh is transformed incrementally using the warping equations, but once the desired image coordinates of the vertices of the micro-triangles are established, polygon-rendering hardware may be used to reconstruct the final image.

The method solves the problems described above since the triangles can be rendered carefully, using antialiasing techniques. Thus the continuity of the surfaces is maintained and the sub-pixel accuracy is obtained by antialiased rendering of the warped mesh.

As stated in section 1.1.2.2, the mesh method is not a satisfactory solution to reconstruction for IBRW since it is costly and inefficient.

3.3 Other Reconstruction Methods

In finding a suitable solution for the reconstruction problem, a research path strove towards finding an approximate expression for the size and shape of the splats that is both accurate and inexpensive.

This is a problem that received attention before, when Levoy and Whitted studied the use of points as a fundamental rendering primitive [Levoy85]. They introduced the idea of computing the desired-image area of the samples using the normal of the surface. The surface normal is the dual of depth information: using the depths of neighboring samples one can compute the normal and the normals of the samples together with some boundary conditions can be used to infer per-pixel depth. This reinforces the claim that accurate splat evaluation cannot be done without considering neighbors' information. The idea of using the normals for splat evaluation was reused several times. In [McMillan97], the samples are modeled by discs of radii determined by the normals; the disks project as ellipses in the desired image. In [Shade98] the normals are sorted in bins according to their orientation. A splat is computed for each bin representative, at each frame, and is used for all the normals of the bin. Mark proposes approximating the splats by axis-aligned bounding-boxes [Mark99]. All these methods offer approximate solutions to splat computation, and the quality of the resulting images is considerably lower than in the case of the mesh-reconstruction method.

[Shade98] proposes forward-mapping the depth, using a 1x1 splat followed by hole-filling or using a 2x2 splat. The color of the desired image is evaluated by inverse-mapping, made possible by the warped depth map. The depth map acts like an item buffer that identifies the color samples needed for each output pixel. The method relies on the idea that the errors committed by the forward mapping of the depth information will be less visible than the errors of splatting the color directly to the final image. The method was presented for sprites enhanced with depth, which are a particular case of depth images. The additional assumption is that depth (recorded as out-of-plane displacements) varies smoothly for the surface modeled.

Another improvement on splatting is proposed in [Chang99]. The depth images are stored at several levels of details. At rendering time, the appropriate level of detail is used such that the projected samples cover about one pixel of the desired image. Since splats are never bigger than one pixel, the reference images must always have at least the resolution of the desired view. The level-of-detail selection is made once per reference image. This approximation is made possible by the fact that a reference image samples only surfaces that are located in a bounded volume. Since the surfaces are close to each other it is likely that a unique level of detail is a fairly good approximation. However, the orientation of the surface is not considered, which is also an important factor in determining the splat size, along with the distance from the surface to the viewer.

Grossman and Dally also use several resolution layers but for the desired image (as opposed to the reference image) in their point-rendering system [Grossman98]. The rendering is done at the appropriate resolution level of the desired image, such that no holes remain. The final image, the highest resolution level in the pyramid, has its holes filled using an algorithm similar to the pull-push algorithm presented in [Gortler96].

A different approach to reconstruction for IBRW are the *relief textures* proposed by Oliveira; the method relies on factoring the 3D warp into a pre-warp followed by a texture-mapping operation [Oliveira00]. The pre-warp corresponds to computing an intermediate image that corresponds to the original image plane but the new center of projection (see **Figure 3.3**). Since there is no translation between the pre-warped image and the desired image, the final transformation is an affine transformation, equivalent to texture-mapping.

The pre-warping operation is further decomposed in two 1D warps. The samples of the original reference image are first shifted on rows to their final column and then on columns to their final row. Surface connectivity is maintained by interpolation between samples. The texture-mapping following the pre-warping can be done using existing polygon-rendering hardware. However, the more expensive stage is the pre-warping.

McAllister [McAllister99] uses the programmable-primitives feature of *PixelFlow* ([Molnar92], [Eyles97]) in order to reconstruct the warped image. Each sample is rasterized as a solid-colored circle that has a quadratic falloff in depth. The quadratic splats can be rasterized (on PixelFlow) more efficiently than the two triangles per sample of the mesh method. The falloff in depth normally causes each pixel to receive the color of the closest sample. This is similar to the technique of rendering Voronoi regions by rasterizing a cone for each sample [Haeberli90], which [Larson98] used for reconstruction in image-based rendering. Unlike cones for Voronoi regions, the splat centers do not all have the same Z value. Rather, they have the sample's Z value at the center and increase quadratically toward the circle's edge. This allows compositing for visibility and reconstruction to both be done with a single Z composite. The results are esthetically pleasing. The only major disadvantage of the method is relying on PixelFlow, which is the only system that allows the definition and efficient rendering of the quadratic splats.

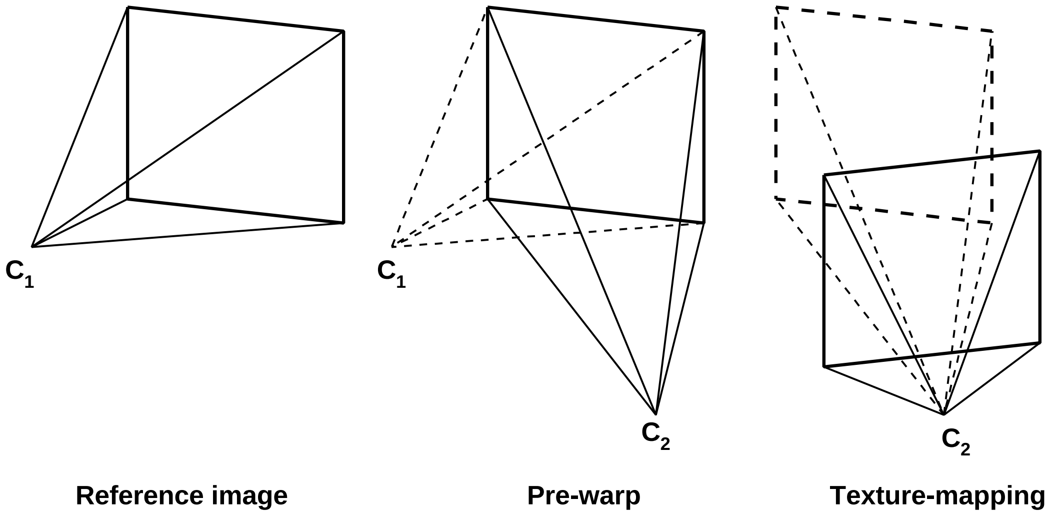


Figure 3.3. Relief texture-mapping. The reference image is first pre-warped, by keeping the same image plane and changing the center of projection from the original C_1 to the desired C_2. The pre-warped image is transformed by a texture-mapping operation in the final image.

A recent variant of the splat method is presented in QSplat [Rusinkiewicz00]. The scene (typically a densely-scanned object) is stored as a hierarchy of bounding spheres, which at rendering provides visibility-culling and level-of-detail adaptation. As with the previous splatting methods, determining the shape and size of the splat is difficult. The system guarantees no holes if the splats are circular (or square, as requested by OpenGL). However, coercing the splats to be symmetrical degrades the reconstruction. The authors report aliasing at the silhouette edges. Probably aliasing occurs at all output-image locations with high frequency, but in the case of scanned sculptures (for which QSplat was designed), high frequency is typically encountered at the silhouette edges. If the splats are elliptical, hole-free reconstructions are not guaranteed. An empirical method is used that prevents most of the holes by essentially overestimating the size of the splat.

The *surfel* method [Pfister00] is another variant of the splatting technique. Surfels are data structures containing information sufficient for reconstructing a fragment of a surface. The surfels are projected using the optimized-block-warping technique presented in [Grossman98]. The technique doesn't ensure surface continuity and one must solve the problem of *holes*, which is done in two stages, visibility and then reconstruction.

The goal of visibility is to determine the visible surfels, which are then used at reconstruction. I am glad to report that the authors were inspired to separate visibility from reconstruction from an earlier publication of some of my dissertation research [Popescu99], which they reference. In addition to the current minimum z, the z-buffer also stores, for every pixel, a pointer to the closest surfel. A technique called *visibility splatting* ensures that only pointers to visible surfels remain in the z-buffer. In other words, the occluded splats are erased from the holes. Visibility splatting implies projecting oriented model-space disks on the desired-image plane and then scan-converting their bounding parallelogram into the z-buffer. The splat evaluation is fairly laborious and moreover there is no guarantee that no holes remain.

After the visibility stage, every z-buffer location stores a pointer to a visible surfel (or corresponds to background). There are at most one surfel per z-buffer location (if two surfels project to the same z-buffer location, a choice is made according to their z at the visibility stage). But there are typically several z-buffer pixels covered by the same surfel. In other words, after the visibility stage there are still holes, except that they are not filled with surfels that are not visible. At the reconstruction stage one must fill in the holes in between visible surfels. This is done by using the appropriate mipmap level of the texture associated with the surfel. The level is computed according to the distance between consecutive visible surfels.

In conclusion, the surfel method has the advantage of correctly reconstructing the final image by blending only visible samples (surfels); however the visibility stage is still implemented by splatting and is thus prone to errors introduced by coarse splat approximation.

3.4 Conclusion

This chapter stated the reconstruction problem for IBRW and reviewed the current solutions. The splatting methods are approximate solutions and do not provide high-quality results. The mesh method produces the best results but is expensive and polygon-rendering hardware has not been optimized for sub-pixel triangles.

A closer analysis reveals that connecting the samples with triangles is, at the conceptual level, the simplest (coarsest) approximation possible that maintains C^0 continuity. In other words, assuming that the surfaces are planar between the depth samples is the approximation of lowest order possible. In establishing the rendering method presented in this dissertation, I concentrated on making the rendering of the triangles cheaper. The method presented in the next chapter achieves the same conceptual approximation of inter-sample planarity but at a much lower cost.

Chapter 4 Forward Rasterization for IBRW

It is generally accepted in the field that good results are obtained if several color samples are computed for each output pixel and they are then blended in order to obtain the final pixel value. The number of samples required is debatable, [Molnar91] indicates that good results are obtained with as little as five samples per output image pixel; nine is another frequently used number of subsamples and sixteen is too costly to be used in practice. In the case of IBRW, three sub-problems must be solved:

1. *Visibility*: eliminate all samples of surfaces not visible in the destination image.
2. *Adequate sample density*: ensure the minimum number of samples per output pixel.
3. *Reconstruction / resampling:* compute the output pixel colors.

The next subsections treat each of the three problems individually.

4.1 Interpolation in continuous parameter domain

The visibility problem is solved by maintaining C^0 continuity of surfaces, similarly to the mesh-method. The expensive per-sample rasterization of two triangles is replaced with bilinear interpolation of the parameters. The classic, backward rasterization is also an interpolation but in my forward rasterization solution no image-plane discrete variations of the parameters have to be computed. The interpolation occurs in the continuous parameter domain. There are two slightly different variants of the forward rasterization algorithm.

4.1.1 Interpolate and then warp

Four neighboring reference-image samples are bilinearly interpolated both in color and depth. The sub-samples created are then warped independently. Here is an outline of the algorithm.

```
// current sample is u0, v0; u1 and v1 define the other 3 samples
// ifu and ifv are the horizontal and vertical interpolation factors

for (i = 0; i < ifv; i++)
   for (j = 0; j < ifu; j++) {
       currentColor = Bilinear(color[u0, v0], color[u1, v0], color[u1, v1], color[u0, v1],
                                                  i/ifv, j/ifu);
       currentDisp = Bilinear(disp[u0, v0], disp[u1, v0], disp[u1, v1], disp[u0, v1], i/ifv, j/ifu);
       ($u_2$, $v_2$) = Warp(u0 + j/ifu, v0 + i/ifv, currentDisp);
       // process warped sample ($u_2$, $v_2$, currentColor)
   }
```

Pseudocode 4.1. Interpolation and then warping. Subsamples are created by interpolating the generalized disparity and color of four neighboring samples. The subsamples are then warped. The bilinear interpolation and the warping can be evaluated incrementally (not shown for improved readability).

Reference-image interpolation is a first order approximation of the surface between the four samples. In other words, if the four interpolated samples are planar, the resulting subsamples will belong to the same plane (**Figure 4.1**).

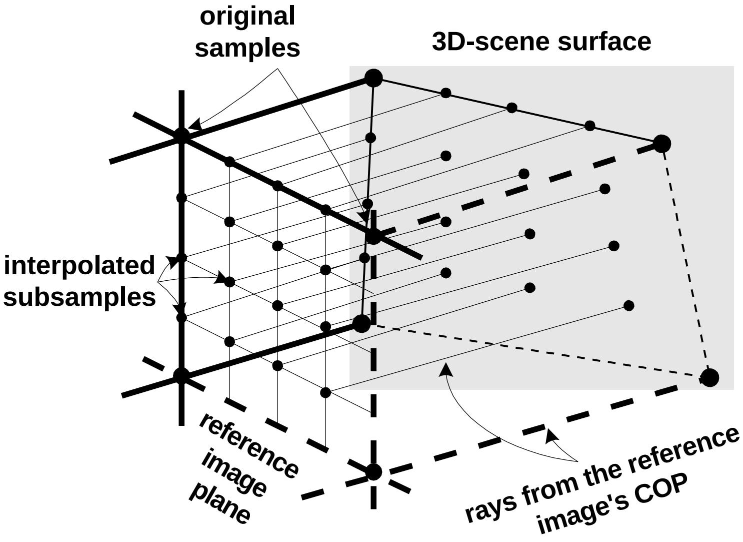


Figure 4.1. Reference image interpolation. Four reference-image samples are shown. The interpolation factor is *4x4*. In this case the scene surface is planar and all the 16 subsamples that are created by bilinear interpolation are also located on the surface. Only the left and top edges of the quad are sampled. The right and bottom edges (shown with dotted lines) are generated when the neighboring groups of four pixels are processed, and surface continuity is maintained. The subsamples are then warped to the desired-image plane (not shown).

I argued informally in section 1.3.1 that the subsamples are indeed located on the plane defined by the four coplanar samples since the generalized disparity is linear in image space. A formal proof follows.

All that must be proved is that interpolating between two samples in the reference image creates samples in the 3D world that are located on the line connecting the 3D points defined by the two samples. If this is true, all the interpolated subsamples will belong to the four-sample plane as they are obtained by interpolation between two points that are also located on that plane.

Claim. Linearly interpolating the generalized disparity of two reference-image samples generates subsamples that lie on the line defined in 3D space by the 3D points corresponding to the samples.

Proof.

In **Figure 4.2** lines through B_1, B_2, and B_3, parallel to the image plane, intersect the 3 rays in O_1, B_2', B_3' and O_3. What I must show is that B_2 belongs to the line defined by B_1B_3.

The generalized disparities for A_1, A_2 and A_3 are defined as OA_1 / OB_1, OA_2 / OB_2 and OA_3 / OB_3, respectively. Since the generalized disparity for A_2 was generated by linear interpolation (as an intermediate point on segment A_1A_3), we know that it satisfies **Equation 4.1**.

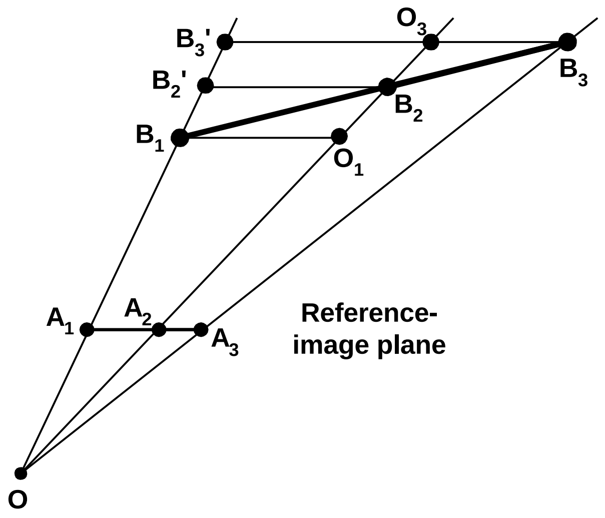


Figure 4.2. Generalized disparity linear in screen space. O is the center of projection of the reference image. A_1 and A_3 are two reference image samples. A_2 is a sample generated by interpolating between A_1 and A_3, in both reference-image coordinates and generalized disparity. If B_1, B_2, and B_3 are the corresponding 3D points, B_2 lies on line B_1B_3.

$$\frac{OA_2}{OB_2} = \frac{A_1A_2}{A_1A_3} \bullet \frac{OA_3}{OB_3} + \frac{A_2A_3}{A_1A_3} \bullet \frac{OA_1}{OB_1}$$

Equation 4.1

Using the parallel lines A_1A_3, B_2B_2' and B_3B_3', one can rewrite the disparities as:

$$\frac{OA_2}{OB_2} = \frac{OA_1}{OB_2'}$$
$$\frac{OA_3}{OB_3} = \frac{OA_1}{OB_3'}$$

Equation 4.2

Equation 4.1 can then be simplified to:

$$A_1A_2 \cdot OB_1 \cdot B_2'B_3' = A_2A_3 \cdot OB_3' \cdot B_1B_2'$$

Equation 4.3

Using the parallel lines A_1A_3, B_1O_1 and B_3B_3', one can write the following equations:

$$\frac{A_1A_2}{B_1O_1} = \frac{OA_2}{OO_1}$$
$$\frac{A_2A_3}{B_3O_3} = \frac{OA_2}{OO_3}$$

Equation 4.4

Using **Equation 4.4** to express the ratio A_1A_2 / A_2A_3 and then simplifying using the similar triangles OO_1B_1 and OO_3B_3', **Equation 4.3** becomes:

$$\frac{O_1B_1}{B_3O_3} = \frac{B_1B_2'}{B_2B_3'}$$

Equation 4.5

Since the lines B_1O_1, $B_2'B_2$, and $B_3'O_3$ are parallel, the second term of **Equation 4.5** is equal to O_1B_2 / B_2B_3 so we have:

$$\frac{O_1B_1}{B_3O_3} = \frac{B_1B_2'}{B_2B_3'} = \frac{O_1B_2}{B_2O_3}$$

Equation 4.6

Since the lines B_1O_1 and B_3O_3 are parallel, the angles $B_1O_1B_2$ and $B_2O_3B_3$ are equal. Together with **Equation 4.6**, these are sufficient for the triangles $B_1O_1B_2$ and $B_2O_3B_3$ to be similar. Thus the angles $O_1B_2B_1$ and $O_3B_2B_3$ are equal and since the points O_1, B_2 and O_3 are collinear by construction, the points B_1, B_2 and B_3 are also collinear. This concludes the proof that B_2 belongs to the line B_1B_3.

If the four neighboring samples are not coplanar, the subsamples define a general bilinear patch. Adjacent patches exhibit C^0 continuity. An alternative rendering algorithm, to save the cost of warping the sub-samples (dominated by the inverse computation, see **Equation 1.1**) is to *first* warp the reference-image samples and *then* interpolate.

4.1.2 Warp and then interpolate

The algorithm is sketched in **Pseudocode 4.2**.

```
// current sample is (u1,v1); u0 and v0 define the other 3 samples
//          that have already been warped at ($u0_2$, $v0_2$), ($u1_2$, $v0_2$),
//          and ($u0_2$, $v1_2$)
// ifu and ifv are the horizontal and vertical interpolation factors

($u1_2$, $v1_2$) = Warp(u1, v1, disp[u1, v1]);

for (i = 0; i < ifv; i++)
   for (j = 0; j < ifu; j++) {
      ($u_2$, $v_2$) = Bilinear(($u0_2$, $v0_2$), ($u1_2$, $v0_2$), ($u1_2$, $v1_2$), ($u0_2$, $v1_2$), i/ifv, j/ifu);
      currentColor = Bilinear(color[u0, v0], color[u1, v0], color[u1, v1], color[u0, v1],
                                        i/ifv, j/ifu);
      // process interpolated sample ($u_2$, $v_2$, currentColor)
}
```

Pseudocode 4.2. First the samples are warped and the bilinear interpolation occurs between the warped samples. Both the warping and the bilinear interpolation can be evaluated incrementally (not shown for readability).

This still avoids the triangle setup costs. Just as when interpolation is done in reference-image space, if the original samples lie on a plane, the sub-samples are also on the plane. The same proof can be applied to the desired image. **Figure 4.3** illustrates the algorithm.

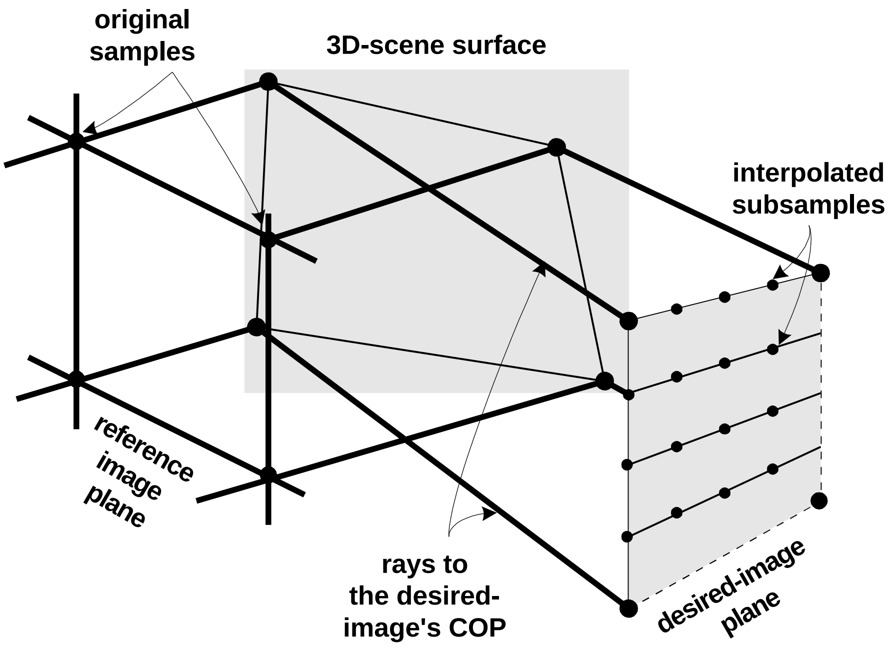


Figure 4.3. Warping followed by interpolation. The samples are warped and then subsamples are created by bilinear interpolation (in row-major order in the case shown[15]). The interpolation factor is again *4x4*.

As I pointed out in section 1.3.1, if the four original samples are not coplanar it is possible that the resulting screen-space quad is concave. Interpolating in reference image space and then warping produces a more accurate projection of the patch than warping and then interpolating. Although the IBRW architecture presented later is programmable and supports both variants of forward rasterization, such cases are infrequent and do not warrant the substantial extra cost of an inverse per subsample.

I analyzed the effects of interpolating in generalized disparity, and I showed that the generated subsamples belong to the same plane if the master samples are coplanar. Color however is not linear in image-space (a result well known from triangle rendering). In other words if the color of a scene surface varies linearly, the image-space variation is, in general, not linear. It is generally acceptable not to correct for the perspective non-linearity of color since the color differences are typically insignificant.

The cases where the artifacts are visible are the cases when very long polygons have a great depth variation (close to a normal angle with the image plane). An example is modeling the runway in a flight-simulator with a polygon. Not correcting for perspective produces an optical illusion that makes the user perceive the runway as being curved and standing up in the distance. This problem is probably not encountered anymore since runways are likely texture-mapped and not color interpolated. In the case of IBRW the "polygons" are very small in screen-space and linear color interpolation does not constitute a

[15] The exact same subsamples are obtained regardless of whether the interpolation is done in row-major or column-major order.

problem. The IBRW architecture presented later in this dissertation could compute perspective-correct colors, but the additional expense of a per-sample inverse is not justified.

Of course, not all four neighboring samples of the desired image should be connected since they might belong to different surfaces. A simple algorithm for deciding when to connect (and interpolate) and when not is presented next.

4.1.3 Depth Discontinuity Detection

One must segment the reference images into surfaces, and the continuity should be maintained only across the individual surfaces. Interpolating between the samples of two difference surfaces creates an artificial surface our research group usually calls a *skin*. The skins hide surfaces that become visible between the two original surfaces that slide apart in the desired image due to motion parallax. The surfaces are delimited by what I call depth discontinuities, and finding them can be done as a preprocess if the reference images are available before-hand or at rendering time if the reference images are acquired in real time. Because of the second case I wanted to develop a depth-discontinuity detection algorithm that is simple enough to be executed in real time.

I experimented with several simple filters on the generalized disparity map. For example, just testing the depth difference of adjacent samples against a threshold does not work since it is incorrectly sensitive to the surface orientation: if a surface is close to normal to the image plane, the sample-to-sample depth variation is considerable and the first-order filter incorrectly detects depth discontinuities.

I obtained good results with the second order derivative of the generalized disparity map, which computes the curvature of the surface. If the curvature is larger than a threshold, the samples are marked as disconnected. The threshold is unique per scene and is relatively easy to determine by trial and error. The detected depth discontinuities are examined and the threshold is adjusted according to whether the detector is too sensitive or not sensitive enough (**Figure 4.4**). The second derivative of the generalized disparity map is indeed a measure of surface curvature. If the surface sampled is planar, the second derivative is exactly zero. The informal argument is again the linearity in image space of the generalized disparity. The formal proof relies on the following claim.

Claim. If A_1, A_2, and A_3 are three collinear and equidistant reference image samples that sample a planar surface, their generalized disparities δ_1, δ_2, and δ_3 have the following property:

$\delta_2 - \delta_1 = \delta_3 - \delta_2$

The proof is similar to the proof that interpolated samples belong to the 3D line defined by the master samples (section 4.1.1), so it is not presented here.

In order to find depth discontinuities of all orientations, the second derivative is evaluated along four directions: N-S, E-W, NW-SE, NE-SW. The depth discontinuities are marked with flags that also store the orientation of the depth discontinuity (the relative position of the sample with which the current sample conflicts).

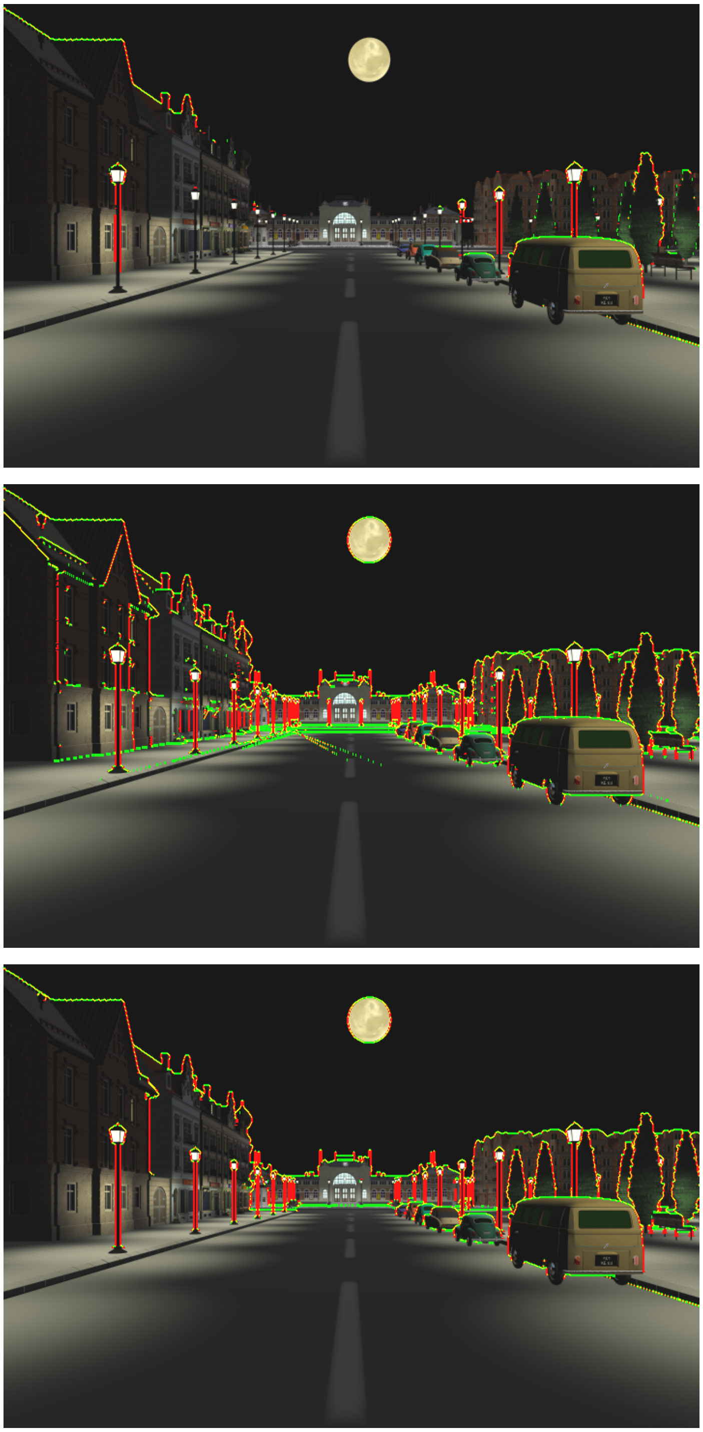


Figure 4.4. Threshold adjustment for depth-discontinuity detection. For the top image, the threshold is too big (insensitive). The middle image shows the case of a threshold that is too small. The bottom image shows the depth discontinuities detected with the preferred threshold.

Special care must be taken at the edges of the surfaces. All the images used in this work are either antialiased renderings or acquired images that necessarily collect area samples. Antialiasing an image introduces view-dependent information that cannot be used in other views. The color samples at the edge are a blend of the foreground and background surfaces. When the view changes, the closer surface moves relatively to the more distant, background surface and the blended samples of the background surface will make the silhouette of the foreground object incorrectly persist. What used to be the silhouette of the foreground object will also incorrectly be visible on the foreground object.

In order to prevent these artifacts one can discard the samples along the depth discontinuities. The trade-off is shaving off one or two samples from the edge of the surfaces. Sometimes the discarded samples are replaced with good samples from another view. If the object is thin however, no good samples can be found in any *nearby* reference image. Not being able to render features that are 1-pixel wide does not come as a surprise since it is known that rendering from images that are already filtered halves the maximum frequency.

I experimented with attempting to reverse the antialiasing by running a filter on the edge samples that increased the difference between neighboring samples. The results were not satisfactory as there is not enough information for accurately reversing the averaging occurring at the silhouette. Sometimes the samples were not changed enough and sometimes they were changed too much.

Accurate rendering of the edges is a difficult problem that has no perfect solution yet. It is one of the most important sources of artifacts in the images I generated. **Figure 4.5** illustrates the thin-features artifact. Also in **Figure 1.6**, the antenna-like structure at the nose of the Kamov helicopter (visible in the geometry-based rendering) disappeared in the image generated by warping.

In the case of scenes that have the depth and color acquired separately (the depth by a rangefinder and the color by a camera) the imperfect registration of color and depth aggravates the edge problem and more samples at the edges must be discarded to avoid the silhouette artifacts.

I discussed the effects of averaged color samples. Averaging at the edges occurs also in the depth channel. For example, when the laser beam of a rangefinder sweeps across an edge, the depth returned is an average of the depth of the two surfaces. Such a depth value cannot be used since it does not correspond to any surface and, when the viewing location changes from the acquisition location, the sample will appear to be floating in the air. All such depth samples have to be detected and eliminated before rendering.

A different depth discontinuity detection algorithm is given in [McAllister99] that relies on an image from a different view in order to find where to disconnect the samples of the current image. If the second image "sees" between two surfaces then the surfaces belong to different objects and should not be connected. The algorithm does not rely on any heuristic but is much more laborious and cannot be executed at rendering time.

4.1.4 Computation of the Interpolation Factor

The continuity of the surfaces is ensured by determining an appropriate interpolation factor. In the case of the classic backward rasterization, all the framebuffer locations that are covered by the image-plane projection of the polygon are guaranteed to get a sample of the polygon. In the case of forward rasterization, an interpolation factor that is too small causes holes in the surfaces. If the interpolation factor is too large however, several subsamples will land at the same warpbuffer location, which is inefficient.

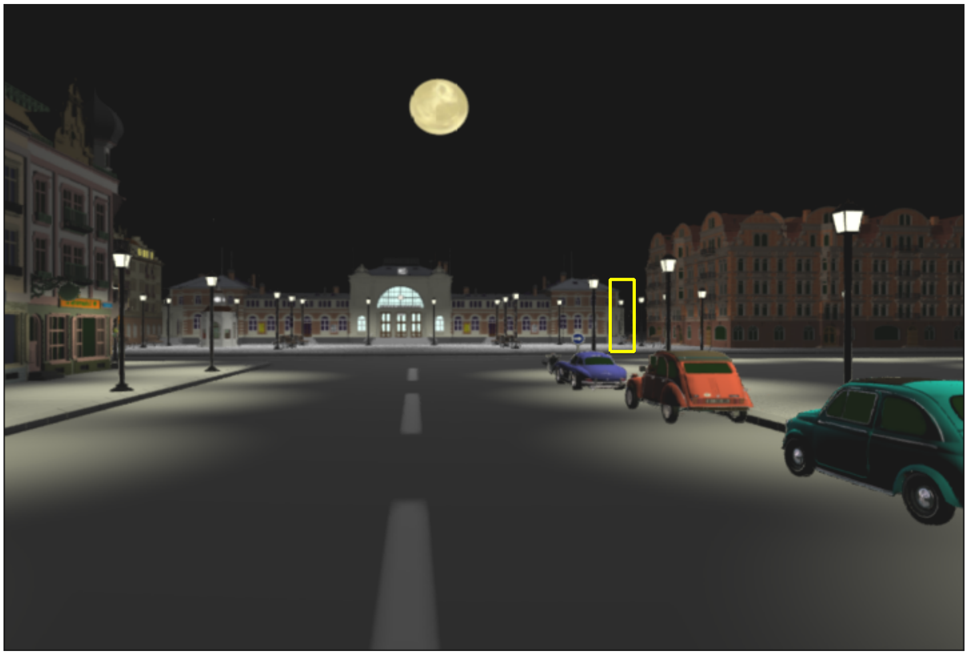


Figure 4.5. Elimination of thin features as edge samples are discarded. The artifact is well illustrated by some of the light poles in the distance, like the one in the box.

Determining the minimum interpolation factor that guarantees that all the warpbuffer locations get at least one subsample is essential to the forward rasterization method. The appropriate interpolation factor is given by **Equation 4.7**.

$$ifu = \sqrt{2}\max(V_0V_1, V_2V_3)$$
$$ifv = \sqrt{2}\max(V_0V_3, V_1V_2)$$

Equation 4.7. $V_0V_1V_2V_3$ is the image-plane quad and *ifu* and *ifv* are the interpolation factor values along the two directions.

Claim. If a quad is bilinearly interpolated with an interpolation factor given by **Equation 4.7**, every warpbuffer location it covers will get at least one subsample.

The proof is in two steps. First I will show that limiting the length of the sides of quads in a mesh guarantees that every warpbuffer location gets at least a node of the mesh. Next I will give a way of limiting the length of the sides of the bilinear-interpolation mesh by limiting the length of the sides of the mesh that are at the perimeter of the mesh.

Claim. Consider a mesh of quads with sides at most 1/sqrt(2) pixels[16]. Every pixel covered by the mesh has at least one node of the mesh (vertex of the quads) within its boundaries.

[16] For convenience throughout this proof I will use the term *pixels* for *warpbuffer locations*.

Proof (see **Figure 4.6**). Let's assume that that there is a pixel (covered by the mesh) with no mesh node within its boundaries. Let $V_1V_2V_3V_4$ be the quad that contains the center of that pixel. Such a quad exists since the mesh covers the pixel, including its center.

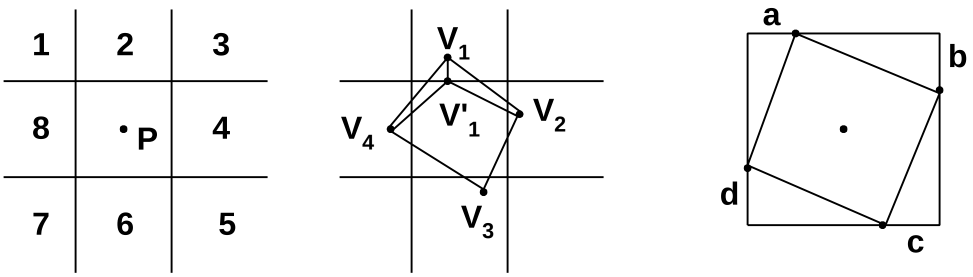


Figure 4.6. Ensuring at least one sample per pixel by limiting the length of the mesh segments. P is the center of the hypothetical pixel that does not contain a node of the mesh. $V_1V_2V_3V_4$ is the quad of the mesh that contains P. The possible locations for the vertices define eight zones (numbered 1 to 8).

Next I will analyze the possible locations of the vertices of the quad.

None of the four vertices can belong to one of the zones 1, 3, 5, 7. If a vertex belongs to zone 1 let's say, then the adjacent vertices belong in zones 8 or 2 since the sides cannot be longer than 1/sqrt(2) pixels and sides longer than 1 pixel would be needed to reach zones 3 or 7. The remaining vertex must go to a different zone in order for the resulting quad to contain the center of the pixel. However, any of the remaining zones are more than 1 pixel away from either zone 2 or zone 8.

At most one vertex can belong to any of the zones 2, 4, 6, and 8. If all four vertices belong to, let's say, zone 2, then the resulting quad doesn't contain the center of the pixel. If three vertices belong to zone 2, the fourth vertex can belong to zone 4 or 8 only if one of the sides of the quad is a diagonal in the pixel, which is impossible since the diagonal is longer than the limit allowed. If the fourth vertex belongs to 6 it will be more than a pixel away from the other three vertices, which again, is impossible. If two vertices belong to one of the even-numbered zones, by similar reasoning, the other two vertices cannot be placed such that no two vertices are farther apart than the imposed limit.

We established that the quad has to have its vertices in the zones 2, 4, 6, and 8. In the second (central) subfigure in **Figure 4.6** we built an alternative position for one of the vertices by projecting it on the side of the pixel. V_1V_4 and V_1V_2 are the largest sides of the triangles $V_1V'_1V_4$ and $V_1V'_1V_2$ respectively since they are opposite to angles larger than 90 degrees, which are guaranteed to be the largest angles in the triangles. Consequently $V'_1V_4 < V_1V_4$ and $V'_1V_2 < V_1V_2$. Thus the quad $V'_1V_2V_3V_4$, which also contains the center of the pixel, has the same property of having sides no longer than 1/sqrt(2). According to similar reasoning, one can project all the vertices to the corresponding pixel sides (see third subfigure). **Equation 4.8** uses the maximum length of the sides and establishes a contradiction. This invalidates the hypothesis that there is a pixel with no mesh-node within its boundaries.

$$a^2+(1-d)^2<\frac{1}{2}, b^2+(1-a)^2<\frac{1}{2}, c^2+(1-b)^2<\frac{1}{2}, d^2+(1-c)^2<\frac{1}{2}$$

$$a^2+b^2+c^2+d^2-a-b-c-d+1<0$$

$$(a-\frac{1}{2})^2+(b-\frac{1}{2})^2+(c-\frac{1}{2})^2+(d-\frac{1}{2})^2<0$$

Equation 4.8.

Consequently it is enough to ensure that the sides of the quads of the bilinear-interpolation mesh are less than 1/sqrt(2) pixels in length. The next claim provides a convenient way of doing just that.

Claim. Consider a quad $V_0V_1V_2V_3$ that is bilinearly interpolated. Let $V_{0x}V_{1x}V_{2x}V_{3x}$ be a random sub-quad formed by four neighboring sub-samples generated by interpolation. Then the longest of $V_{0x}V_{1x}$ and $V_{2x}V_{3x}$ is at most as long as the longest of the sub-segments that are part of V_0V_1 and V_2V_3.

Proof

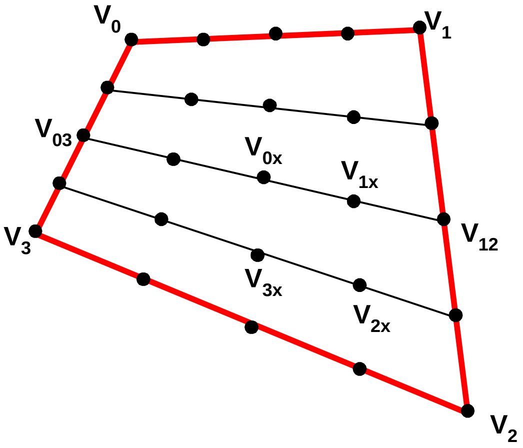


Figure 4.7. Subsample mesh independent of the interpolation order. The quad $V_0V_1V_2V_3$ is bilinearly interpolated in row-major order.

When bilinearly interpolating the same nodes are obtained no matter what interpolation direction is chosen first, so it is sufficient to consider only one direction. In **Figure 4.7** the interpolation is done “vertically” first and then "horizontally".

First I will prove that the medial segment $V_{03}V_{12}$ is shorter than the maximum of the sides V_0V_1 and V_2V_3. When interpolating, all the incremental segments are of equal length, so if $V_{03}V_{12}$ is shorter than *max*(V_0V_1, V_2V_3), then the incremental segments along $V_{03}V_{12}$ will be at most as long as the incremental segments along the maximum of the two sides. One can subdivide the quad recursively, which concludes the proof that the segments along any of the interpolation lines will be shorter than the segments on the maximum side.

In order to prove that the medial segment is shorter than the maximum of the two quasi-parallel sides I will use the inequality of the quadratic mean shown in **Equation 4.9**.

$$\sqrt{\frac{a^2+b^2}{2}} \leq \max(a,b)$$

Equation 4.9

The last step of the proof consists of proving that the medial segment is shorter than the quadratic mean of the opposite sides (**Equation 4.10**).

$$\left(\frac{(x_0+x_3)}{2}-\frac{(x_1+x_2)}{2}\right)^2+\left(\frac{(y_0+y_3)}{2}-\frac{(y_1+y_2)}{2}\right)^2 \leq \frac{(x_0-x_1)^2+(y_0-y_1)^2+(x_2-x_3)^2+(y_2-y_3)^2}{2}$$
$$x_0{}^2+x_1{}^2+x_2{}^2+x_3{}^2-2x_0x_1-2x_2x_3-2x_0x_3-2x_1x_2+2x_0x_2+2x_1x_3+Y \geq 0$$
$$(x_0+x_2-x_1-x_3)^2+(y_0+y_2-y_1-y_3)^2 \geq 0$$

Equation 4.10. The image plane coordinates of vertex V_i are given by (x_i, y_i).

Since the interpolation factor must be a pair of integers, one chooses the smallest integer that is larger than the quantities prescribed by **Equation 4.7**. Thus the interpolation factor is given by **Equation 4.11**.

$$ifu = \left\lceil \sqrt{2}\max(V_0V_1, V_2V_3) \right\rceil$$

$$ifv = \left\lceil \sqrt{2}\max(V_0V_3, V_1V_2) \right\rceil$$

Equation 4.11. Computation of interpolation factor for a quad $V_0V_1V_2V_3$.

4.1.5 Visibility Testing Along Different Rays

An important observation is that in the case of forward rasterization, the two samples tested in the visibility z-test do not belong to the same ray. The rays are close, since the samples land at the same warpbuffer location, but are not identical. As described in Chapter 1, this can potentially lead to incorrect visibility determination. In the case of IBRW, such cases are extremely rare and do not constitute a reason for concern. As we will see in section 4.2, the warpbuffer is supersampled, which reduces the differences in orientation between the rays that intersect the same warpbuffer location. Note that none of the researchers that use splatting for reconstruction do not report this problem, even though they splat in the output framebuffer.

In conclusion, generating subsamples by parameter-space interpolation maintains surface continuity where desired. It is not just an alternative solution to the backward rasterization of the triangle mesh, it is also cheaper since it reduces the per-sample setup cost, as will be shown in the next section where the two rasterization methods are compared in detail.

4.1.6 Setup-cost comparison between forward and backward rasterization

As described in the previous sections, after the depth-and-color samples are warped, one must rasterize the resulting screen-space quads in order to reconstruct the final image. **Table 4.1** gives a side-by-side comparison of the code required to rasterize a quad according to the classic and to the forward rasterization methods. The screen-space quad is defined by $V_i(x_i, y_i)$, i = 0, 1, 2, 3. A count of the required arithmetic operations is given for each stage. **F** stands for floating-point, **I** for integer and **A**, **M**, **D** for add, multiply and divide, respectively. The total number of parameters interpolated, including x and y, is denoted by *params*.

The backward-rasterization code using edge equations is taken from the PixelFlow quad rasterizer. The quad is rasterized by subdividing it into two triangles along a diagonal. The rasterization code is given only for one of the two triangles. First the bounding box is computed. Then the edge equations are used for testing whether the current pixel-center (or sampling location, in the case of supersampling) is inside the triangle or not. Then one must establish the parameter-plane equations for each of the parameters. These will give the discrete parameter derivative along the x and y directions. The last step is the rasterization itself.

Forward rasterization begins with computing the interpolation factors according to the formulas derived in section 4.1.4. Since the interpolation factors are typically small integers, the square root and the inverses can be looked up in most of the cases. Moreover, in the IBRW case, the same interpolation factors are used throughout the tile so the computation can be amortized over 256 quads. The rasterization is equivalent to bilinear interpolation of the parameters. Assuming that R, G, B and z are the only parameters interpolated (*params* = 6), the number of floating-point operations required at rasterization setup using the classic method is:

20FA + 18FA + 12FM + 4FA + 2FD + 56FA + 64FM + 24FM + 24FA = 122FA + 100FM + 2FD

The forward method requires the following operations at setup:

5FM + 3FA + 24FA + 18FM = 27FA + 23FM

The setup costs above include all operations independent of the quad size, and were obtained from the costs given in **Table 4.1** by ignoring the *X*, *Y*, *ifx* and *ify* terms. Assuming that a floating-point multiply is about as costly as a floating-point add, the setup for rasterizing a quad is between 4 and 5 times less expensive in the case of forward rasterization.

Backward rasterization	**Forward rasterization**

For each of the two triangles	*Compute interpolation factor ifu, ify, and some common expressions*
	float $l_{01} = (x_1-x_0)* (x_1-x_0)$;
	float $l_{30} = (x_3-x_0)* (x_3-x_0)$;
	if ($l_{01} < l_{30}$)
	ifx = ceiling(sqrt(2*l_{30})); // look up
	else
	ifx = ceiling(sqrt(2*l_{01})); // look up
	same for ify
	float inv_ifx = 1/ifx; // look up
	float inv_ify = 1/ify; // look up
	float inv_ifxify = inv_ifx *inv_ify
	(1FM + 1FM + 1FA) x 2 + 1FA + 1FM = 5FM + 3FA

Compute bounding box
int tmp = (int) x_0;
int minx = tmp; int maxx = tmp;
tmp = (int) x_1;
if (tmp < minx) minx = tmp;
if (tmp > maxx) maxx = tmp;
tmp = (int) x_2;
if (tmp < minx) minx = tmp;
if (tmp > maxx) maxx = tmp;
int X = (floor) maxx - (floor) minx + 1;
same for y
(1FA + 1FA + 1IA + 1IA + 1FA + 1IA + 1IA + 2FA + 2IA) x 2 = 10FA + 12IA	

Compute edge equations
float $e^{01}_{A} = y_0 – y_1$;
float $e^{01}_{B} = x_1 – x_0$;
float $e^{01}_{C} = x_0y_1-x_1y_0$;
same for the other two edges
**(1FA + 1FA + 2FM + 1FA) x 3 =
= 9FA + 6FM**
	For each parameter, including x and y
	float $d_1 = p_1 - p_0$;
	float $d_2 = p_3 - p_0$;
	float $d_3 = p_2 - p_3$;
	float hincr = d_1 * inv_ifx;
	float deltahincr = ($d_3 – d_1$) * inv_ifxify;
	float vincr = d_2 * inv_ify;
	float $p_{line} = p_0$;
	for (i = 0; i < ify; i++)
	float p = p_{line};
	for (j = 0; j < ifx; j++)
	float p += hincr;
	p_{line} += vincr;
	hincr += deltahincr;
	(1FA + 1FA + 1FA + 1FM + 1FA + 1FM + 1FM + (1FA + 1FA + (1FA) x ifx) x ify) x params = (4FA + 3FM + (2FA + FA x ifx) x ify) x params

Compute area
area = $e^{01}_{C} + e^{12}_{C} + e^{20}_{C}$
inv_area = 1 / area
2FA + 1FD	

For each parameter p, but not for x and y, compute parameter planes
float $d_0 = p_0 – p_1$;
float $d_1 = p_1 – p_2$;
float $p_A = (e^{12}_{A}d_0 - e^{01}_{A}d_1)$ * inv_area;
float $p_B = (e^{01}_{B}d_1 - e^{12}_{B}d_0)$ * inv_area;
float $p_C = p_0 – p_Ax_0 – p_By_0 + .5$
**(1FA + 1FA + 2FM + 1FA + 1FM + 2FM + 1FA + 1FM + 2FM + 3FA) x (params – 2) =
(7FA + 8FM) x (params – 2)**	

For each parameter, including edge expressions float $p_{line} = p_Ax_{min} + p_By_{min} + p_C$; for (i = 0; i < Y; i++) float p = p_{line}; for (j = 0; j < X; j++) p += p_A; p_{line} += p_B; **(2FM + 2FA + (1FA + (1FA) x X) x Y) x params**	

Table 4.1 Quad rasterization according to the backward and forward methods.

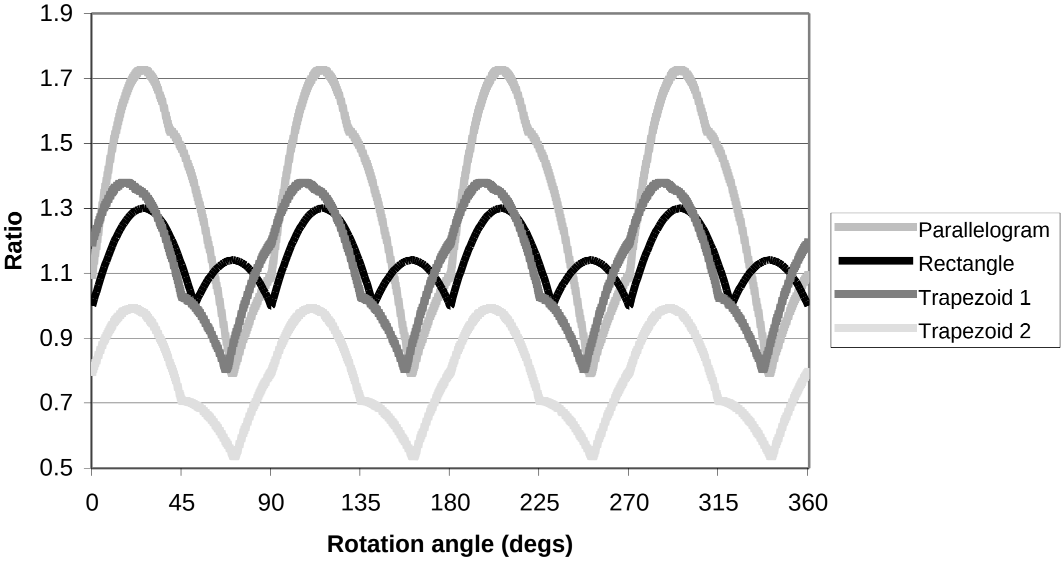
4.1.7 Inner-loop comparison between forward and backward rasterization

The next important question is how the inner-loops of the two rasterization methods compare to each other. Note that for both algorithms the inner loop consists of incrementing the interpolation parameters. It is therefore sufficient to compare the number of times the inner-loop is executed. For the classic method, the inner loop is executed X_1 *x* Y_1 + X_2 *x* Y_2 times (using the notation introduced in the previous section), corresponding to the sum of the areas of the bounding boxes of the two triangles that form the quad. For the forward method, the inner loop is executed *ifu x ifv* times.

I compared the two quantities for various quads. For each quad I tested all possible orientations (with one-degree increments). The results are given in **Graph 4.1**. The test-quads used are shown in **Figure 4.8**.

It is interesting to note that for the rectangle, the classic rasterization algorithm always executes the inner loop more times than the forward rasterization algorithm (ratio greater than one). This is of course due to the overlap between the bounding boxes of the two triangles, and a formal proof of the result is relatively simple. The minima in the graph correspond to axis aligned rectangle sides or axis aligned diagonal (along which the rectangle is split). In such a case the ratio is exactly one and the total number of times the inner loop is executed is *2ab*, where a and b are the sides of the rectangle.

For the parallelogram, the minima are reached when the diagonal is axis aligned. The minima are below one, so for some orientations of the parallelogram the forward rasterization algorithm executes the inner loop more times than the classic algorithm (ratio below one).



Graph 4.1 Ratio between the number of times the inner loop is executed when quads are rasterized using the classic, backward method versus the forward method.

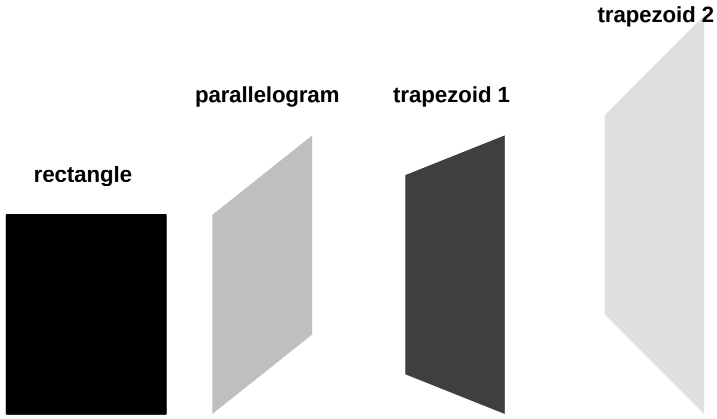


Figure 4.8 Various test quads

The two algorithms behave similarly in the case of *trapezoid 1*. The worst case for the forward rasterization algorithm corresponds to quads that have opposite sides of very different length, as is the case for trapezoid 2. For such a quad, numerous redundant iterations occur when interpolating close to the shorter side. In the case of IBRW, such quads are unlikely. First, in IBRW the quads are small in screen space; since the length of the sides is limited, the difference in length is also limited. Second, the quads are one-pixel squares in the reference image; typically, the desired view is not radically different from the reference view, so the shape of the quad in the desired view does not become too unbalanced.

Another important aspect of the comparison of the two methods is the number of samples produced. The classic method generates the minimum number of samples, which corresponds to the area of the quad (or polygon, in general). The forward rasterization method can guarantee surface continuity only at the price of some redundant samples. Redundant samples are costly if shading them is costly. In the case of IBRW, shading is inexpensive since all that is required is simple color interpolation. Moreover, the average interpolation factor is small, so substantially reducing setup remains a big advantage.

This concludes section 4.1 that describes how surface continuity is maintained between warped samples at a considerably lower cost than the cost of conventionally rasterizing two triangles per warped sample.

4.2 Higher-resolution warpbuffer

At the beginning of the chapter, I enumerated as one of the sub-problems of IBRW the need to reconstruct the final pixel from several color samples. In order to accommodate several color samples for each final-image pixel, the warpbuffer must have a higher-resolution than the final image.

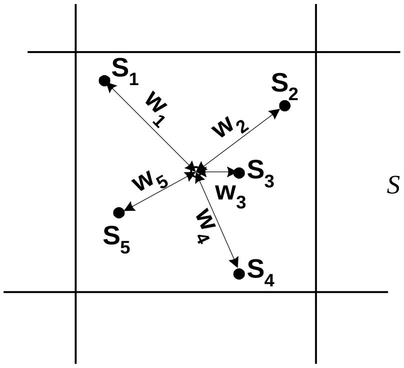
A higher warpbuffer resolution does not make the warping equation more expensive: it merely changes a few of the transformation parameters. However it does imply higher interpolation factors (in order to cover more warpbuffer locations). Good results were obtained when the warpbuffer was twice as dense in each direction than the final image, which implies four color samples per pixel.

The third sub-problem of IBRW is accurate reconstruction. The next section describes a reconstruction technique I developed to work in conjunction with the forward rasterization method.

4.3 Delaying the Inverse Mapping Until the Last Moment: Offset-Reconstruction

It is known that good reconstruction / resampling can be obtained by computing the final pixel color as a weighted average of the subsamples close to its center. One should not consider subsamples that are farther than a pixel from the center of the pixel to be computed. In practice, good results were obtained even if one averages only the subsamples that are within the boundaries of the pixel.

In the case of backward-rasterized polygons, one can compute color at locations defined with respect to the pixel boundaries. In the case of supersampling, for example, each pixel is evaluated at a set of locations within its boundaries. The subsamples that are obtained are averaged using weights according to their distance to the center of the pixel (**Figure 4.9**).



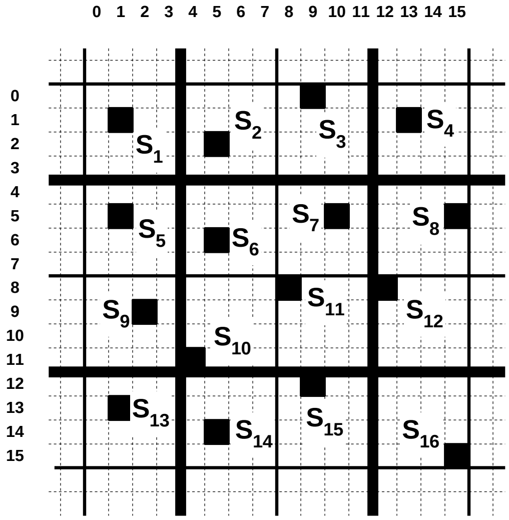
$$S = \frac{S_1 \cdot w_1 + S_2 \cdot w_2 + S_3 \cdot w_3 + S_4 \cdot w_4 + S_5 \cdot w_5}{w_1 + w_2 + w_3 + w_4 + w_5}$$

Figure 4.9. Jittered supersampling. The pixel is evaluated by backward rasterization at five locations, resulting five subsamples S_1 to S_5. The final pixel value S is computed as a weighted average, where the weights depend on the distance to the center of the pixel. The function that describes the weight - distance dependence is a nonlinear decreasing function (e.g. a gaussian, a raised cosine). The locations of the subsamples are known beforehand so the weights can be normalized, or, in the case of integer weights, the inverse of their sum is precomputed, which eliminate the per-pixel inverse computation.

The forward rasterization technique, by definition, doesn't have the ability to produce subsamples at precisely specified image-plane locations. Although the subsamples land at locations that are independent of the warpbuffer-locations grid, they are nonetheless well specified. Just truncating the warping result and assuming that the subsample lands at the center of the warpbuffer location produces noticeable aliasing, even when the warpbuffer is refined at twice the resolution of the final image, in each direction.

The solution is to compute the location of the warp at a higher precision than the precision of the warpbuffer and save the extra information to be used at reconstruction. This can be efficiently done by using a pair of *offsets* that locate the warp within the warpbuffer location. I have found that two 2-bit offsets are sufficient to satisfactory reduce aliasing, thus the extra warpbuffer storage is negligible. Two bits locate the subsample to within a quarter of warpbuffer location or one eighth of the final-image pixel.

The forward rasterization algorithm remains unchanged. Each warpbuffer subsample (generated by warping reference-image-interpolated subsamples or by interpolating warped samples) is z-compared with the current closest subsample at that warpbuffer location. If the new subsample is closer, the z, color and offset information is updated. After all samples are warped and interpolated and all subsamples are z-buffered, the warpbuffer colors and offsets are used to reconstruct the final image.



$$S = \frac{S_1 \cdot w_{1,1} + S_2 \cdot w_{2,5} + S_3 \cdot w_{0,9} + \ldots + S_{16} \cdot w_{15,15}}{w_{1,1} + w_{2,5} + w_{0,9} + \ldots + w_{15,15}}$$

Figure 4.10. Reconstruction using offsets.

The reconstruction is illustrated in **Figure 4.10**. The thick lines delimit the final image pixels. The warpbuffer is two times more refined in each direction. There are four warpbuffer locations inside each pixel (shown with the thinner continuous lines), thus there are exactly four color samples per pixel. The locations where the subsamples S_1 to S_{16} land are recorded with two 2-bit offsets. The offsets achieve a virtual supersampling of the warpbuffer (shown with dotted lines).

The final color pixel is computed as a weighted average of the subsamples that land (approximately) within one pixel of the center of the pixel. Consequently, in the case presented in the figure, 16 subsamples are used to reconstruct the final pixel. The weights used to modulate each subsample's contribution depend on the offsets. For example subsample S_{11} (offsets (0, 0)) is assigned a much higher weight ($w_{8,8}$) than the weight ($w_{11,11}$) it would have been assigned if it landed at the bottom right corner of the warpbuffer location (offsets (3, 3)).

The kernel used for reconstruction / resampling stores *16x16* weights and only 16 are used at each sample. Not using all the weights has the consequence of having to compute the sum of the weights for each pixel and then invert it. With integer weights, the inversion is not a performance-affecting burden. The number of possible sums is too large for a single lookup table. One could imagine a collection of lookup tables that subdivide the sparse domain of possible sums in linear subintervals.

The kernel shape I used is a raised cosine, with the two zeroes placed at the corner of the *2x2* pixel neighborhood. For integer weights, all the weights are divided by the smallest weight. In my experiments, the best balance between blurriness and aliasing was obtained when the cosine was raised to the second or third power.

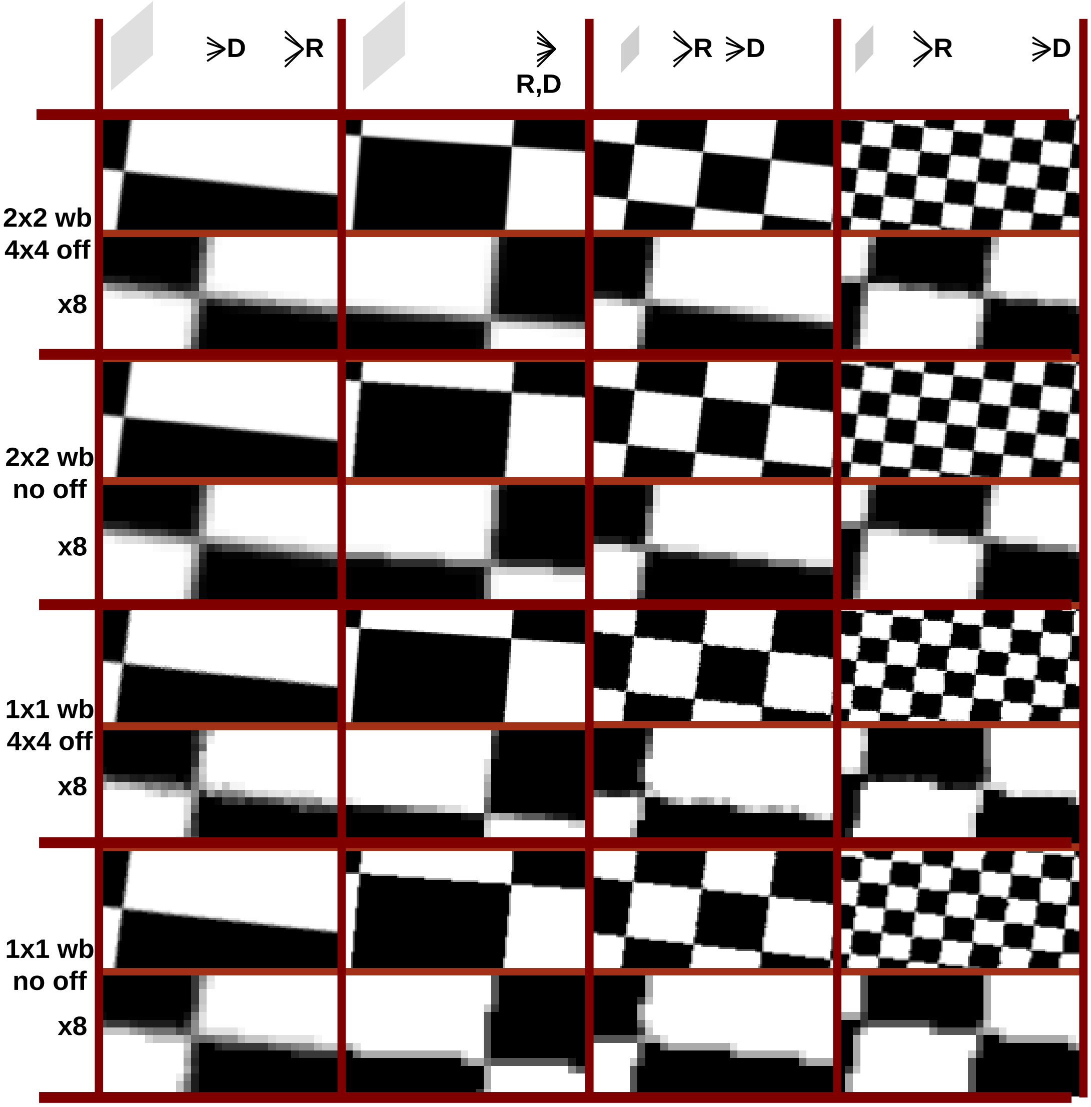


Figure 4.11. Offset reconstruction study

Figure 4.11 shows images of a checkerboard reconstructed with several parameter combinations. The images are obtained by warping a reference image of the checkerboard to four desired views. The table of images has four columns corresponding to the four desired views. The desired view is slightly rotated around the z-axis to test the antialiasing properties on the edges of the checkers. The first column corresponds to a location that is closer to the grid than the location of the reference image. For the second column the only difference between the reference and desired views is the rotation along the z-axis. In the third and fourth columns the desired camera is farther and farther away from the reference position.

The rows show four reconstructions, both in normal and magnified format (*x8*). From top to bottom they are: *2x2* supersampled warpbuffer and *4x4*[17] offsets, *2x2* warpbuffer and no offsets, *1x1* warpbuffer (same resolution as output image) and *4x4* offsets, and *1x1* warpbuffer and no offsets.

In column 1 the interpolation alone generates sufficient intermediate gray levels for an acceptable reconstruction, and the offsets are not necessary.

In column 2, the offsets improve substantially the image-quality. Even when the warpbuffer has the same resolution as the final image, the reconstruction is aliasing free.

For the top half of column 3 the use of offsets helps considerably. The *x8* magnifications show more intermediate gray levels. In the bottom half of column 3 aliasing occurs since the level of detail of the original image is too high for the *1x1* warpbuffer. Similarly in column 4, the camera is very far away and the excessive level of detail cannot even be accommodated by the 2x2 warpbuffer. In the common case, when the desired image required sampling of the surface is within 40-250% of the reference image sampling, the 2x2 warpbuffer with 4x4 offsets provides a good reconstruction. Outside that interval different reference images should be used, and Chapter 6 describes how to select the appropriate samples for a certain desired view.

4.3.1 Temporal antialiasing

So far I have described how the offset reconstruction method keeps aliasing under control, and the results were illustrated with still frames. However, obtaining antialiased stills is not sufficient. For a high-quality high-framerate exploration of a scene, one has to address the problem of temporal aliasing.

Temporal aliasing is abrupt, non-uniform changes in the color of a pixel from one frame to the next. This is a complicated problem and it was addressed by many researchers. A common problem of all the antialiasing methods used in conjunction with the classic backward rasterization of polygons is what has been called "the problem of slowly moving edges". Since the locations at which the pixel is evaluated are fixed with respect to the pixel boundary, any two such locations are collinear. In an unfavorable situation, two or more subsamples can change sidedness from one frame to the next, causing a large color change. If the edge is moving slowly, the color pop is especially disturbing.

If antialiasing is achieved by computing, let's say, *4x4* regular pixel coverage masks, and the edge is almost horizontal, it can happen that 4 subsamples change color from a frame to the other, causing a very noticeable artifact (**Figure 4.12**). This is precisely the reason jittered supersampling is preferred to regular supersampling, but even so, any two subsamples will always be collinear. A possible solution is to increase the resolution of the coverage masks, which is very expensive (even *4x4* masks are too expensive and are not used in practice). An interesting but still expensive solution is proposed by [Schilling91]: the coverage

[17] Two 2-bit offsets, producing a *4x4* virtual supersampling.

masks are determined by the area covered (scalar) that is then converted into the most appropriate coverage pattern. For the case presented in **Figure 4.12**, the transition between 12-4 and 8-8 is made gradually by going through 11-5, 10-6 and 9-7.

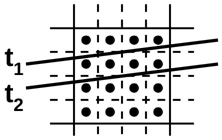


Figure 4.12. Temporal aliasing. From frame t_1 to frame t_2, the coverage mask changes from *4-12* to *8-8*, causing an important color change.

The forward rasterization followed by offset reconstruction suffers less from temporal aliasing. The edge can move continuously on the warpbuffer location (limited by the offset resolution). The offsets, together with using a 2-pixel wide reconstruction kernel, guarantee a sufficient number of intermediate levels for a slowly moving edge.

4.4 Conclusion

I have presented a new rasterization technique that relies on continuous-parameter-space interpolation, which used together with offset reconstruction, generates high-quality images efficiently. The fundamental observation is that visibility can be solved before the inversion to the discrete pixel grid (**Figure 4.13**).

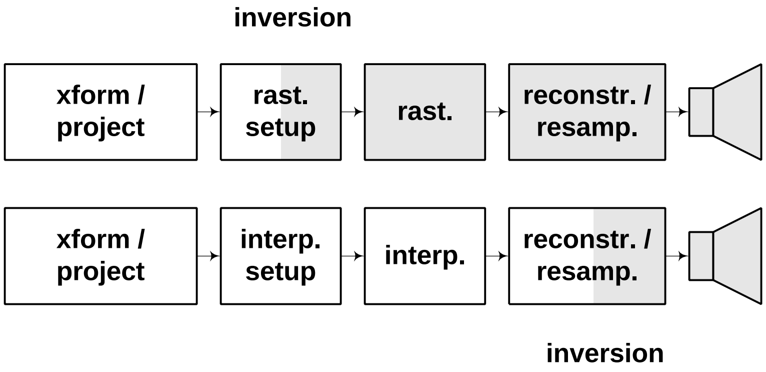


Figure 4.13. Graphics pipeline with forward rasterization. In the classic graphics pipeline, the inversion from model space to pixel-grid occurs at setup. With forward rasterization the inversion is delayed until the last moment, and is done at the reconstruction / resampling stage when the color at the center of the pixel is computed.

The next chapter presents the WarpEngine [Popescu00], an architecture for IBRW that relies on the algorithm described.

Chapter 5 WarpEngine

5.1 Parallel Warping

5.1.1 Cost of IBRW

In order to estimate the cost of IBRW one must estimate the number of samples that must be warped each second and the bandwidth to the buffer that stores the warped samples (warpbuffer). It is assumed that one needs as a working minimum two samples per output image pixel. This figure obviously depends on how the scene is modeled with depth images and on how efficiently one chooses the samples needed for each frame. The best current solutions for these complex problems are described in Chapter 6; for the purpose at hand we believe it is sufficient to point out various reasons why the ideal one sample per output pixel is exceeded:

- *Oversampling*: some surfaces are better sampled in the reference (input) image than in the desired image, which leads to more than one warped sample per output pixel.
- *Depth complexity*: it is impossible to efficiently discard, before warping, *all* samples that are not visible in the current view.
- *Redundancy*: some surfaces are sampled by more than one reference image; not *all* such redundant samples are discarded before warping.

The number of samples that must be warped each second (with the two samples-per-output-pixel assumption) is given in **Table 5.1**. The estimates are given for three output resolutions:

- VGA (640 x 480 pixels, or approximately .3 megapixels)
- XVGA (1280 x 780 pixels, or approximately 1 megapixel)
- HDTV (2048 x 1024 pixels, or approximately 2 megapixels)

Output resolution		**VGA** 0.3 Mpix	**XVGA** 1.0 Mpix	**HDTV** 2.0 Mpix
MWarps / s	30 Hz	18	60	120
	60 Hz	36	120	240

Table 5.1 Number of warps per second required for various output resolutions

With *2x2* warpbuffer supersampling we found that every sample generates between 8 and 16 subsamples that have to be z-buffered in the warpbuffer. The figure depends on the scene and on how it is modeled (number and placement of depth images). Assuming 8 byte subsamples (color and z), a total depth complexity of two[18], and that 50% of the hidden subsamples are initially visible, there will be on average 10 byte warpbuffer accesses per subsample. This translates to the warpbuffer bandwidths (16 subsamples per sample) shown in **Table 5.2**.

Output resolution		**VGA** 0.3 Mpix	**XVGA** 1.0 Mpix	**HDTV** 2.0 Mpix
GB / s	30 Hz	2.88	9.6	19.2
	60 Hz	5.76	19.2	38.4

Table 5.2. Warpbuffer bandwidth requirement for various output resolutions

At the present, a single ASIC can provide neither the warping power nor the warpbuffer bandwidth required. Several ASICs must be used and thus the need for a high-level parallelization scheme to distribute the work.

5.1.2 Sort-first, -middle and -last taxonomy

A useful taxonomy for describing parallel polygon-rendering architectures is the sort-first, sort-middle and sort-last taxonomy formalized in [Molnar94].

An architecture is said to be sort-first if the primitives (polygons, higher order surfaces) are sorted before transformation according to the screen region onto which they map and are then assigned to renderers that completely render their screen region (**Figure 5.1**). Sorting the primitives implies pre-transformation computations, which are typically substantially less expensive than fully transforming each primitive. Examples of sorting techniques include computing screen-space bounding boxes of objects composed of many polygons, or coarse pre-tessellation of higher order surfaces ([Mueller97], [Mueller00]).

Sort-middle architectures sort fully transformed primitives. The geometry processors, separated from the rasterizers, transform a subset of the primitives and send them to the rasterizers that are each responsible for a portion of the screen. The rasterizers process only the primitives pertinent to their screen region (**Figure 5.1**).

Sort-last architectures are composed of several renderers that fully process a subset of the primitives. The final image is obtained by compositing the partial images built by the individual renderers. Sorting occurs after rasterization, as samples (subsamples) of the primitives are composited (**Figure 5.1**).

Considering the depth samples as primitives, the taxonomy can be readily applied to IBRW. Transforming the primitives corresponds to warping the depth samples, and (backward) rasterization is replaced by interpolation between the warped samples (forward rasterization). The next subsection will investigate the suitability of the three types of architectures for IBRW.

[18] Computed as the ratio between the number of subsamples and the number of warpbuffer locations.

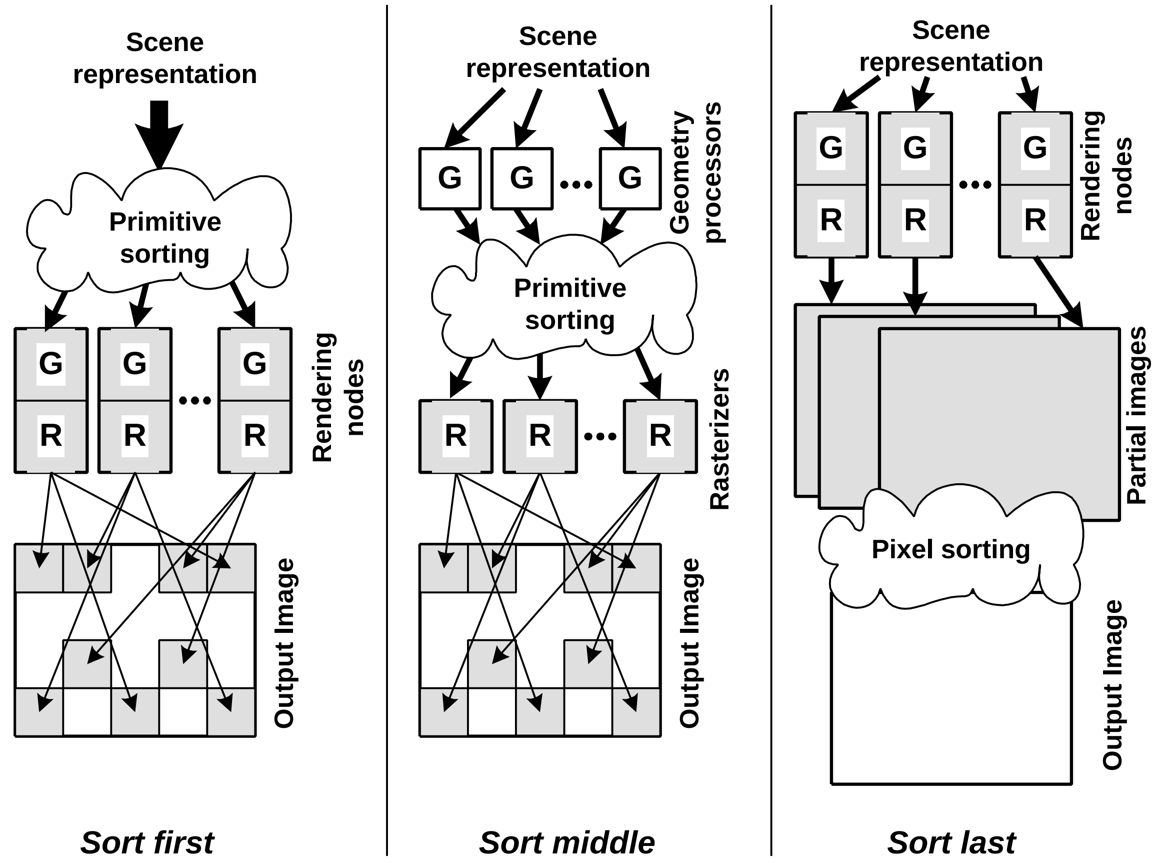


Figure 5.1. Sort-first, sort-middle and sort-last parallel graphics architectures. For sort-first and sort-middle, the output-image regions left white are rendered by nodes not shown.

5.2 Sort-first for IBRW

5.2.1 Why not sort-middle?

For polygon-rendering, a natural place to sort the primitives is after they are transformed (UNC's *Pixel-Planes 5* [Fuchs89], SGI's *RealityEngine* [Akeley93] and SGI's *InfiniteReality* [Montrym97]). The transformation burden can be equitably distributed among the geometry processors and the transformed primitives are usually small enough in screen space to map to only one screen region, thus requiring processing by only one rasterizer.

One problem with sort-middle is the high communication cost, which is proportional to the number of visible primitives. Complex scenes, finely modeled for high-quality renderings, imply a large number of primitives, which in turn translates into large communication costs. This disadvantage is accentuated with IBRW as the number of primitives (samples) is at least twice the number of output pixels. Another problem normally associated with sort-middle is poor load balancing among the rasterizers. Some regions of the screen may get a large number of primitives. Primitive clumping is due to:

- poor or no visibility culling inside the view frustum (rendering of objects (primitives) that are completely occluded)
- improper levels of detail (LOD) (objects that rendered with considerably fewer primitives would have produced the same image)
- scene complexity variation (e.g. the walls of a room require fewer primitives per pixel than a complex piece of furniture).

Numerous efforts have been targeted at eliminating the first two causes. The third cause of primitive clumping cannot be eliminated. An architectural-level solution is to make the screen regions smaller and improve the load balancing by dynamic (or statically-interleaved) assignment of rasterizers to regions [Ellsworth96].

Translated into the IBRW context, primitive clumping is less of an issue, although it cannot be ignored (see section 5.4.1.3). The depth images were hopefully acquired from nearby locations so the LOD is well adjusted and depth complexity is low. If all samples that project in the view-frustum are used, they spread evenly in the warpbuffer. If occlusion culling (extended to discarding redundant samples of coincident surfaces) is used, the total number of samples is low and there will be only a few samples per output pixel. Thus, again, the primitives are likely to be well distributed. Since the sampling resolution is constant throughout the depth image (it doesn’t decrease as simpler surfaces are encountered), one cannot talk about scene-complexity variation since the modeling paradigm does not offer this flexibility to begin with[19].

In conclusion, the main challenge of a sort-middle IBRW architecture is the high-bandwidth required for the interconnection network that delivers the transformed primitives (warped samples) to the rasterizers that interpolate between the warped samples.

5.2.2 Why not sort-last?

Sort-last (*PixelFlow*, [Molnar92]) is appealing since the renderers process sets of primitives, independently computing partial solutions which are composited at a cost linear with the number of renderers. Unlike sort-middle, the communication cost does not depend directly on the number of primitives. However this advantage is irrelevant for IBRW since the total number of primitives is proportional to the number of output image pixels anyway. When super-sampling is used for antialiasing, compositing has to be done before the framebuffer is filtered down to create the output image. Thus, in the case of IBRW with a 2x2 super-sampled warpbuffer, the communication cost is 4 times the number of output pixels times the number of renderers used.

[19] An alternative to IBRW is to first use the depth images to generate a geometric model of the scene and then render the model on polygon-rendering hardware. Such models might group nearly coplanar samples as a single textured polygon.

Pure sort-last architectures instantiate full framebuffers at each renderer so they are less prone to load imbalance. However, single-chip high-resolution super-sampled framebuffers cannot be currently built because of silicon technology limitations, so region-based rendering has to be employed.

5.2.3 Why sort-first?

Sort-first has the great advantage of exploiting the frame to frame coherence associated with typical 3D-graphics applications. The view changes little from one frame to the next, so a renderer that is responsible for a certain screen region needs only a modest percentage of new primitives for the new frame.

To our knowledge no sort-first polygon-renderer has ever been built. Such an architecture was proposed by [Mueller95]. There are serious challenges associated with sort-first in the case of polygon-renderers [Mueller97]. We will next show that these can be overcome in the case of IBRW.

First, one must establish a pre-transformation operation to be executed (by the nodes or host) that is inexpensive and accurately predicts the screen location of primitives. In the case of polygons, such an operation is hard to find. Mueller groups sets of primitives and transforms their bounding box [Mueller00].

In the IBRW case, one can take advantage of the locality of depth images, and inexpensively and accurately predict the screen regions to which a set of samples warp. This is easily done for a rectangular subregion of a depth image (tile) by computing the screen bounding box of a frustum that contains all the depth samples of the subregion. Such a frustum is easy to find, and we use the frustum defined by the four rays that go through the corners of the tile and by the planes of minimum and maximum depth in the tile (**Figure 5.2**). Assuming that the hither plane does not clip the frustum, the bounding box can be easily found by warping the four corners of the tile once with minimum and once with maximum depth, and computing the bounding box of the eight resulting points. If the frustum crosses the hither plane, clipping has to be done first and the same procedure can be applied to the clipped frustum.

An ideal primitive-group pre-transformation operation assigns a certain group only to renderers that need it, that is at least one of the primitives projects inside the screen region of the renderer. The method described deviates from the optimal behavior because:

- the bounding box is the axis-aligned rectangular superset of the warpbuffer locations actually impacted by the frustum projection;
- the frustum is a superset of the depth samples of the tile, as if the tile had, at every location, samples with all possible depths between the tile's minimum and maximum (**Figure 5.2**).

The first problem is not so severe. It could be eliminated by projecting and scan-converting the frustum and choosing only the renderers whose regions are impacted, but the gain is too small to justify the scan-conversion of the 12 triangles.

- The second problem is important, especially for tiles that cover two or more objects at different depths. The resulting frustum is big due to the large depth variation in the tile (**Figure 5.3**). Such tiles can be segmented into several groups of similar depth, one for each object sampled by the tile. In retained mode, when the depth images are available beforehand, the tile segmentation can be done as a preprocess. For immediate mode, when the depth images are acquired in real time, the cost of segmenting the tiles on the fly must be compared with the savings achieved.

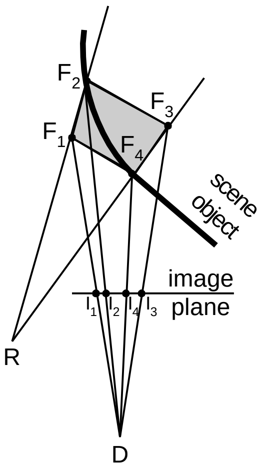


Figure 5.2. Over-conservative estimate of tile projection. A single tile of the reference image R is shown (2D view). The gray area corresponds to the $F_1F_2F_3F_4$ minimum and maximum depth frustum used to approximate the tile. All the samples of the tile will warp inside the desired-image (D) projection of the frustum $I_1I_2I_3I_4$. One can see that the projection contains pixels to which no tile sample will warp (I_1I_2).

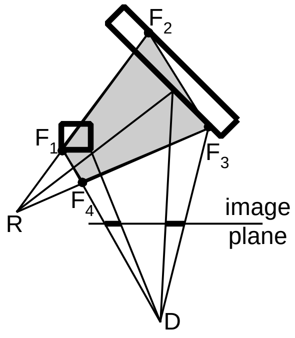


Figure 5.3. Severe overestimation of tile projection. The tile shown spans two objects in the scene (rectangles) that are far apart. The frustum becomes big and its desired-image-plane projection is much larger than the actual projection of the tile (shown with thicker lines).

The tiles used are *16x16* samples in size and, as will become apparent from section 5.4.1.1, the pre-transformation operation is very effective.

Another problem ascribed to sort-first architectures is the difficulty in using hierarchical databases of primitives. Such databases have tree like structures where the internal nodes are modifiers of the objects obtained by aggregating the primitive groups stored at the leaves. Problems arise especially with the replication modifier, when the database needs complex editing.

For the first generation of IBRW hardware we do not target rendering from hierarchical databases because of the problems it introduces.

First, natural-light-source modeling from photographs and image-based-object re-lighting are complex problems not yet solved (research is underway [Yu98] and [Debevec98]). For now we concentrate on prototyping hardware that renders interactively complex natural scenes reusing the lighting originally captured in the photographs.

Second, having all the instances of a replicated object in the image-based representation of a scene has the advantage of providing pre-computed appropriate level of detail for each instance. Also the images provide a partial occlusion-culling solution, which limits the number of samples that have to be warped at a frame. In the case of hierarchical databases this advantage is lost and the number of primitives is unbounded, like in polygon rendering.

We conclude that sort-first is well suited for IBRW when the current frame is rendered from samples of a fixed number of nearby depth images. We devised such an architecture, the *WarpEngine*, which we present next.

5.3 Architecture

We decided to partition the reference images into 16x16-sample *tiles* (with a 15x15 payload) and to use these as the basic rendering primitive. Tiles provide several important advantages:

- we can selectively use portions of reference images as needed for adequate sampling and coverage of visible surfaces (Chapter 6);
- as we have seen in the previous section, one can easily estimate the screen area to which a tile transforms, enabling efficient high-level parallelism;
- tiles are small enough that the same interpolation factor can be used for all samples, enabling SIMD low-level parallelism (section 5.3.2.1).

When designing the WarpEngine, two major problems had to be addressed: computation volume implied by warping and interpolation, and warpbuffer bandwidth.

Warping and interpolation

All the samples of a tile can be warped and interpolated with the same set of instructions so a SIMD implementation is, we believe, the most efficient. We opted for an array of simple byte-wide processors, similar to the one used in PixelFlow [Molnar92]. For a computation that can be efficiently mapped, a SIMD array provides efficient use of silicon, since control is factored out over all the processors. A large array of simple processors is more easily programmable than a complex pipelined processor. The programmability is necessary for use of the WarpEngine as a research tool.

We designed the SIMD array equal in size to the reference-image tile, since the warping calculation is the same for every pixel, with minimal branching required, resulting in a good uitlization of the Processing Elements (PEs). Nearest neighbor PE connectivity provides each PE with access to the three other samples needed for interpolation.

Warpbuffer bandwidth

As stated before, an important design concern was providing sufficient warpbuffer bandwidth. To achieve the enormous warpbuffer bandwidth required (**Table 5.2**), a very large number of commodity DRAMs is required (well over 100); similarly, the warping/interpolation processors would require hundreds of pins dedicated to interfacing with the warpbuffer. By placing the warpbuffer on-chip, that is, on the same ASIC as the processors that generate the warped samples, very wide and fast memory interfaces can be used.

5.3.2 WarpEngine Implementation

Our architecture, the *WarpEngine*, consists of one or more identical *Nodes* (typically 4 to 32); each Node consists of an ASIC and a Tile Cache. The ASIC contains:

- a 16x16 SIMD Warp Array, for warping and interpolating reference-image samples;
- a Region Accumulator, which includes a double-buffered warp buffer for a 128x128 screen region and 4 sample processors for resolving visibility;
- a Reconstruction Buffer, for computing final pixel values;
- a Network Interface, which connects the Nodes together into a high-bandwidth ring, and provides a connection to the host, a connection to each of the Warp Arrays, and a connection to the Tile Cache.

The Tile Cache is a commodity DRAM device; it is used for caching both reference-image tiles and instructions. A double-buffered Frame Buffer receives the final pixel values from the Nodes for display.

The basic operation of the system is as follows (see **Figure 5.4**):

- The host determines which reference-image tiles are to be used to compute the destination image, and computes the screen-space bounding box for each of these tiles. For each screen region, the host maintains a *bin*; each bin contains pointers to the tiles whose bounding boxes intersect that screen region.
- For each screen region, the host assigns a Node to be responsible for that screen region. The host sends each tile in the region's bin to the Node. (Tiles are cached in each node's Tile Cache. If a tile is resident in one of the caches, the host instructs the Network Interface to forward it to the appropriate Node. If not, the host must send the tile data to the Node).
- Each tile received by each Node is loaded into the Warp Array, which performs the warping and interpolation calculations for the tile, and forwards the warped samples to the Region Accumulator.
- The Region Accumulator collects the warped samples into its sub-pixel resolution warp buffer.
- After all tiles in the region's bin have been processed, the Region Accumulator swaps its buffers and initializes the visibility buffer, in preparation for processing the next screen region.
- Concurrently with processing the next screen region, the Region Accumulator steals memory cycles to send the previous region's data to the Reconstruction Buffer. The Reconstruction Buffer computes the final pixel values for the region and forwards them to the Frame Buffer.

- After all regions have been processed and the final pixel values calculated and forwarded to the Frame Buffer, the Frame Buffer swaps buffers.
- The system can function in *retained* mode, in which there is a fixed set of reference images describing an environment, or *immediate* mode, in which new reference images are being received “on the fly”.

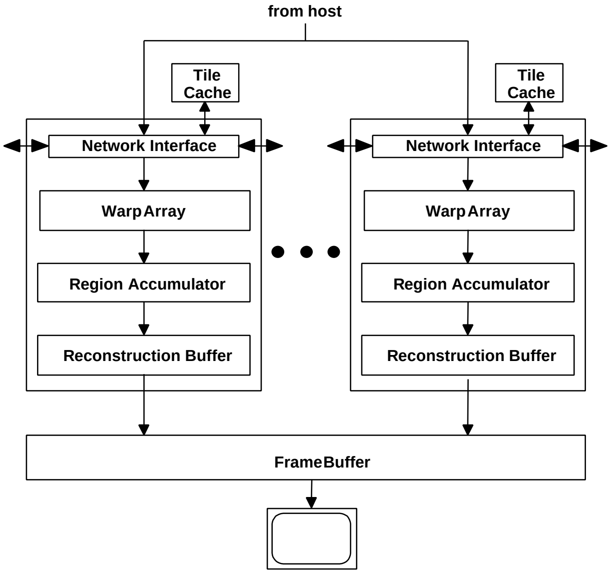


Figure 5.4. Block-diagram of the WarpEngine

5.3.2.1 WarpArray

The Warp Array (see **Figure 5.5**) consists of 256 processing elements (PEs), arranged as a 16x16-pixel array. Each PE consists of a simple byte-wide ALU and 160 bytes of local memory partitioned as: 128 bytes main memory, 16 bytes IO Buffer, 16 bytes Sample Buffer.

A distributed linear expression evaluator provides values of the linear expression Ax+By+C to each PE simultaneously, in byte-serial form (*x* and *y* represent the position of the PE in the 16x16 array). It is used for very fast computation of the linear part of the numerator and denominator of the warp-equation expressions (see **Equation 1.1**). Each PE includes a byte-wide connection to its neighbor in each dimension. Clock rate for the PE and local memory will be 300 MHz or more.

The IO Buffer is used for inputting reference-image tiles (from the Tile Cache or host, via the Network Interface) via a 300 MByte/sec interface. The Sample Buffer is used for exporting warped samples to the Region Accumulator, over the sample port, via an on-chip 4.8 GigaByte/sec interface. Access to these buffers may occur simultaneously with accessing of the main memory by the ALU, so that the next tile may be loaded during processing of the current one, and one set of interpolated samples can be computed while the previous set is being output to the Region Accumulator.

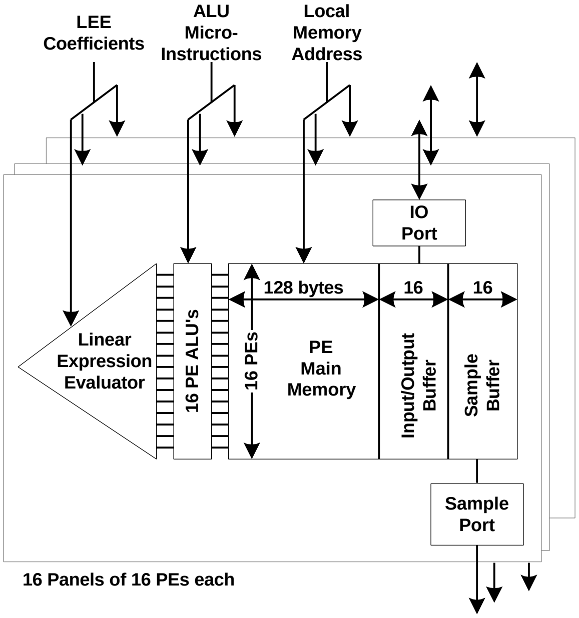


Figure 5.5. Block-diagram of the Warp Array

5.3.2.2 Region Accumulator

The Region Accumulator (**Figure 5.6**) consists of a large SRAM warp buffer (the Region Buffer) and a set of 4 Sample Processors, which combine warped samples into Region Buffer memory.

The Region Buffer contains data for a 128x128-screen region, at 2x2 sub-pixel resolution; a half-pixel wide boundary is added, to allow reconstruction kernels up to two pixels wide. The Region Buffer is partitioned into 4 sections, interleaved 2x2 across the sub-pixel grid.

Each word of Region Buffer memory is divided into three fields. Two double-buffered fields (the RGB/Offset/Present fields) include RGB values, the offsets used for reconstruction, and a *present* bit (used to avoid z-buffer initialization). One buffer is used for accumulating samples for the current region, while the other buffer contains the previous region's values for output.

The third field contains values that are not required for reconstruction and need not be double-buffered. Besides z value, we are reserving space for measures such as the quality of each sample [Mark99]. If the z of two samples are similar, the sample processor gives preference to the better sample. The quality of the sample is derived differently according to the scene. In the context of imperfect registration characteristic of our (and probably all) current depth-image acquisition devices, we obtained better results when we consistently chose the samples of *one* sampling location and used the additional samples from other images just to fill in holes. Synthetic data can achieve perfect registration and the quality of the samples was derived from the interpolation factor of the tile it belonged to: the closer the interpolation factor was to 2x2, the higher the quality[20].

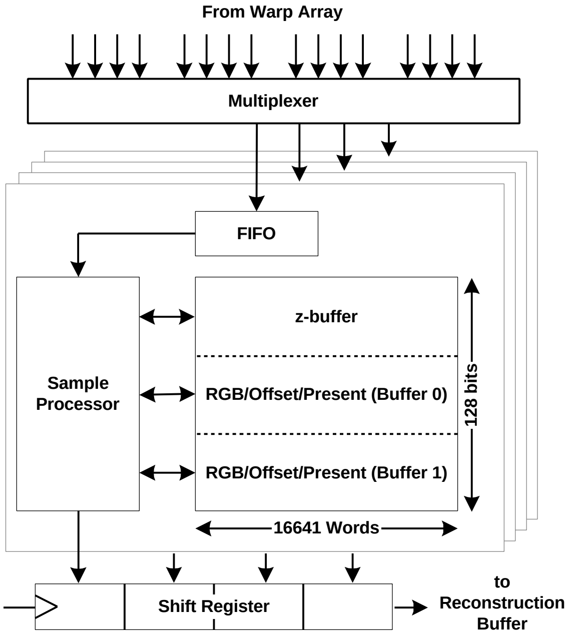


Figure 5.6. Block-diagram of the Region Accumulator

A 128-bit wide memory interface provides read/write access to all three buffers in parallel. Each Sample Processor processes a sample every two clock cycles; this is the maximum possible rate, since 2 Region Buffer accesses (1 read and 1 write) are required for each sample. The Sample Processor is pipelined, so that each computation has several cycles to execute, while sustaining the rate of a sample every two clock cycles. Thus 4 Sample Processors handle an aggregate rate of 2 samples per clock cycle, or 600 million samples/sec at 300 MHz.

The back buffer sends samples from the previous region to the Reconstruction Buffer, via a shift path that spans all 4 partitions of the Region Accumulator. A small fraction of memory cycles are stolen from the Sample Processors, to feed this scan-out path.

5.3.2.3 Reconstruction Buffer

The Reconstruction Buffer accepts the stream of final warped subsample values from the Region Accumulator, and filters them to produce final pixel values for the *128x128* pixel region. The Reconstruction Buffer includes two scan-line-sized accumulators, and four simple processors. For each RGB/Offset/Present value, each color component is multiplied by a weight from the filtering kernel and added to a sum. Normalization by the sum of weights produces the final pixel value, which is output from

[20] A 2x2 interpolation factor implies destination image sampling close to reference image sampling, which is desirable.

the ASIC to the Frame Buffer. The filter kernel is 2x2 pixels in size, with 4x4 sub-pixel resolution. The two 2-bit offset values select the proper kernel element within each sub-pixel.

5.3.2.4 Frame Buffer

The Frame Buffer is a straightforward assembly of commodity DRAMs and programmable parts. It must absorb the full bandwidth of the Reconstruction Buffers on all Nodes, so the peak output rate of the Nodes must be tuned to avoid over-running the Frame Buffer.

5.4 Performance

We ran experiments on three scenes:

- *Eurotown*, a complex model of a city; the depth images are placed on a regular grid and were rendered with a polygon-renderer (see **Figure 5.7**);
- *Kamov helicopter;* eight depth images also generated from a polygonal model are placed around the helicopter and one samples it from above (see **Figure 5.8**);
- *Reading Room;* two depth panoramas of the reading room of our department were acquired with the *DeltaSphere* [Nyland99], our in-house laser rangefinder (see **Figure 1.7**).

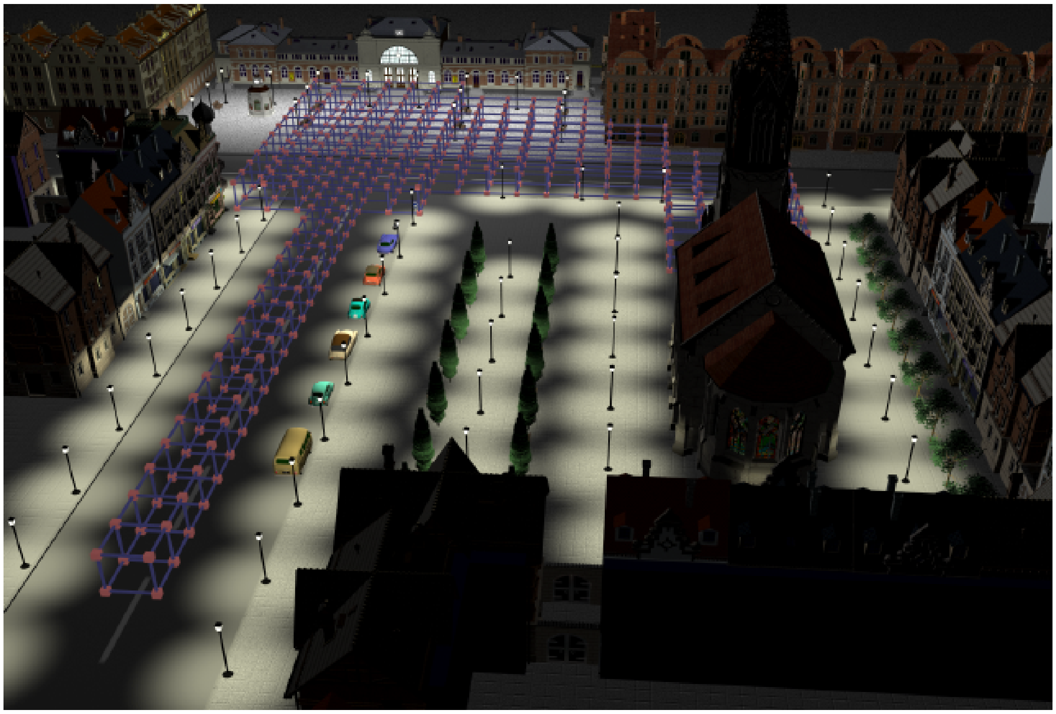


Figure 5.7. Overview of the Eurotown scene. The little solid (pink) cubes indicate the sampling locations from which the depth and color panoramas were acquired. A panorama consists of 6 depth images with 90 degree horizontal and vertical fields of view that correspond to the faces of a cube. The depth images are 1K x 1K in the case of VGA output (70 deg horizontal field of view) and 2.8K x 2.8K in the case of HDTV (70 deg FOV). The total space consumed by all of the depth images is 4.33GB and 26.06GB respectively, with lossless RLA compression. The wire-frame defines the cells that subdivide the viewing volume.

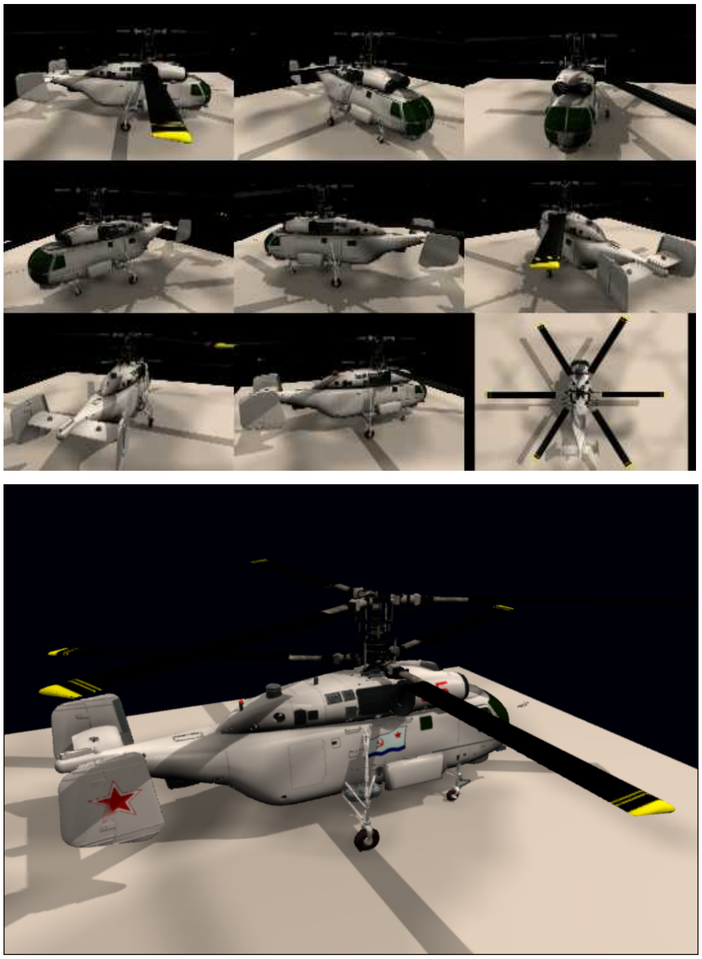


Figure 5.8. Kamov test scene. The top image shows the nine depth-images used to model the Kamov helicopter. The bottom image was rendered on the WarpEngine simulator.

We measured the efficiency, communication costs, and load balancing for various host-software modes:

- VFC: tile selection based on view-frustum culling only; all tiles in the view-frustum are sorted and sent to the appropriate node;
- VFCTS: tile selection based on view-frustum culling and tile segmentation: tiles with depth discontinuities were divided into up to eight segments; view-frustum culling and sorting was done at the tile-segment level;
- OC: tile selection based on view-frustum and occlusion culling, with tile segmentation. This is our most elaborate host tile-selection algorithm and it is presented in detail in Chapter 6. The tiles are approximated by quads that are rendered into a low-resolution buffer for occlusion culling. The coincident tiles (same surface sampled in more than one depth image) are arbitrated according to the sampling quality: the tiles that have the sampling rate closest to the sampling rate required by the desired image are chosen.

5.4.1.1 Efficiency

An important measure of the performance of the parallelization scheme is efficiency, that is, the ratio between the useful and total work. For the *WarpEngine* this translates into the *overlap factor*, which is defined as the total number of tiles in the region bins over the number of distinct tiles. In other words, the overlap factor indicates to how many screen regions, on average, a tile maps to. The average frame overlap factors for the various scenes and various tile selection methods are given in **Table 5.3**

Scene	Tile-selection method	Selected Tiles	Overlap	Error (%)
Eurotown	VFC	17056	1.330	8.16
	VFCTS	17022	1.283	4.92
	OC	4224	1.341	5.60
Kamov	VFC	9121	2.650	50.50
	VFCTS	9108	1.505	12.81
	OC	2737	1.82	22.88
Reading room	VFC	6246	1.219	5.73
	VFCTS	6245	1.218	5.63
	OC	4756	1.237	5.68

Table 5.3 Efficiency of sort-first parallelization scheme for various test scenes

The screen resolution is 720x486, and the screen region size is 128x128 pixels, which is equivalent to 256x256 warpbuffer locations. The tiles are 16x16 reference-image samples in size.

The selected-tiles column gives the average number of tiles that are selected for a frame (before bucket sorting). The number of selected tiles decreases slightly when segmentation is introduced since some tiles conservatively ruled as visible by the VFC host-mode are correctly detected as invisible by the VFCTS host-mode. This happens when a whole tile frustum is visible but none of the tile-segment frusta are.

The number of selected tiles decreases substantially when occlusion culling is added. The overlap is reduced by tile segmentation. The slight increase with occlusion culling is due to the fact that the average screen-projection size of the tile increases. (Tiles that sample too densely are eliminated in favor of tiles that better match the desired-image sampling-rate.)

The last column gives the approximation error of the pre-transformation operation used to bucket sort the tiles; it is computed as the percentage of tiles that are unnecessarily rendered at a region (from the total number of tiles rendered). In our simulator such an error is detected when none of the samples of a tile warps to the region to which the tile is allocated. In the case of the helicopter with no tile segmentation the error is quite important, due to the tiles of the top image that have large screen projections. When tile segmentation is used, the error is not very important and it doesn't justify a more expensive pre-transformation operation.

The overlap does not depend on the number of nodes, it depends only on the size of the regions. How does the overlap vary with the output resolution? If the average screen-projection size of the tile and the screen-region size remain the same, the overlap factor should not change. Out of sampling considerations, the tile screen-projection size *has to* remain the same. Increasing the resolution of the output without increasing the resolution of the input accordingly just increases the amount of interpolation between the samples, which, of course, produces unsatisfactory results. In all our experiments the resolution of the reference images was the same as the resolution of the output image. **Table 5.4**[21] shows the invariance of the overlap factor with respect to output resolution (*Eurotown* scene).

Res	**Tile-selection method**	**Selected Tiles**	**Tiles / pixel**	**Overlap**	**Error (%)**
VGA **720 x 486**	VFC	14031	0.0400	1.318	7.00
	VFCTS	14002	0.0400	1.283	4.94
	OC	4243	0.0121	1.324	5.53
XVGA **1280 x 1024**	VFC	49890	0.0380	1.345	7.04
	VFCTS	49872	0.0380	1.316	4.97
	OC	12619	0.0096	1.334	5.96
HDTV **2000 x 1000**	VFC	74679	0.0373	1.431	11.56
	VFCTS	74603	0.0373	1.364	7.14
	OC	21696	0.0109	1.377	7.94

Table 5.4 Efficiency of sort-first parallelization scheme for various output resolutions

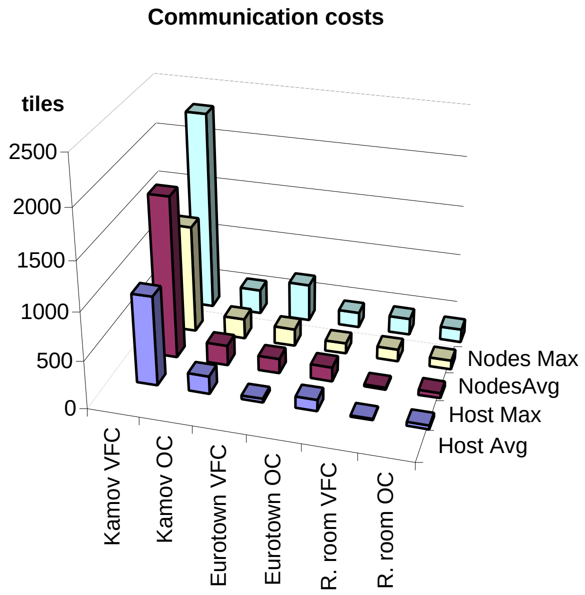
The number of tiles increases close to linearly with the screen area, as can be seen from the resolution-invariant tiles/pixel figure. We conclude that the overlap factors are small, which makes sort-first an efficient parallelization scheme for IBRW, capable of substantial speed-ups.

5.4.1.2 Communication costs

Another crucial factor in determining the performance of the parallelization scheme is the amount of communication required, both between the host and the *WarpEngine*, and between the nodes of the *WarpEngine*. Before a node can render a tile it must have it in its local Tile Cache. The node can get a

[21] The VGA numbers are slightly different than in the previous table since the path used for this table is different (shorter)

missing tile either from another node, or, if no node has it, from the host. **Graph 5.1** shows the per-frame average and maximum communication requirements measured in our experiments.

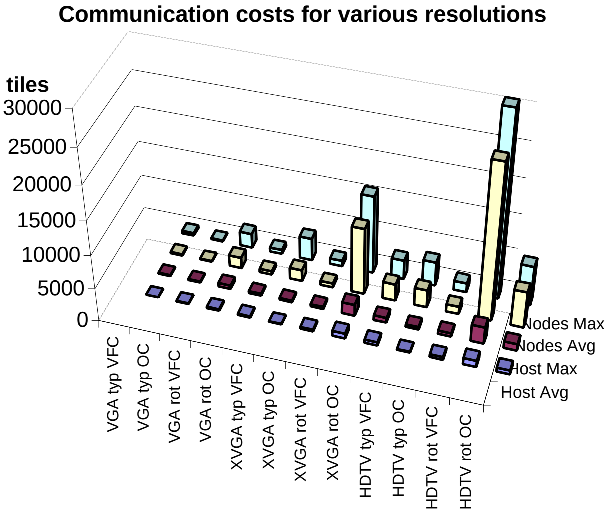


Graph 5.1. Per-frame maximum host- and node-to-node- tile traffic.

The output resolution is VGA and the *WarpEngine* is configured with 4 nodes to which the 6x4 regions are statically assigned in an interleaved pattern. The size of a tile is 256 samples, which at 8 bytes per sample totals 2KB. The tile-caches will probably be implemented with 256-Mbit SDRAM chips, so each can store up to 16 Ktiles.

In computing the figures above it was assumed that a tile is discarded from the Tile Cache if not used for a frame. This is over-conservative since Tile Caches are large enough to store tiles for several frames, which is useful when the path re-explores parts of the scene. Even so the communication volumes are manageable. A high-end PC can send at peak rate, assuming 30Hz frame rate, at most 8 Ktiles per frame through an AGP2x 533 MB/s interface. The Kamov scene requires the most communication when no occlusion culling is used since the tiles clump in the region to which the helicopter maps.

For *Eurotown*, the numbers given do not take into account the transitions from one cell to the other, when about half of the tiles are replaced with new tiles since 4 out of the 8 sampling locations are new. For such cases we found that about 15 Ktiles must be sent from the host in VFC mode and 6 Ktiles in OC mode. The projected AGP4x 1GB/s interface might provide enough host bandwidth even for these cases. For higher output resolutions the host will have to predict the new cell and amortize the high host-bandwidth requirement of cell transition over several frames. Of course this limits the translational speed of the camera for a certain cell size. Higher speeds can be supported by increasing the size of the cell accordingly. High rotational speeds of the camera also increase the communication requirements. The following graph shows the per frame communication costs for various resolutions for a typical path (*typ* in the graph) and also for the case of a 90 degrees / second rotation (*rot*).



Graph 5.2. Maximum per frame communication volumes for various resolutions

The host bandwidth required can be supplied by the AGP interface, as all maximum host-to-*WarpEngine* communication volumes are below 4 Ktiles. The Network Interface that inter-connects the nodes is not designed yet; from experience we are confident that ring bandwidths of 8 to 64 Gb/s are attainable. At 60Hz this translates into 8 to 64 Ktiles / frame which from the above graph appears to be sufficient for the highest node-to-node traffic.

A potential bottleneck is the bandwidth in and out of the Tile Caches. Using 16-pin 133 MHz memory chips for the Tile Caches provides 266 MB/s, which at 30 Hz (60 Hz) means about 2 (1) Ktiles / frame if half of the bandwidth is reserved for the instructions for the SIMD *Warp Array*. For the worst case in the graph (HDTV rot VFC), assuming a full-blown system with 32 nodes, the maximum number of new tiles a node needs at any given frame is 8.6Ktiles. Conservatively increasing the figure by a factor of two, since a node has to provide tiles to other nodes, we obtain about 16 Ktiles/frame maximum Tile Cache accesses, which is considerably more than the 2 Ktiles computed at 30 Hz. However, as will become apparent from the next subsection, in the VFC host mode only 8 fps can be achieved, reducing the mismatch considerably. Conversely, 30 fps are achieved in the OC host mode when there are considerably fewer tiles, thus fewer Tile Cache accesses. We will investigate the Tile Cache access bottleneck further. An attractive solution is to use 2 or 4 memory chips for each Tile Cache, which, besides providing the required bandwidth, also has the advantage of providing more storage room. The trade-off is a higher number of pins for the rendering-node chip.

The host main-memory accesses are not a bottleneck since the host communication volume is much less important.

5.4.1.3 Load balancing

Finally we investigated the load balancing of the *WarpEngine* architecture. **Table 5.5** (VGA output resolution) shows the ratio between the number of tiles assigned to the busiest node and the least busy node. The other measure of imbalance given is the ratio between the time it takes the busiest node to finish rendering and the time required by the least busy node. The performance of the architecture is determined by the performance of the Warp Array, which warps and interpolates between the warped samples. Processing a tile takes approximately 2000 cycles for warping and *n* * 256 cycles for outputting the *n*-per-sample interpolated sub-samples.

Scene	**Tile Selection Method**	**Nodes**	**Load Imbalance**		**Sustained fps**
			tiles	**time**	
Eurotown	VFC	4	1.315	1.358	12
		9	3.09	2.52	20
	OC	4	1.36	1.15	47
		9	2.25	2.34	80
Kamov	VFC	4	1.83	1.89	2
		9	5.36	5.32	2
	OC	4	1.22	1.36	7
		9	1.76	3.86	12
		24	8.16	10.62	19
Reading room	VFC	4	2.02	1.90	26
		9	2.85	3.09	47
	OC	4	1.68	1.62	35
		9	2.34	3.62	70

Table 5.5 Load balancing study for VGA output resolution.

The sustained frame rate is computed by finding the busiest renderer at any of the frames of the animations. The Warp Array clock-rate is 300 MHz. Load balancing is acceptable in the case of the *Eurotown* and *Reading Room* scenes. The two imbalance figures are close since the tiles are interpolated with similar interpolation factors. The regions do not divide evenly among the nodes, and the extra region assigned to some of the nodes causes imbalance, especially in the case of higher number of nodes. The imbalance is moderate and a higher number of nodes provides very high refresh rates.

For the Kamov scene some regions, thus some nodes, are assigned a considerably higher number of tiles, and the interpolation factors are also higher, due to the way the reference images are placed. Even if each of the 24 regions has its own node and even if the most efficient tile selection method is used, the sustained frame rate is below 30Hz. Modeling the scene with regularly placed depth images has the advantage we anticipated of better load distribution.

Table 5.6 presents the load balancing measurements for higher resolutions.

Resolution	Tile Selection Method	Nodes	Load Imbalance		Sustained framerate (fps)
			tiles	time	
XVGA	VFC	16	2.33	2.75	9
		30	7.34	8.34	14
	OC	16	2.73	3.24	42
		30	6.35	5.85	65
HDTV	VFC	16	1.50	1.83	6
		32	1.76	2.75	8
	OC	16	2.18	1.96	23
		32	2.76	4.68	29

Table 5.6 Load balancing for the Eurotown model at XVGA and HDTV output-resolutions.

A 32-node *WarpEngine* system is quite potent and can render in OC mode at high resolutions at sustained interactive rates.

For systems with many nodes, load imbalance becomes an issue. For the HDTV case, if each region is assigned a node (a hypothetical 128 node-system), the sustained frame rate is 39 fps for the OC case. This is the upper bound on the refresh rate achievable by improving the regions-to-nodes assignment. The static allocation has the advantage of low node-to-node communication since it takes full advantage of the frame-to-frame coherence. Once regions are assigned dynamically to the nodes, one has to keep the node-to-node communication under control. We experimented with a greedy allocation scheme that assigned iteratively the hottest region to the least busy node. When the greedy allocation scheme was utilized at every frame for the *Eurotown* HDTV OC case, we obtained an almost perfect 1.031 worst rendering-time load-imbalancing with sustained 30 fps refresh rate for a 16-node system. A 32-node system rendered at 38 fps with a time imbalance of 1.689.

Since the allocation tables differed substantially from one frame to the next, the volume of node-to-node communication was quite high (maximum 26,594 tiles per frame), but it did not exceed the projected capacity of the ring network. One could employ quasi-static allocation schemes that exploit the frame to frame coherence and change the node allocation of a region only rarely. Such schemes seem practical especially since tiles are not used for a very long time as the camera translates to another cell and fresh tiles are downloaded from the host.

To attain even higher framerates, one has to use smaller regions in order to reduce the maximum rendering time spent at a region. A *WarpEngine* node needs about 4 Kcycles to process a tile (assuming 8 subsamples per sample), thus the 22 Ktiles per frame for the OC HDTV *Eurotown* case could be rendered theoretically (ideal parallelism) 110 times a second by 32 nodes running at 300MHz. Thus there is room for improvement. The overlap factors computed for the 128x128 regions are small, which makes us believe that 128x64 regions will still yield reasonable overlap factors. For the *WarpEngine* architecture, the price of smaller regions is mainly wasting some of the warpbuffer memory of the *Sample Processors*; it does not imply processor underutilization. Moreover it is not necessary to use smaller regions for the entire screen: one could subdivide only the bottleneck regions.

5.5 Conclusions

Sort-first is a very attractive parallelization scheme for IBRW. The regular structure of depth images makes it possible to estimate accurately and inexpensively the screen projections of a set of depth samples.

Some of the difficulties associated with rendering from polygons are pushed upstream to the modeling stage. Partial solutions for occlusion culling and level-of-detail adaptation are computed inherently as the depth images are acquired and they are successfully used for an entire viewing-volume subdivision cell. The *WarpEngine* IBRW sort-first architecture is fairly simple (one ASIC design) but promises high refresh rates.

The load balancing and thus the performance can be improved by perfecting the regions-to-node allocation scheme and by splitting the regions into smaller regions where / when necessary. The additional node-to-node traffic generated by occasional region-to-node reallocations is small and can be easily managed by a high-capacity ring network.

For high output resolutions, more sophisticated tile-selection host algorithms must be employed rather than just choosing all the tiles in the view frustum. The occlusion-culling partial solution provided by the images acquired from the corners of the current cell must be further refined to reduce the number of tiles. I developed such an algorithm and it is presented in detail in the next chapter.

Chapter 6 Tile Choosing with the Vacuum-Buffer Algorithm

Modeling a scene with depth images lets one automatically capture intricate details hard to model conventionally. Also, rendering from such representations has the potential of being efficient since it seems that the number of samples that must be warped is independent of the scene complexity and is just a fraction above the number of samples (pixels) in the final image. However, selecting the subset of reference-image samples that must be warped to generate the new view is a very difficult task. When an input (reference) image is warped, surfaces that were originally hidden can become disoccluded due to motion parallax and gaps form in the warped image (disocclusion errors). The gaps can only be filled with samples from other reference images. Also, the sampling of surfaces from reference to output (desired) image changes differently for every surface.

I developed an algorithm, which I call the *vacuum-buffer* algorithm. This chapter presents the use of the vacuum-buffer algorithm as part of a new sample-selection technique, although it potentially has also other applications, as indicated in the future work section 8.2.2. Like other techniques, my sample-selection method proceeds by considering samples of reference images that were acquired from locations close to the current camera position. Unlike other techniques however, given a set of reference images and a desired view, the vacuum buffer is able to conservatively decide whether the reference images could be missing a surface visible from the desired view. Moreover, the algorithm also points to the scene locations where such surfaces might be. The vacuum buffer is a generalized z-buffer and it stores the sub-volumes of the current view-frustum for which the reference images considered so far do not provide any information.

Another important difference is that the vacuum-buffer method uses the current view, which allows it to reduce the number of samples chosen more than other methods that offer a sample-selection solution to be used for several desired camera views. The tradeoff for using the current view is having to solve the sample-selection problem at each frame. By exploiting the coherence in the reference images, groups of nearby samples become the actual primitive for sample selection, which substantially reduces the total cost. Unlike previous methods (see section 6.1), no intermediate scene representation is necessary, which avoids the undesirable additional resampling and is also better suited for use in conjunction with IBRW hardware. Here is a brief review of previous sample-selection techniques for IBRW.

6.1 Related work

One simple solution is to just warp all samples of reference images that were taken from locations near the desired camera position. Complicated scenes, which require dense sampling with reference images, generate too many samples with this approach. Layered Depth Images (LDIs) bring substantial improvement [Shade98]. LDIs generalize the concept of a depth image by allowing for more than one sample along a ray. Consequently, an LDI can store samples of surfaces that are hidden from the view of the LDI. As the view changes, the originally hidden samples become visible avoiding disocclusion errors. Since there are only few samples in the deeper layers, the total number of samples in an LDI is only marginally larger than the number of samples in an equivalent depth image. The LDIs are constructed as a preprocess by warping reference images to the view of the LDI and discarding samples that warp to the same location and at the same depth.

An important question is what reference images must be combined in an LDI in order to completely eliminate disocclusion errors? One approach is to combine as many as possible regularly spaced reference images, hoping that all potentially visible surfaces are sampled in at least one of the reference images ([Shade98], [Popescu98]). Such an approach can evidently miss surfaces.

Another concern is that an LDI offers only one sampling rate for a particular surface[22], which has to be adapted to the desired-image sampling-rate at rendering time. Chang addresses this problem by using a tree of LDIs [Chang99], choosing the appropriate resolution level according to the desired-image sampling requirement for each surface. Also, since LDIs are used to recreate several views, an LDI will inherently contain samples that are hidden for a particular view, and thus are unnecessarily warped. Another important motivation in looking for a new sample-selection technique is that LDIs are complicated structures that cannot be easily warped in hardware.

I devised a sample-selection technique based on the vacuum-buffer algorithm that conservatively decides whether a set of reference-image samples is sufficient for a particular desired view. If not, the algorithm will indicate where in the scene surfaces might have been missed. The next section presents the vacuum-buffer algorithm in detail and section 6.3 describes its use at choosing reference-image samples to adequately reconstruct the desired view.

6.2 The Vacuum-Buffer Algorithm

6.2.1 Overview

The main idea behind the vacuum-buffer algorithm is to determine the subvolumes of the desired view frustum that could contain visible surfaces that were missed by the current set of reference images. I call these undetermined subvolumes *vacuum*.

The vacuum-buffer algorithm makes use of information contained in depth images that is usually ignored. The depth to the sample is used to warp (reproject) the sample to the output image but it also tells us the distance to the *first surface* in the reference image. Consequently one knows that there are no other occluders between the center of projection of the reference image and the surfaces sampled.

Figure 6.1 shows a reference image with center of projection R used to reconstruct the desired image with center of projection D. The scene is shown by the surface $A_0A_1 \ldots A_5$. The area (volume in 3D)

[22] Moreover it is a *resampling* of the surface, which always introduces additional errors.

shown in light gray (blue) is determined as being free of occluders by the reference image R. For simplicity I call it *air*, since in most scenes it indeed corresponds to air. The desired image D sees part of the air seen by the reference image R and that subvolume of the desired image is determined as empty.

The A_1A_2 segment of the scene is a connected opaque surface (occluder) and although other surfaces might be located behind it as seen from D, such surfaces cannot affect the desired image since they are hidden. The "shadow" that is cast in the desired-image view-frustum by occluders define a *don't care* subvolume. For the purpose at hand, don't care subvolumes are equivalent to air volumes.

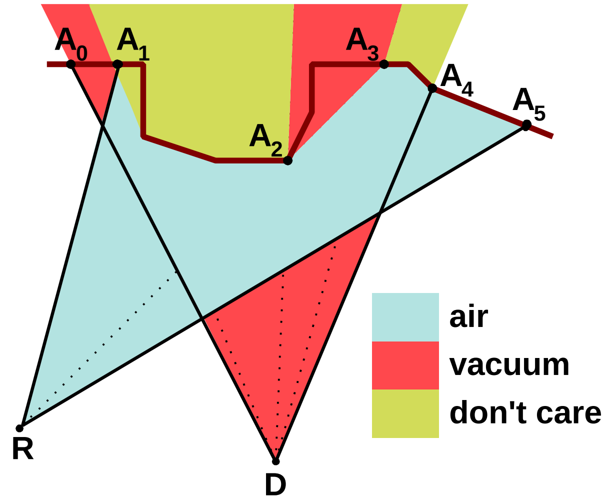


Figure 6.1. *Air* and *vacuum* concepts. The reference image (R) is used to determine the volume of the view frustum of the desired image (D)

The *vacuum* subvolumes shown in dark gray (red) could contain surfaces that are not sampled in the reference image R and are visible in the desired image D. Vacuum is not a guarantee that visible surfaces were missed. In **Figure 6.1** for example, the vacuum zone close to D might resolve to air when an appropriate reference image that encompasses it is used. However vacuum zones *might* contain visible surfaces and one has to resolve them.

Considering the case presented in **Figure 6.1**, one could naively imagine that the problem of missing samples could be detected by searching the frame buffer for pixels that are not set. Indeed, the framebuffer will contain a gap between the new positions of A_2 and A_3 but missing samples can occur even when the framebuffer is instantiated as seen in the simple case presented in **Figure 6.2**.

The next subsection presents the algorithm in more detail.

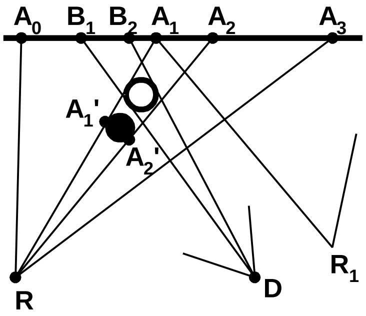


Figure 6.2. Difficult-to-detect disocclusion error case. The reference image R samples the surfaces $A_0A_1A_1'A_2'A_2A_3$. It does not sample the hollow sphere. When used to reconstruct the desired image D, the hollow sphere projects to B_1B_2 and the corresponding pixels are already instantiated with a fragment of A_0A_1. Choosing another reference image to fill the gap between A_1 and A_2 might not suffice to reveal the missing object as is the case of reference image R_1.

6.2.2 Algorithm Description

Given a set of reference images and a desired view, the algorithm computes the amount and the location of vacuum that remains after all reference images are used. Initially, the entire view frustum of the desired image is undetermined, thus filled with vacuum. The air and the occluders of each reference image are used to resolve vacuum by intersecting the vacuum with the air and with the shadows of the occluders. The volume intersections are computed efficiently using a generalized z-buffer, which I call the *vacuum buffer*. The vacuum buffer is similar to the z-buffer but it stores z intervals, or *spans*, along each ray, whereas the classic z buffer stores a single value[23]. The list of z spans at one vacuum buffer location corresponds to the vacuum remaining in the view frustum along that particular ray.

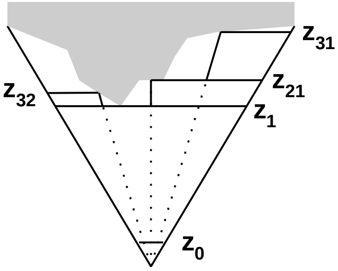


Figure 6.3. 2D view of the quadtree subdivision of the reference image frustum. Only three levels are shown. At each subdivision one of the children will have the same closest z as its parent so three new frusta are created with each subdivision (one in the 2D view shown).

For each reference image, the algorithm first processes the air of the current reference image. As a preprocess, the reference image is recursively subdivided in quadtree fashion and the closest z values are precomputed for each subregion. This subdivides the reference image frustum in subfrusta as seen in

[23] The method of intersecting volumes using rasterization buffers was first used in constructive solid geometry ([vanHook86], [Salesin90] and others).

Figure 6.3. The first frustum is defined by the hither plane and the plane of closest z. The next level frusta are defined by the parent-region's closest z plane and each subregion's closest z plane.

The vacuum-buffer algorithm processes the frusta recursively, starting from the root of the quadtree. All six quadrilateral faces of the frustum are transformed, projected to the desired image plane and scan-converted into the vacuum buffer. Since the frustum is a convex polyhedron, a vacuum buffer location will be either hit twice or not hit at all. When a vacuum buffer location is hit by two samples of the faces of the air frustum, the list of intervals is updated by eliminating the vacuum according to the span of air between the two samples (see **Figure 6.4**).

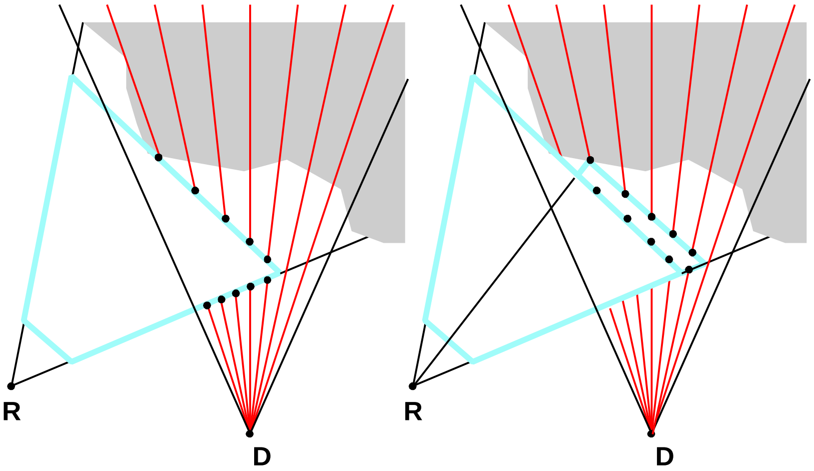


Figure 6.4. Illustration of the recursive vacuum-buffer algorithm. Initially, each vacuum buffer location contains the span (*hither*, *yon*) since nothing has been determined. The figure on the left shows the vacuum buffer after the first level frustum of the reference image R has been processed. Several locations contain two spans since part of the vacuum has been determined. The right figure shows the vacuum buffer after the second level frustum has been processed.

The next important question is when to stop the recursive subdivision of the reference image frustum. Lower levels of the recursion eliminate little vacuum while increasing exponentially in cost. On the other hand, stopping the recursion early doesn't use all the information available in the reference image; some of the air information is wasted.

There is another consideration crucial for deciding what the smallest subregion (tile) of the reference image should be. Remember that the second step of the algorithm is to ignore all vacuum behind occluders. If the tiles are small enough, they are, in general, part of the same occluder and one can ignore everything behind the desired image projection of the tile. However, computing the exact desired image projection of the tiles is expensive and it is equivalent to warping the tile. This obviously defeats the purpose when the vacuum-buffer algorithm is used to choose the tiles needed for the current frame. In order to compute the desired image projection of tiles efficiently, the height field corresponding to the tile is replaced with a single quad. For surface continuity, one chooses the four corner samples that are connected by two triangles; note that in general the four samples are not coplanar but there is always a way of selecting the two triangles such that the resulting polyhedron is convex. All vacuum behind the projection of the quad is eliminated (see **Figure 6.5**).

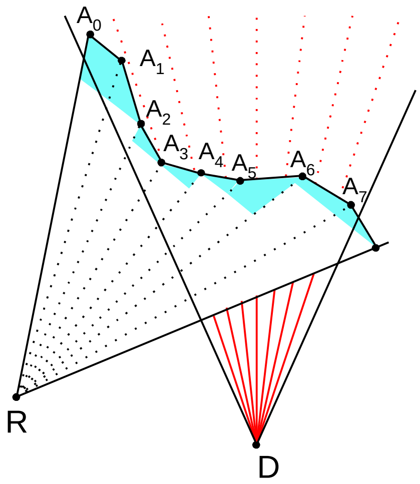


Figure 6.5. Vacuum-buffer algorithm: leafs of recursion tree. The reference image is split into eight tiles. The leaves of the quadtree are frusta defined by the closest-z plane of the previous level and the quads (here A_kA_{k+1} segments) that approximate the height field of the tile. The leaf frusta, shown in gray (blue), are processed similarly to the other frusta. The occluding faces of the frusta are used to eliminate all vacuum behind them (the dotted rays in D's frustum indicate the eliminated vacuum).

If the tiles are too large, then the approximation is too coarse and one can incorrectly eliminate vacuum where unsampled surfaces could potentially hide. I found that *16x16* sample tiles are adequate for a variety of test scenes. Moreover, as seen in the previous chapter, *16x16* tiles are rendering primitives favorable to efficient IBRW architectures.

No matter how small the rectangular tiles are, there will be some tiles that stretch from one object to the other. Such silhouette tiles are not patches of a single occluder and cannot be assumed to be a continuous surface, thus the vacuum behind them cannot be eliminated. These tiles are found in a preprocess using the depth discontinuities in the reference image (see section 4.1.3). Tiles with depth discontinuities are still useful for the first phase of the algorithm when the air of reference images is used to resolve vacuum from the vacuum buffer. At the second step, depth discontinuity tiles can be ignored, relying on tiles from other reference images to sample the two occluders independently. However, for small features like the light poles in **Figure 6.6**, it is very likely that no reference image will have tiles that map entirely on the light pole. Thus one should segment the tile according to the surfaces sampled (see **Figure 6.6**). The resulting tile segments will each model one object now and they can be treated as regular tiles. Note that tile segmentation does not depend on the desired view so it can be done as a preprocess.

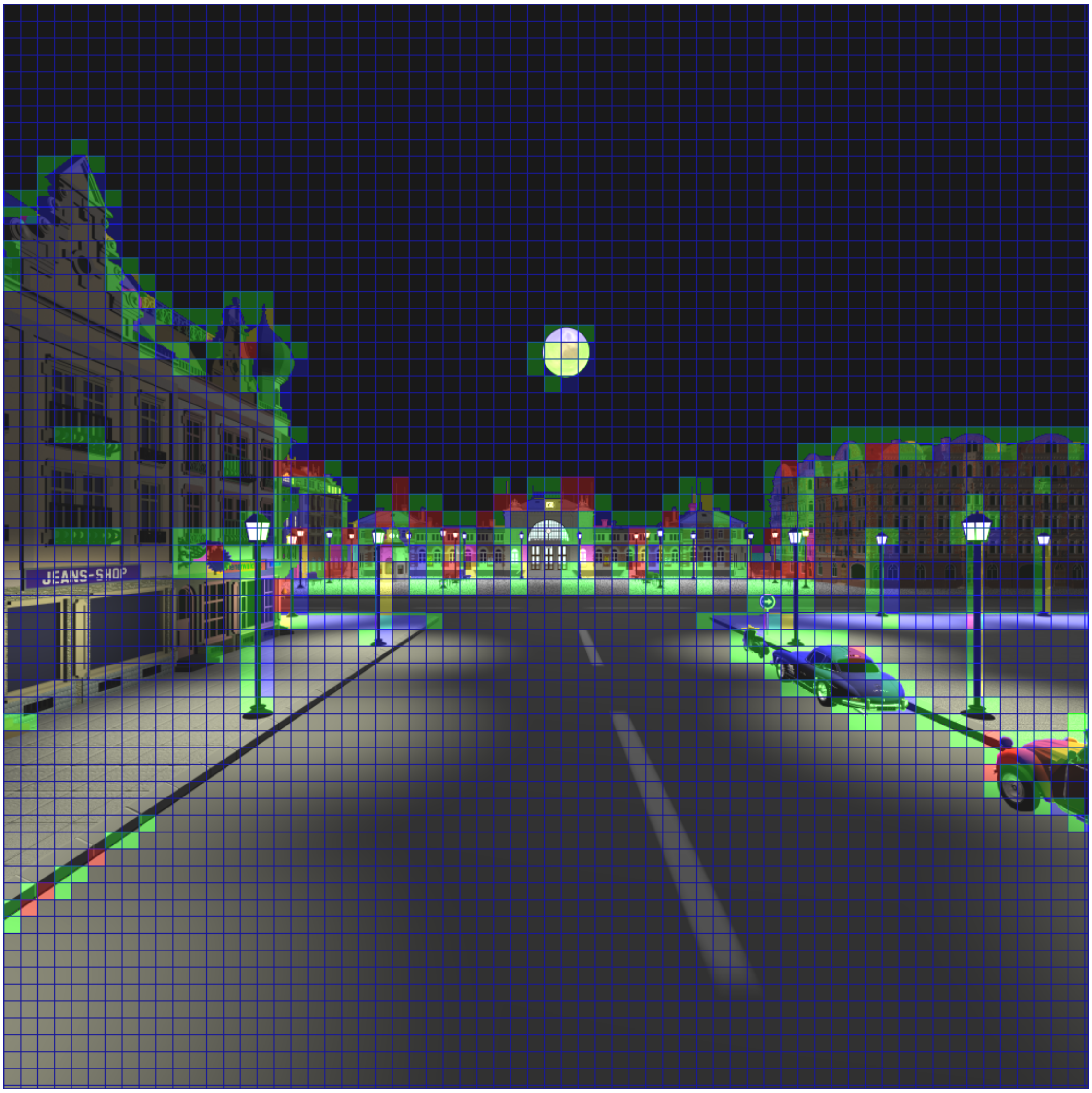


Figure 6.6. Tile segmentation. The *1028x1028* image is subdivided into *16x16* tiles. Most of the tiles that stretch across two or more surfaces were successfully segmented. The maximum number of subsets allowed was 6. The tiles that could not be segmented are shown in red. The segments are shown in different colors. For example the moon splits tiles in two segments, that are shown with green and blue highlights. The closest light pole from the right splits tiles in three segments, shown in green, blue and yellow.

I implemented two versions of the tile segmentation algorithm. One version proceeds conservatively by not adding a new sample to a segment if it has a depth discontinuity with a neighboring sample that is already in the segment. The aggressive version includes a new sample into an existing segment if it has at least one neighboring sample already in the segment with which it does not have a depth discontinuity. An example of the aggressive segmentation can be seen on the cast iron fixtures at the windows to the left. All samples belong to the same segment (green) since there are wall samples that are close to iron samples thus merging the segments. Representing the complicated structure with a single quad proved to be a reasonable approximation and thus we preferred the aggressive version that successfully segments a larger number of tiles (by using fewer segments).

6.2.3 Algorithm Implementation

```
1.  For each reference image
    1.1. Compute depth discontinuities
    1.2. Segment tiles
    1.3. Build quadtree of frusta
2.  Initialize vacuum buffer
3.  Initialize vacuum accounting tree (VAT)
4.  Clear item buffer
5.  For each reference image
    5.1. ProcessFrustum($F_0$)
6.  Done: VAT measures and locates the amount of vacuum left in the view frustum
```

The recursive ProcessFrustum(Frustum *F) routine is summarized next:

```
1.  if (F == null) return
2.  if F not leaf and F->closestZ is F->parent->closestZ
    2.1. go to 6
3.  Transform, clip, and project frustum
4.  Scan-convert faces in item buffer
    4.1. if item buffer location hit first time
         4.1.1.  UpdateZBuffer($z_0$)
         4.1.2.  UpdateItemBuffer(F)
    4.2. if item buffer location hit second time
         4.2.1.  dv = UpdateVacuumBuffer($z_0$, $z_1$)
         4.2.2.  UpdateVAT(dv)
    4.3. if current face is occluder
         4.3.1.  dv =UpdateVacuumBuffer($z_0$, yon)
         4.3.2.  UpdateVAT(dv)
5.  if F is leaf
    5.1. ProcessFrustum(F->next)
6.  else
    6.1. for (i = 0; i < 4; i++)
         6.1.1.  ProcessFrustum(F->child[i])
```

The reader will recognize the algorithm described previously. The resolution at which vacuum is determined, in other words, the number of rays in the vacuum buffer, does not have to be the resolution at which desired images will ultimately be produced. A lower vacuum-buffer resolution can be used (I used 320x240 for the VGA resolution output). However, the resolution cannot be lowered indefinitely since the projection of the quads that approximate the tile height-field must have meaningful sizes and shapes.

The Vacuum Accounting Tree (VAT) is a fast way of knowing how much vacuum is left in the desired view frustum and where it is located. It is the quadtree subdivision of a buffer of vacuum buffer resolution. For every vacuum buffer location a leaf node stores the sum of the vacuum spans in that linked list. In other words a leaf stores the amount of vacuum along a certain ray of the vacuum buffer. A higher level node stores the sum of the vacuum stored at its four children. The root stores the amount of vacuum left in the entire vacuum buffer. The VAT is initialized to full once per desired view: vacuum from hither to yon along all rays in the vacuum buffer. It is then updated as frusta are processed, using the amount of vacuum determined by the current air-faces sample pair or current occluder sample.

The item buffer is also at the vacuum buffer resolution and it stores unique frusta identifiers. It is used to detect second hits of samples from the faces of the same frustum. Since the frusta identifiers are unique, the item buffer does not need to be cleared from one frustum to the next. The z-buffer, also at the vacuum buffer resolution, is used to record the z of the first hit. When the second hit occurs, the value in the z-buffer and the new z value are used to update the vacuum buffer. Updating the vacuum buffer is equivalent to subtracting the newly determined interval from the vacuum intervals stored in the list at that location. The amount of vacuum determined is returned and used to update the VAT. The z-buffer does not need to be cleared since it is always used in conjunction with the item buffer.

The tiles that are segmented generate two or more frusta at the leaf level of the reference image quadtree. In this case the additional frusta need also to be processed (`ProcessFrustum(Frustum *F)`, step 5.1 and 1). The frusta are processed recursively. If the current frustum has the same closest z as its parent, it doesn't need to be processed since no progress can be made (`ProcessFrustum(Frustum *F)`, step 2).

A case that must be treated carefully is the case of frusta clipped by the hither plane. New faces must be introduced to make sure that the faces that are scan-converted indeed form a convex polyhedron; scan-converting a frustum without the top face incorrectly misses second hits since one can "see inside" the frustum.

One might wonder what happens when only one hit occurs or several hits occur, due to potential scan-conversion precision limitations. If only one hit occurs, typical for the desired-view silhouette of the frustum, there will be no air span generated to update the vacuum. This is the correct degenerate-case behavior. If more than two hits occur, typical for edges of the frustum that project over another face, the vacuum buffer is updated using tiny air spans. The results are correct and the sole penalty is in efficiency. Such cases are avoided by setting a threshold below which air spans are discarded.

From **Figure 6.4** one can see that the typical vacuum buffer update is done with adjacent air spans. To insure the adjacency in the presence of numerical error, the vacuum-buffer update routine starts by increasing the air span with epsilon at each end. Adjacency is of course important since it keeps the vacuum-span lists short, which translates into efficient update times.

The next section presents the application of the vacuum-buffer algorithm to reference-image sample choosing, called tile choosing since tiles are the level at which samples are selected.

6.3 Tile choosing

In Chapter 1 I discussed the requirements for a sample-selection method. Here they are again, phrased more concisely. An ideal set of samples satisfies the following conditions:

- *completeness*: all surfaces visible in the desired view should be represented
- *good quality*: the resolution at which the surfaces are sampled in the reference images should be as close as possible to the resolution at which the surfaces are sampled in the desired view; *oversampling* leads to aliasing artifacts and high cost and *undersampling* leads to poor quality (blurriness)

- *non-redundancy*: invariably some surfaces are sampled in more than one reference image; assuming that the surfaces are close to diffuse, the multiple samples are equivalent and one can / should discard the redundant copies[24].
- *low depth complexity*: for efficiency, the set should contain very few or no samples that are hidden in the desired image.

For the reasons discussed previously, we split the reference images into tiles and the set of samples that has to be determined is actually a set of tiles. It is obvious that the vacuum buffer measures the completeness of a set of tiles but more must be said in order to explain how it can help with satisfying the other three conditions.

I measure the quality of the samples in a tile by analyzing how much it stretches or shrinks when projected in the desired image. Small changes indicate similarity between the sampling rates in the reference and output images thus the samples of such a tile are preferred. From an implementation point of view, the quality is efficiently derived from the size of the bounding box of the projection of the quad corresponding to the tile. Having defined a quality metric we must explain next how to determine whether two samples originating from different reference images sample the same surface.

This is done with an additional z-buffer and an additional item buffer at the resolution of the vacuum buffer that simply store at each location the z of the closest occluder and a pointer to the tile it originates from. When an occluding face is processed, the occluder z-buffer is consulted. If the current occluder sample is within epsilon of the closest occluder z, it is assumed to sample the same surface. If the quality of the current tile is better, the occluder item buffer location is overwritten, otherwise the current sample is simply discarded. This eliminates redundant samples of lesser quality.

If the current occluder sample is clearly behind the closest occluder sample, the current sample is again discarded. This reduces the depth complexity.

The tiles that could not be segmented in order to avoid internal depth discontinuities cannot be processed by the algorithm, and, conservatively, have to be chosen.

The tile-choosing algorithm starts with the reference images that were acquired from a location closest to the desired camera location. More and more distant reference images are processed until the amount of vacuum remaining is below a certain threshold. The chosen tiles are the tiles that have at least one sample present in the occluder item buffer: they were not completely occluded nor were they completely replaced by better tiles. In order to avoid scanning through the occluder item buffer, presence counters are maintained for each tile.

Figure 6.7, **Figure 6.8**, and **Figure 6.9** illustrate tile choosing using the vacuum-buffer algorithm.

[24] Even in the case of specular surfaces, ensuring that each surface is sampled only by one reference image helps avoiding inconsistencies that produce disturbing speckling and frame to frame flickering. The shading is of course not correct but the static highlights are by far more esthetically pleasing than perfectly diffuse surfaces.

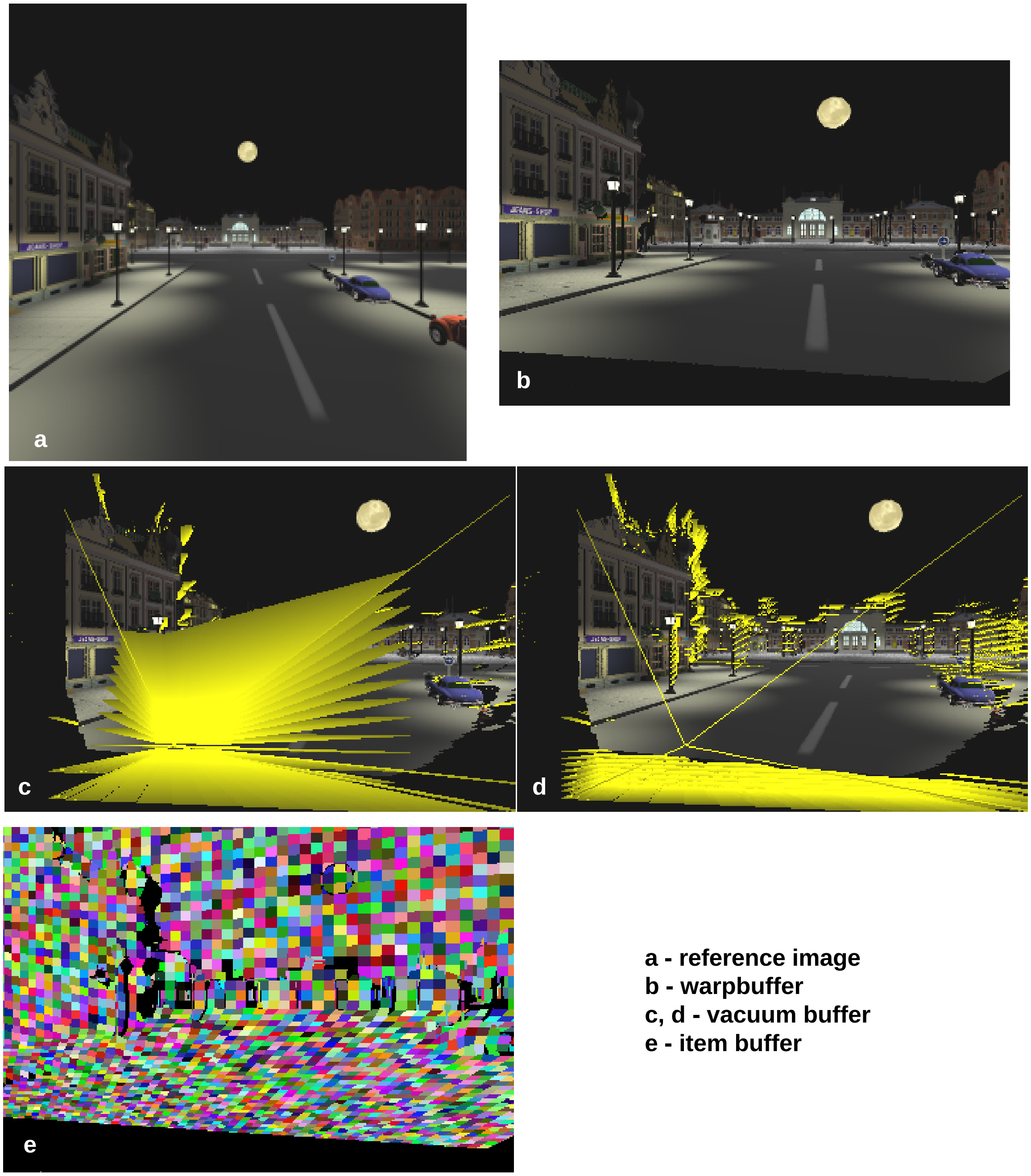
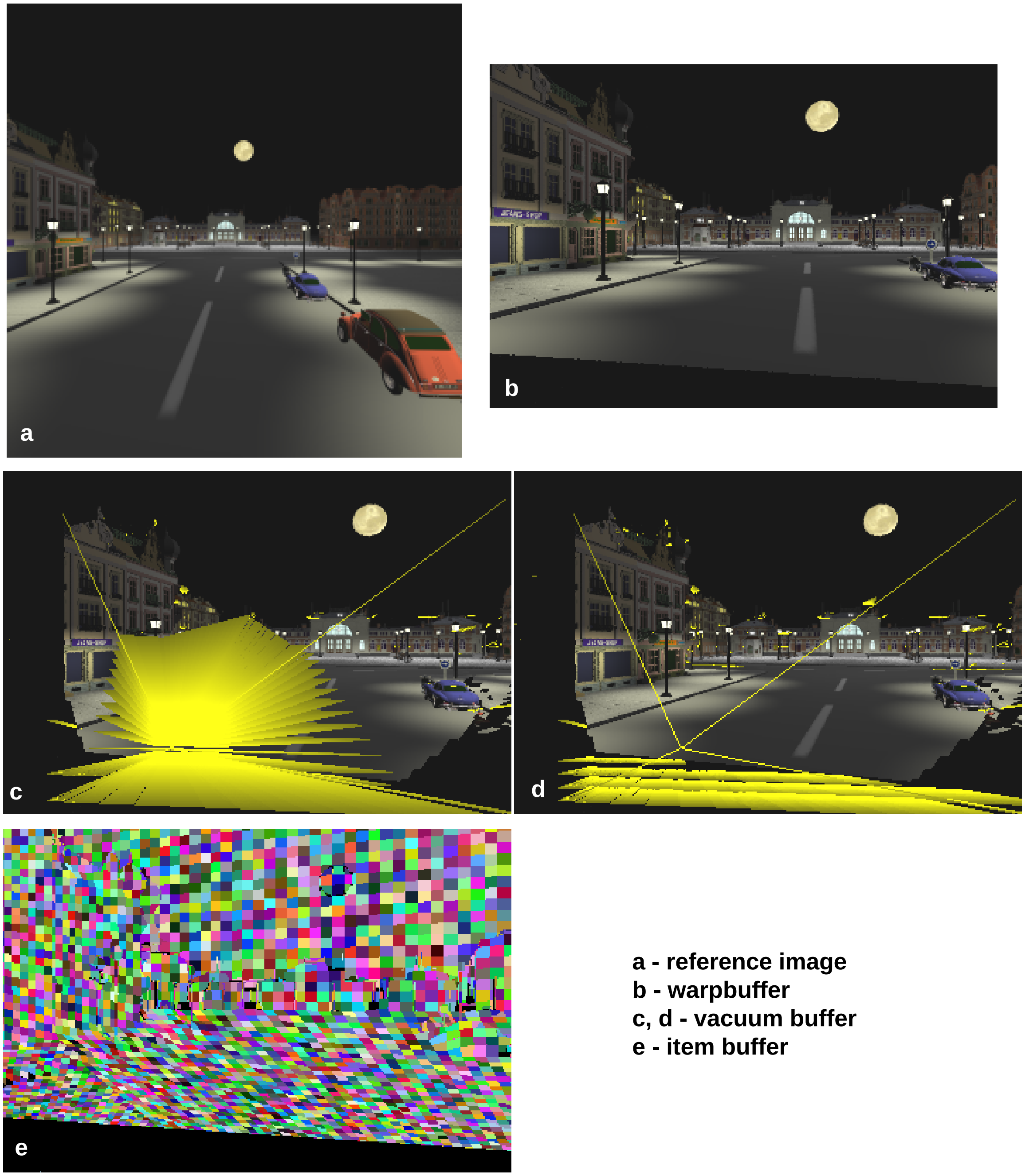


Figure 6.7. Vacuum buffer (1). The reference image showed is the first reference image processed. Image *b* shows the warpbuffer, which is not computed by the algorithm and is shown here just for illustration purposes. (The warpbuffer is obtained by warping the tiles chosen so far.)

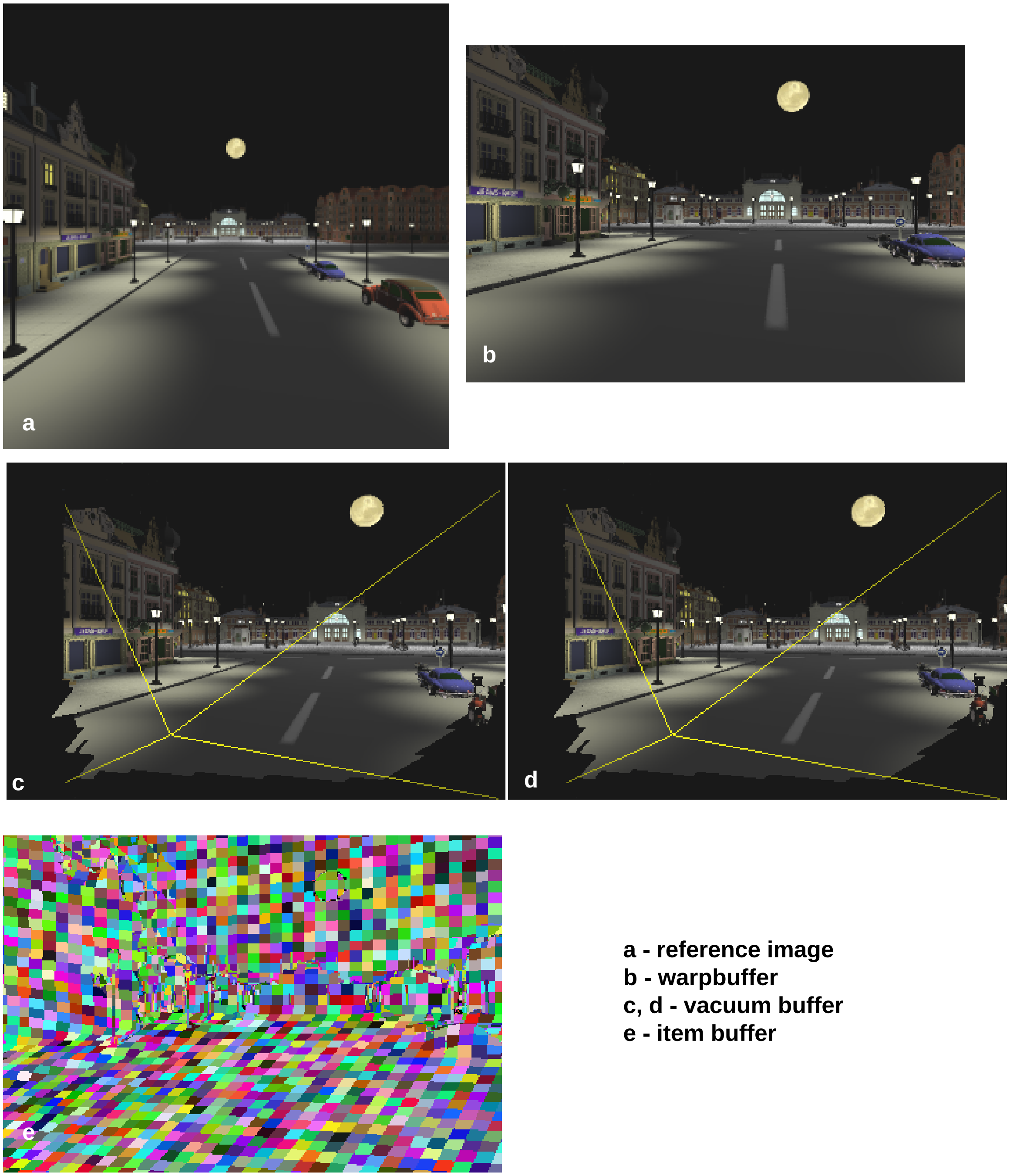
Images *c* and *d* show the vacuum buffer; segments shaded from bright yellow (white) - close end - to dark yellow (gray) - far end- represent the vacuum spans. The vacuum buffer is shown from an offset view to expose the rays. The desired view frustum is shown with the four concurrent yellow (light) segments. Every column and every *k*-th row of the vacuum buffer are shown. In image *d*, nearby vacuum is not shown and the number *k* of vacuum-buffer rows shown is increased. The vacuum buffer is shown composited with the warpbuffer. Vacuum persists close to the camera (since outside of the reference view frustum) and behind occluders (cars, light poles, buildings).

Image *e* shows the occluder item buffer, which stores the tiles chosen so far. Each tile (or tile segment) is shown with a different color (gray shade).



a - reference image
b - warpbuffer
c, d - vacuum buffer
e - item buffer

Figure 6.8. Vacuum-buffer (2). As one more reference image is processed, the vacuum is further reduced, since the second image "sees" behind the occluders of the first reference image. The item buffer is almost fully instantiated now, which does not necessarily mean that no visible surfaces are missed.



a - reference image
b - warpbuffer
c, d - vacuum buffer
e - item buffer

Figure 6.9. Vacuum buffer (3). After more reference images are added (the reference image shown is the sixth and the last one), the amount of vacuum remaining in the vacuum buffer drops below a pre-established threshold. The item buffer contains the chosen tiles.

In order to match the sampling rate of the desired image, the algorithm gives preference to tiles that have the projected size close to the original size. The original size is the size of the sky tiles, which are not affected by the reprojection since they are at infinite distance. For example the tiles that sampled the street in **Figure 6.8** are replaced here with tiles of larger projected area.

6.3.1 Results

The results of the sample-selection technique were presented in section 5.4, when the WarpEngine performance was analyzed. The vacuum-buffer sample-selection algorithm was coded as OC in the tables and graphs of that section.

In this subsection I will compare the method to an earlier version of the tile-choosing algorithm. The earlier version did not use the vacuum buffer. It just considered the 8 sampling locations defining the cell of the current camera position and rendered the tiles in the occluder item buffer and the occluder z-buffer. In order to minimize the chances of missing a visible surface, the cell size was small (3m x 3m x 3m for eurotown). Also no tile segmentation was attempted.

With the vacuum buffer I was able to double the size of the cell. This equates to 8 times fewer reference-images, which is obviously of great importance when real world data is acquired. Like in the previous case the tile choosing starts by considering the images of the current cell. Tile choosing stops when the total amount of vacuum decreases below a threshold. I used the generalized disparity for the vacuum buffer and the initial vacuum span was (100.0, 0). The generalized disparity (1/z when the image plane distance is 1) is convenient since it gives more importance to vacuum spans that are close to the camera. Objects that are close have a large screen area and the artifact resulting from missing them is more noticeable. In the particular case of my simulations, the threshold below which no more disocclusion errors were noticeable was 1500 (the average per location amount of vacuum is about 0.02). Increasing the size of the cell means that occasionally one had to consider images from neighboring cells since the vacuum would not decrease below the set threshold only by using the reference images of the current cell.

Tile segmentation was done allowing a maximum of 6 segments per tile. If a tile could not be successfully segmented it was conservatively chosen. **Table 6.1** presents the results of tile choosing for both the old and the current method with output at VGA resolution.

	Old method	**Vacuum buffer method**
Visible tiles	15K	13K
Selected tiles & tile segments	7K	3.3K
Selected unsegmented tiles	2.5K	1.8K
Selected tiles that are segmented	0	1.1K
Selected tile segments	0	1.5K
Unsegmentable tiles	4.5K	0.4K

Table 6.1 Tile choosing performance with and without using the vacuum buffer.

First, there are fewer visible tiles since the reference images are further apart. The number of selected tiles (including segments) is much lower mainly because the number of unsegmentable tiles decreased substantially. The number of selected tiles that did not have to be segmented (no depth discontinuities) also decreased since the tile segments eliminated some of the full tiles.

In a VGA resolution image there are *1.2K 16x16* tiles. So the ratio between the number of reference-image samples selected and the number of output image samples is 2.75. We counted 256 samples per tile segments since in the case of the WarpEngine architecture the samples of a tile are processed in SIMD fashion and one cannot save if some samples are "off". Other factors that make this figure deviate from the ideal value of 1.0 are:

- The closest reference-image sampling rate for a surface is not *exactly* the desired image sampling rate
- There still are a few tiles that are not segmented, which cause undetected redundancy.

The next section analyzes the complexity of the algorithm and investigates possible hardware acceleration.

6.4 Hardware Acceleration For Vacuum-Buffer Algorithm

As stated earlier, the resolution of the buffers used in the algorithm can be lower than the resolution of the final image. Thus clearing the buffers (once per frame) is not a substantial burden. The bulk of the work is done at step 4 of the `ProcessFrustum` routine.

If at most eight sampling locations are considered at each frame, and if cube depth-panoramas are used centered at the sampling location, there could be as many as *6 x 8 = 48* reference images that must be considered.

Of course not all reference images have samples that project in the view frustum. At step 3, if the current sub-region of the reference image is completely outside the view frustum it is culled and the recursion is stopped early. In order to determine whether the sub-region is needed, one transforms, clips and projects a second bottom face of the frustum, defined by the *farthest* z. If the larger frustum defined by the parent's closest z plane and the farthest z is outside of the view frustum, the subregion is not useful for the current view and can be safely ignored (**Figure 6.10**). In our test scenes, using *1k x 1k* reference images[25] on average *13K 16x16* tiles passed the view-frustum-culling test. This is equivalent to 3-4 full reference images.

The number of frusta that have to be processed for each full reference image is:

$$N = F_{1024} + 3F_{512} + 4(3F_{256} + 4(... + 4(3F_{32} + 4F_{16}))) = 5120,$$

where F_w is a frustum that corresponds to a w x w subregion. Analytically, the number of frusta can be computed with the formula

$$N = 4^k + 4^{k-1} = 5120$$

where k is defined as

$$k = \log_2 w_{image} - \log_2 w_{tile} = \log_2 1024 - \log_2 16 = 6$$

[25] The reference images had 90 degrees field of view and the output view frustum was 65 degrees horizontally.

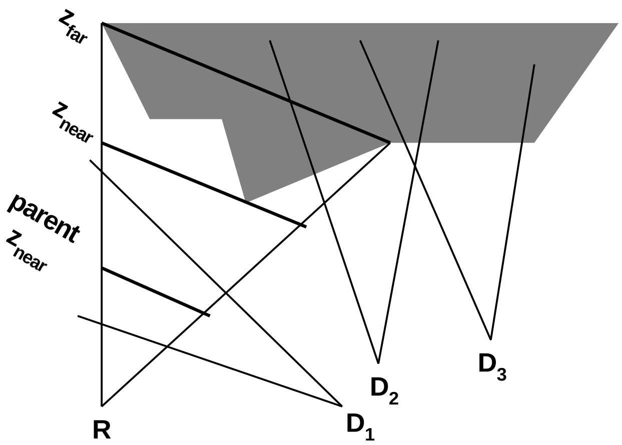


Figure 6.10. Vacuum-buffer algorithm: view-frustum culling. Only the current subregion of the reference image R is shown. Out of the possible desired view frusta D_1, D_2, and D_3, the subregion can be ignored only for D_3. The air information of the subregion is pertinent to D_1, which wouldn't have been detected if we used the frustum defined by the z_{near} and z_{far} planes. If we used the *parent* z_{near}- z_{near} frustum, as required by processing the subregion's air information, the subregion would have incorrectly been ignored for D_2. The only conservative approach is to use the *parent* z_{near}- z_{far} frustum.

Each frustum has 6 quadrilateral faces, thus, the number of triangles that must be rendered per second, assuming 30Hz update rate is[26]:

$$T = 30 * 4 * 6 * 2 * 5120 \approx 7 Mtris / s$$

My software implementation takes about 20 seconds per frame, which is orders of magnitude too slow for interactive rendering. The average number of triangles rendered per frame for the eurotown scene was *263Ktris,* which represents *7.8Mtris/s*, confirming the estimate above.

The obvious way to accelerate the vacuum buffer is to use polygon-rendering hardware. Besides the fact that *7Mtris/s* is a sizeable task even for the most recent polygon-rendering hardware, there is the problem of updating the vacuum buffer.

Graphics hardware doesn't offer the possibility of storing several z values at each location. Having the CPU read back the z-buffer and item buffer after each face of a frustum is rendered is not a practical solution. First, the operation is very costly on most graphics architectures. Second, the CPU would have to traverse the bounding box of the projection of the face in order to use the z's of the current face, which most likely offsets the advantage of hardware rendering.

I believe that the most promising solution is to design hardware for accelerating the vacuum buffer. The architecture would be similar to current triangle rendering architectures. However, the rendering task consists only of flat shaded triangles, so the architecture will be simpler in many ways.

[26] The tiles belong to more than four reference images so there are more higher level frusta than in the ideal case of full images.

The only additional complexity is a z-buffer that can store several samples at each location. Because of the incremental updates to the vacuum buffer, the number of z's that must be stored is small, and it seems practical to provide storage for a fixed number of z values. For our scene the next table shows, how many times per frame, on average, a list grew longer than a certain value. For example on average, at each frame, there were 118 lists that grew from length 7 to length 8.

Length	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Exceeded	2785	925	399	118	28	5	2	0

Table 6.2 Maximum length of lists of vacuum spans

If the limit is exceeded, exceptions can be raised that discard the shortest interval or eliminate the smallest gap by merging nearly adjacent intervals (whichever is smaller). In my simulations, if the maximum hardware list length is set to 8, the amount of vacuum incorrectly eliminated to serve the exception was negligible[27] and the same set of tiles was chosen as in the case of unlimited list lengths. Since the resolution of the vacuum buffer is low, it is feasible to store a maximum of 8 spans at each location[28].

6.5 Conclusions

In this chapter I presented a method of selecting reference samples to be warped to create the desired view. The method uses the vacuum-buffer algorithm to conservatively estimate the subvolumes of the view frustum that have not been determined by the reference images considered so far. A software-only implementation is too slow to be practical. The bulk of the work consists of rendering polygons and the number of polygons that must be rendered per second is within the capabilities of today's polygon-rendering hardware. The modification required is to extend the z-buffer to contain several spans of z. The number of spans at a vacuum buffer location is small. This enables a simple hardware implementation that supports vacuum-span lists of fixed maximum length.

[27] The average was below .1 where the original vacuum span (hither, yon) is (100.0, 0)

[28] Fixed-length span-lists can be implemented on PixelFlow. PEs can store the spans in their local memory and they can be programmed to do the fuzzy-z-test updates. However, the implementation would not be very efficient as updates occur only at one PE, and the cycles of the other PEs are wasted.

Chapter 7 Forward Rasterization for Polygon Rendering

I showed in Chapter 4 that the forward-rasterization algorithm has a reduced setup cost. I also pointed out that the forward-rasterization generates more samples than the area of the polygon (quad) that it rasterizes. In the case of IBRW this disadvantage is minor, mainly because shading is inexpensive and thus shading the extra samples is a small additional cost.

In polygon rendering, shading can potentially be expensive, involving one or more texture look-ups per sample, and / or the evaluation of complicated expressions. The quad forward-rasterization algorithm derived for IBRW could be applied to rendering polygonal models, but the advantage of cheaper setup might be outweighed by the cost of shading the redundant samples. A case when forward-rasterization as described for IBRW might still win is the case of rasterizing the quads resulting from the tessellation of higher-order surfaces. Also, the commonly used primitive in polygon modeling and rendering is the *triangle*.

These considerations prompted me to try to design a forward-rasterization algorithm for triangles that generates as few samples as possible. My first efforts in this direction are presented in this chapter.

7.1 Forward-rasterization algorithm for triangles by minimal baricentric sampling

Forward rasterization is a class of algorithms that share the following properties:

- the samples generated are independent of the pixel grid as they are defined in the continuous parameter domain, and,
- reconstruction is done using offsets in order to control the aliasing due to truncation errors.

Desirable properties of forward-rasterization algorithms are reduced setup cost and reduced number of redundant samples (samples that land at the same pixel). As noted above redundant samples are costly in the case of polygon rendering. I will describe a forward-rasterization-algorithm for triangles that has reduced setup cost and produces only a few redundant samples. If combined with a simple method of early discarding of redundant samples, the algorithm does produces virtually no redundant samples.

7.1.1 Baricentric sampling of a triangle

If the triangle is treated as a degenerate quad, a large number of redundant samples are generated (**Figure 7.1**). The sampling is non-uniform and becomes unnecessarily dense close to the degenerate edge. A much more uniform sampling is achieved if the step along the scan-line[29] is not decreased (**Figure 7.2**).

[29] The term scan-line is used here to denote the samples generated by the inner-loop at each iteration of the outer-loop of the rasterization routine

It is trivial to show that the small quads generated by four neighboring samples are parallelograms. In the case shown in **Figure 7.2**, the sides of the parallelograms are parallel to edge V_0V_1 and edge V_1V_2. This sampling of a triangle is known as baricentric sampling.

When deriving **Equation 4.11** for the interpolation factor, I addressed the general case, when four neighboring samples generate a random quad. Consequently **Equation 4.11** holds in the case of equal parallelograms, but is too conservative. In the next section I will show that the interpolation factor can be much coarser (with the benefit of fewer redundant samples) while all pixels inside the triangle are still guaranteed to be hit, as needed to maintain surface continuity.

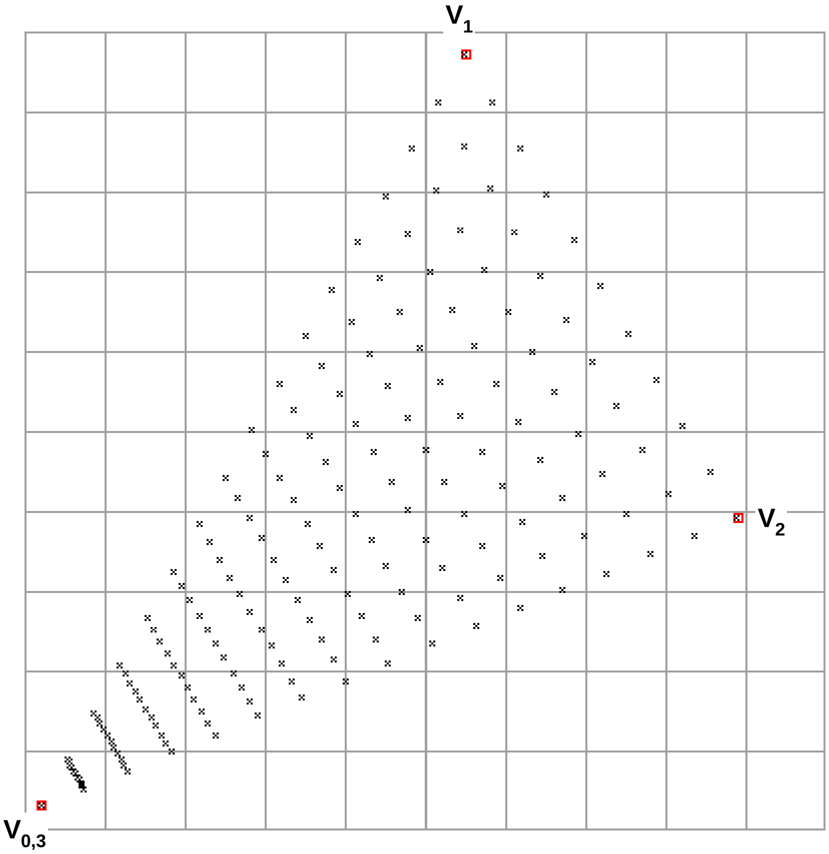


Figure 7.1. Bilinear interpolation of triangles. The figure shows the samples generated if the triangle $V_0V_1V_2$ is considered a degenerate quad (V_0 coincides with the fourth vertex V_3). 187 samples are generated that hit 47 pixels. V_1V_2 and the subsequent scan-lines are sampled 11 times. V_0V_2 and V_0V_1 are sampled 17 times. These interpolation factors were computed according to **Equation 4.11**. Also notice that the scan-lines are parallel to V_1V_2.

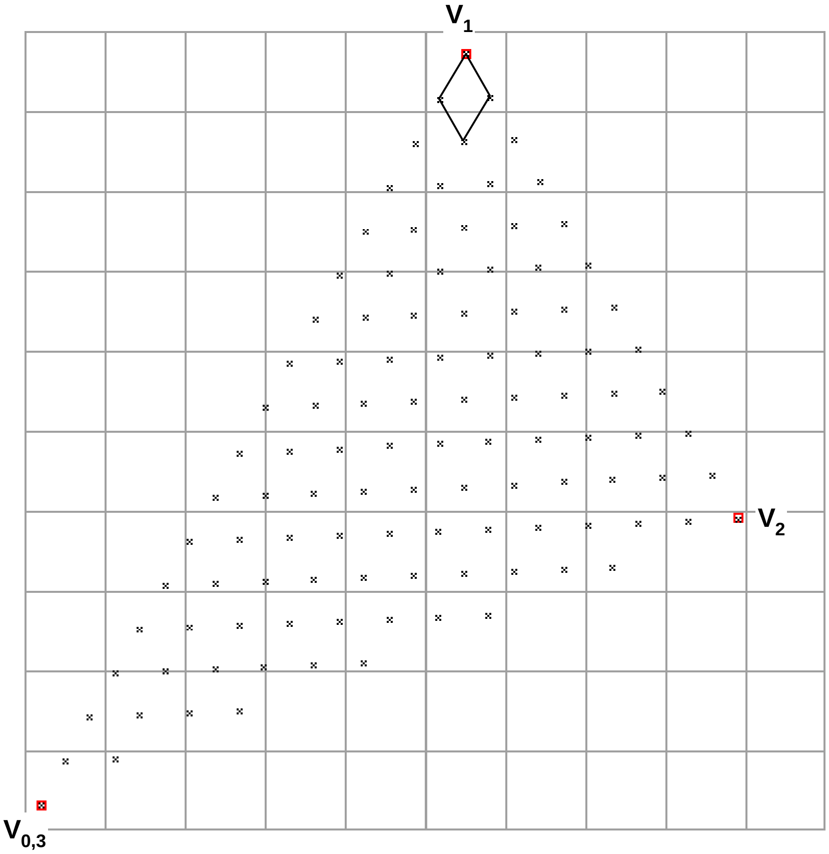


Figure 7.2. Baricentric interpolation of triangles. The step along the scan-line is not decreased and a uniform sampling is obtained. The interpolation starts at the edge V_1V_2 and proceeds towards vertex V_0 on parallel scan-lines. Scan-lines always begin on the edge V_1V_0, and, in order to know when to stop, the sidedness expression for edge V_2V_0 must be evaluated. Four neighboring samples generate a parallelogram[30] (only one shown). 109 samples are generated, and 42 pixels are hit[31].

[30] Except for the effects of the ceiling operation in **Equation 4.11**, the parallelogram is in fact a rhombus, since all sides are 1 / sqrt(2) pixels long.

[31] The difference in the number of pixels hit, when compared to **Figure 7.1**, is due to the different sampling of edge V_2V_0. For now the discussion concentrates on the sampling of the inside of the triangle.

7.1.2 Forward-rasterization of triangles by minimal baricentric sampling

In the previous section we saw how samples can be generated along two directions parallel to two of the edges of the triangle. The next problem is determining the coarsest possible interpolation factor along these two directions such that every pixel inside the triangle is hit.

Let e_1 and e_2 be the angles that the two edges make with the x-axis (measured from x+). Without loss of generality, one can assume that:

$$0 \le e_1 \le e_2 < 180^{\circ}$$

Equation 7.1. Sampling directions defined according to their angle with the x-axis.

According to which half-quadrant(s) the two edges occupy, there are 10 cases to be analyzed (**Table 7.1**, **Figure 7.3**).

Case	**Half-quadrant**	
	e₁	***e₂***
1	1	1
2	1	2
3	1	3
4	1	4
5	2	2
6	2	3
7	2	4
8	3	3
9	3	4
10	4	4

Table 7.1 Classification of edge pairs according to half-quadrants

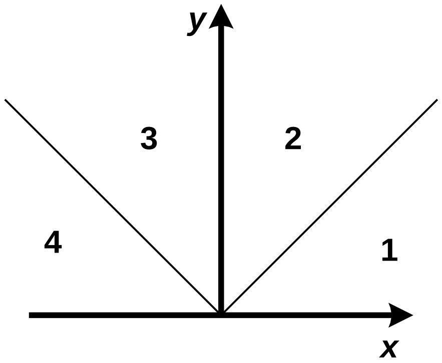


Figure 7.3. The *y* > 0 half-plane subdivided in four half-quadrants.

I will call the parallelogram determined by four neighboring samples the *sampling parallelogram*. For each case, one has to choose the lengths of the sides of the sampling parallelogram to make sure that:

- all pixels covered by a mesh of such sampling parallelograms get at least one sample, and,
- the number of samples generated is as small as possible.

I will formulate an equivalent condition, easier to use in the case analysis. The number of samples generated is given by **Equation 7.2**, which is proved in **Figure 7.4**.

$$N = \frac{if_1 if_2}{2}$$

Equation 7.2. Number of samples generated by baricentric interpolation of a triangle, where $if_1 \ x \ if_2$ is the interpolation factor.

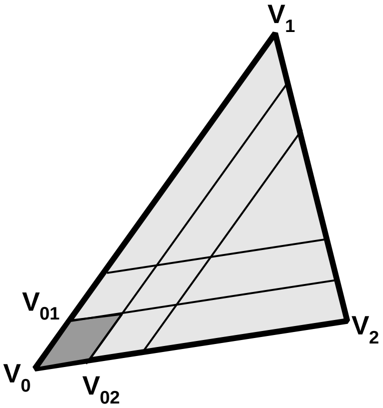


Figure 7.4. Number of samples in baricentric sampling of triangles. The number of samples is equal to the ratio between the area of the triangle and the area of the sampling parallelogram. The area of the triangle is $V_0V_1 * V_0V_2 * sin\ V_0\ /\ 2$. The area of the sampling parallelogram is $V_0V_1\ /\ if_1 * V_0V_2\ /\ if_2 * sin\ V_0$, which proves **Equation 7.2**. An alternative proof is to consider the big parallelogram composed of two triangles $V_0V_1V_2$ with V_1V_2 as a diagonal (not shown). The total number of samples is half the number of sampling parallelograms contained in the big parallelogram.

According to **Equation 7.2**, the only way to obtain a minimum number of samples, is to minimize the product if_1*if_2. The lengths of the sides of the triangle are given, so minimizing the product if_1*if_2 is equivalent to maximizing the product between the lengths of the sides of the sampling parallelogram ($V_0V_{01} * V_0V_{02}$ in **Figure 7.4**).

Making sure that *every* pixel inside the triangle gets at least one sample is equivalent to making sure that an *arbitrary* pixel inside the triangle gets a sample. Since the sides and the angles of the sampling parallelogram are determined by the triangle and the interpolation factor, the sampling parallelogram covering the pixel is determined except for a translation. We also know that there exists a sampling parallelogram that contains the center of the pixel. So one has to choose the interpolation factor such that all sampling parallelograms that contain the center of a pixel also have a vertex inside that pixel.

In conclusion, one must determine the lengths of the sides of the sampling parallelogram for every case such that:

- the product of the lengths is maximum, and,
- all sampling parallelograms that contain the center of a pixel have a vertex inside that pixel.

7.1.2.1 Case 1

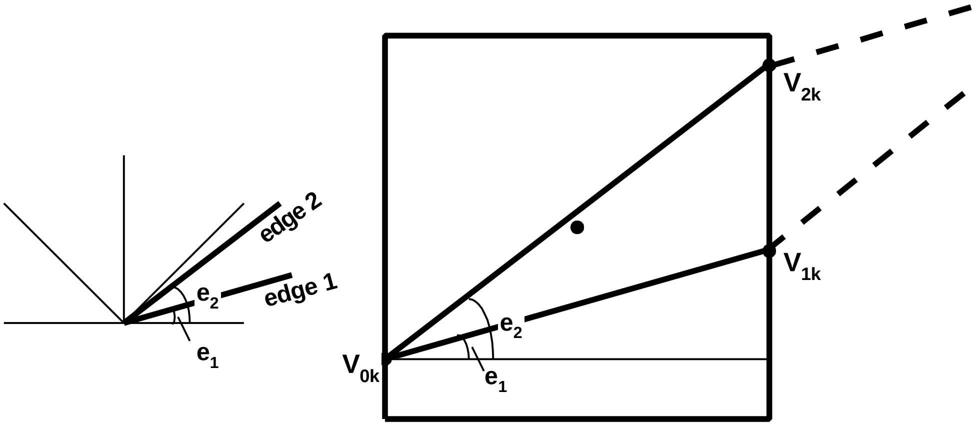


Figure 7.5. Case 1: both edges in lower half of the first quadrant. The right figure shows a random pixel (big square) inside the triangle to be rasterized. V_{0k}, V_{1k} and V_{2k} define half of a sampling parallelogram. $V_{0k}V_{1k}$ is parallel to the edge V_0V_1 of the triangle (not shown), and $V_{0k}V_{2k}$ to V_0V_2.

Equation 7.3 gives the lengths of the sides of the sampling parallelogram if they are chosen according to **Figure 7.5**. All sampling parallelograms that contain the center of a pixel will have at least one vertex inside that pixel[32]. In order to try to move vertex V_{0k} outside of the pixel, one has to translate the sampling parallelogram (that is completely defined except for a translation). When translating downward, V_{0k} has to pass the bottom left corner of the pixel, but at that time the center of the pixel will be on the wrong side of $V_{0k}V_{2k}$ since e_2 is less than 45 degrees. The reasoning is similar for the other translation directions, which shows that the sampling parallelogram shown will always have a vertex inside the pixel.

$$V_{0k}V_{1k} = \frac{1}{\cos e_1}$$

$$V_{0k}V_{2k} = \frac{1}{\cos e_2}$$

Equation 7.3. Conservative lengths of sides of sampling parallelogram for case 1.

Equation 7.3 provides the easy to compute interpolation factor given in **Equation 7.4**.

[32] For the discussion at hand, the edges of the pixel are part of the pixel

$$if_1 = \lceil x_1 - x_0 \rceil, if_2 = \lceil x_2 - x_0 \rceil$$

Equation 7.4. Conservative interpolation factor for case 1. x_0, x_1, x_2 are the x coordinates of vertices the vertices of the triangle.

The lengths given in **Equation 7.3** are over-conservative. The next subsection derives the expressions for the absolute coarsest interpolation factor for case 1. Unfortunately these expressions are expensive to evaluate. Recall that the motivation of forward rasterization is reduced setup cost, so the expressions derived next have only theoretical value. In practice I have successfully used **Equation 7.4**.

Coarsest interpolation factor for case 1

By looking at **Figure 7.5**, one can see that if *both* $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ are longer, the sampling parallelogram can be translated such that none of the vertices belong to the pixel; but if only *one* side is lengthened, there will still be one vertex inside the pixel. In some cases, $V_{0k}V_{1k}$ cannot be lengthened, as can be seen in **Figure 7.6** (part a). However, there are cases in which $V_{0k}V_{1k}$ can be longer.

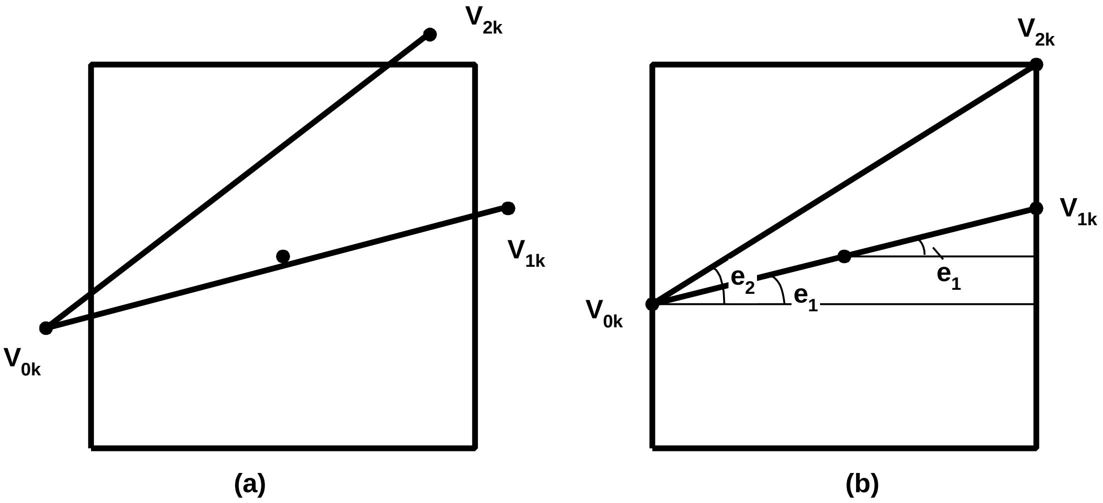


Figure 7.6. Determination of maximum lengths of sampling-parallelogram sides in case 1. (a) If $V_{0k}V_{1k}$ is longer, one can place the sampling parallelogram that contains the center of the pixel such that none of the vertices are inside the pixel. Consequently $V_{0k}V_{1k}$ cannot be longer than 1 / cos e_1. (b) If the angles e_1 and e_2 are such that V_{2k} is below the top right corner of the pixel when the center of the pixel is on $V_{0k}V_{1k}$, $V_{0k}V_{1k}$ can be longer, provided that $V_{0k}V_{2k}$ is exactly 1 / cos e_2.

The condition that determines whether $V_{0k}V_{1k}$ can be lengthened or not can be derived from **Figure 7.6** part b. One can see that in order for $V_{0k}V_{1k}$ to be longer, V_{2k} has to be below the top right corner of the pixel when $V_{0k}V_{1k}$ is shifted up to the center of the pixel. The condition on the angles e_1 and e_2 is given by **Equation 7.5**.

$$\tan e_2 - \tan e_1 \geq \frac{1}{2} - \frac{\tan e_1}{2}$$
$$1 + \tan e_1 - 2\tan e_2 \leq 0$$

Equation 7.5. Condition to prohibit $V_{0k}V_{1k}$ from being longer than 1/cos e_1.

First I will analyze the case in which $V_{0k}V_{1k}$ cannot be lengthened, and then the case in which it can. To compute how long $V_{0k}V_{2k}$ can be in this case, I will use the extreme case shown in **Figure 7.7**. Using the figure, one can compute the length of $V_{0k}V_{2k}$ as shown in **Equation 7.6**.

$$V_{ok}V_{2k} = V_{ok}O + OV_{2k}$$
$$\sin e_2 = \frac{\frac{1}{2} + \tan e_1}{V_{ok}O} \Rightarrow V_{ok}O = \frac{\frac{1}{2} + \tan e_1}{\sin e_2}, V_{2k}O = \frac{\frac{1}{2}}{\cos e_2}$$
$$V_{ok}V_{2k} = \frac{1 + 2\tan e_1}{2\sin e_2} + \frac{1}{2\cos e_2}$$

Equation 7.6. Deduction of maximum length for $V_{0k}V_{2k}$.

The interpolation factors are given by **Equation 7.7**.

$$if_1 = \lceil x_1 - x_0 \rceil, if_2 = \left\lceil \frac{x_2 - x_0}{\frac{1}{2} + \frac{1 + 2\tan e_1}{2\tan e_2}} \right\rceil$$

Equation 7.7. Interpolation factor for the case shown in **Figure 7.7**.

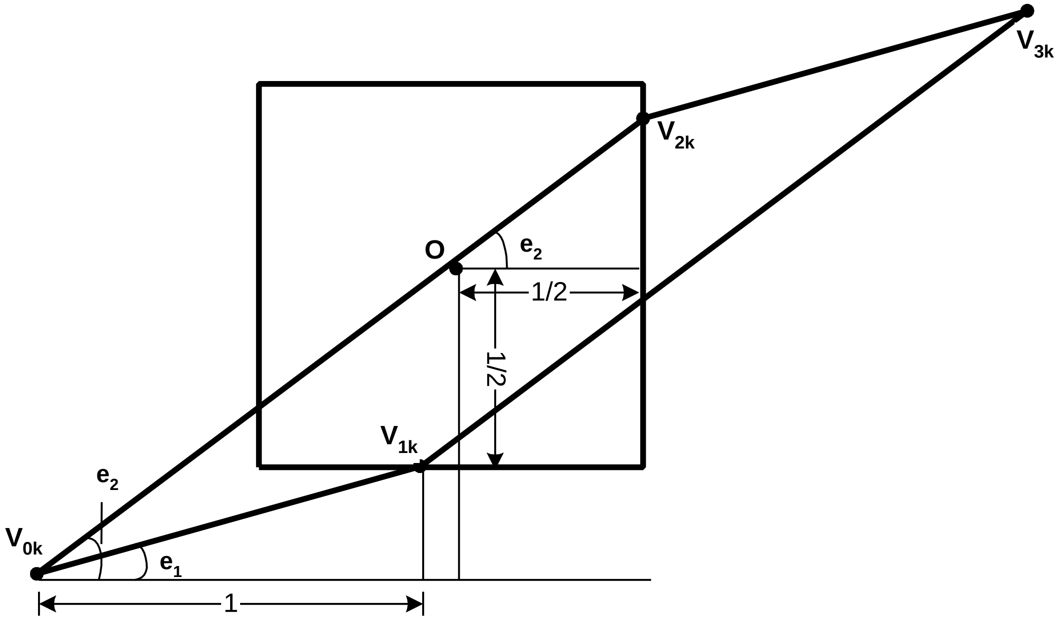


Figure 7.7. Extreme case used to derive the maximum length for $V_{0k}V_{2k}$. *O* is the center of the pixel and it is on $V_{0k}V_{2k}$.

In order to make sure that the case presented in **Figure 7.7** is the only possibility, I have to show that V_{1k} can be only on the bottom edge of the pixel, and not on the left edge. The extreme case that must be used is shown in **Figure 7.8**. I show in **Equation 7.8** that V_{1k} is always on the bottom edge if the condition given by **Equation 7.5** is met.

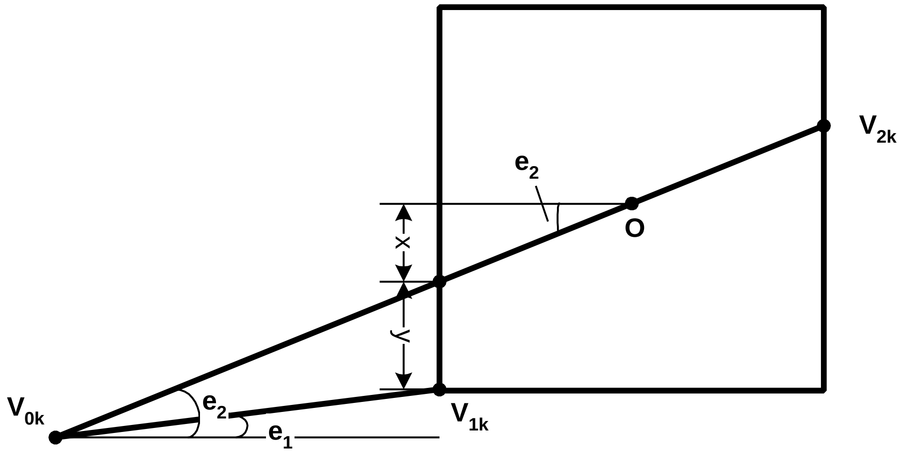


Figure 7.8. Case used to determine the condition for V_{1k} to be on the bottom edge of the pixel.

$$\frac{1}{2} - x < y$$
$$y = \tan e_2 - \tan e_1$$
$$x = \frac{\tan e_2}{2}$$
$$1 - \tan e_2 < 2\tan e_2 - 2\tan e_1$$
$$1 + \tan e_1 - 2\tan e_2 + \tan e_1 - \tan e_2 < 0$$

Equation 7.8. Condition for V_{1k} to belong to the bottom edge of the pixel.

The first three terms of the left-hand side of the inequality in **Equation 7.8** are exactly the left-hand side of the inequality in **Equation 7.5**, thus we know that their sum is negative. Since e_1 is smaller than e_2, the sum of the last two terms is also negative. Consequently the condition in **Equation 7.8** is always met if the condition in **Equation 7.5** is met.

We are left with analyzing the cases in which $V_{0k}V_{1k}$ can be lengthened, that is when the condition given in **Equation 7.5** is not met. **Figure 7.9** shows the extreme case that must be used in order to determine the longest possible $V_{0k}V_{1k}$. In this case, the lengths of the sides of the sampling parallelogram and resulting interpolation factor are given by **Equation 7.9**.

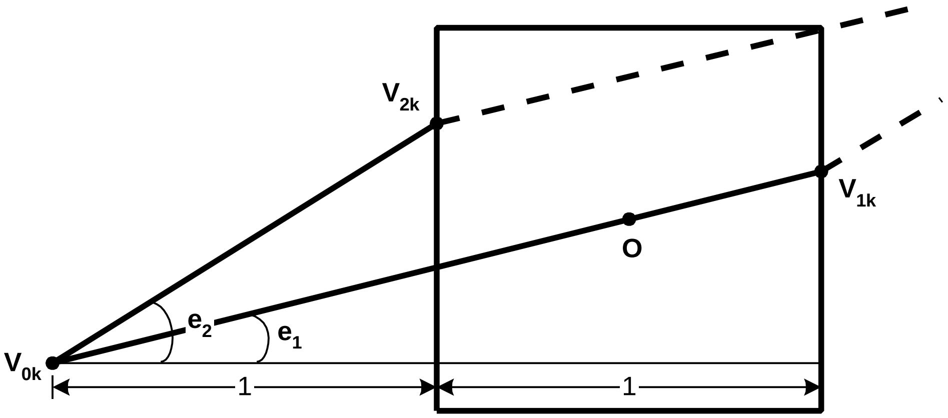


Figure 7.9. Extreme case used to determine the length of the side of the sampling parallelogram. V0kV1k cannot be longer than shown since it would then be possible to shift (laterally) the sampling parallelogram such that both V_{2k} and V_{1k} are outside of the pixel.

$$V_{0k}V_{1k} = \frac{2}{\cos e_1}$$
$$V_{0k}V_{2k} = \frac{1}{\cos e_2}$$
$$if_1 = \left\lceil \frac{x_1 - x_0}{2} \right\rceil$$
$$if_2 = \lceil x_2 - x_0 \rceil$$

Equation 7.9. Lengths of sides of sampling parallelogram and corresponding interpolation factor.

Remember that in this case (when $V_{0k}V_{1k}$ can be lengthened) either $V_{0k}V_{1k}$ is lengthened and $V_{0k}V_{2k}$ is kept unchanged (1 / cos e_2), or vice-versa. Moreover, when $V_{0k}V_{2k}$ is lengthened, either of the cases shown in **Figure 7.7** and **Figure 7.8** can occur. The overall goal is to minimize the product between the lengths of the two sides.

If the case in **Figure 7.8** occurs, the sides will have lengths as given in **Equation 7.10**. The product of the two lengths is the same as it is for the case shown in **Figure 7.9** (see **Equation 7.9**), so one can choose the sides of the sampling parallelogram as described in **Equation 7.9**.

$$V_{0k}V_{1k} = \frac{1}{\cos e_1}$$
$$V_{0k}V_{2k} = \frac{2}{\cos e_2}$$

Equation 7.10. Lengths of sides of sampling parallelogram for the case shown in **Figure 7.8**.

If the case in **Figure 7.7** occurs, we know that the condition in **Equation 7.8** is met. As shown in **Equation 7.11**, the product of the lengths of the sides is always bigger in the case shown in **Figure 7.9**.

$$\left(\frac{1+2\tan e_1}{2\sin e_2}+\frac{1}{2\cos e_2}\right)\frac{1}{\cos e_1}<\frac{2}{\cos e_1\cos e_2}$$
$$\frac{1+2\tan e_1}{2\tan e_2}+\frac{1}{2}<2$$
$$1+2\tan e_1+\tan e_2<4\tan e_2$$
$$1+2\tan e_1-3\tan e_2<0$$

Equation 7.11. Proof that the sides of the sampling parallelogram should be chosen according to the case shown in **Figure 7.9**. The last inequality is exactly the condition in **Equation 7.8**, which is known to be met.

Here is a summary of the discussion for case 1:

- if the condition given in **Equation 7.5** is met, the interpolation factors should be chosen according to **Equation 7.7**;
- else, the interpolation factors are given by **Equation 7.9**.

7.1.2.2 Case 2

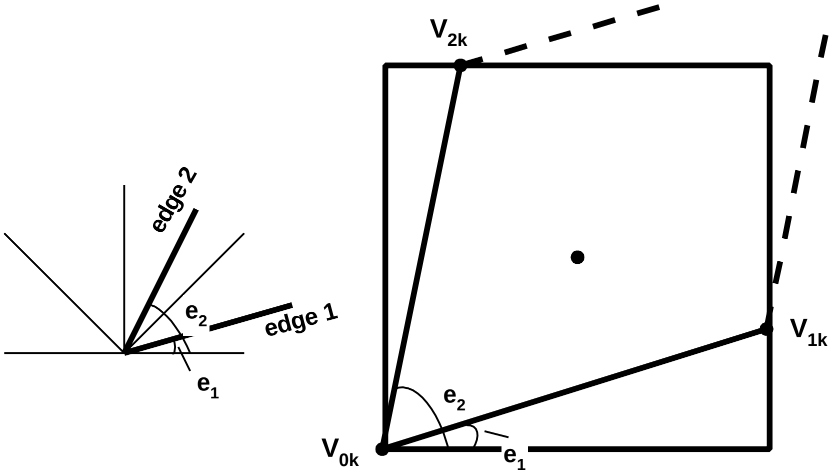


Figure 7.10. Case 2: 0° < e_1 < 45° < e_2 < 90°.

The analysis of this case begins by considering **Figure 7.10**. Choosing $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ as shown clearly guarantees one sampling-parallelogram vertex inside the pixel. However this might be too conservative. The next subsection derives the expressions for the absolute coarsest interpolation factor for case 2. As in case 1, these expressions are complex and are derived here only for their theoretical value.

Maximum interpolation factor for case 2

If $V_{0k}V_{2k}$ is longer than shown (1 / sin e_2), $V_{0k}V_{1k}$ has to be shorter since otherwise one can shift the sampling parallelogram such that no vertex remains inside the pixel. If $V_{0k}V_{2k}$ is longer than 1 / sin e_2, the maximum length for $V_{0k}V_{1k}$, can be deducted from **Figure 7.11**. Interestingly enough, it does not depend on $V_{0k}V_{2k}$ and it is given by **Equation 7.12**.

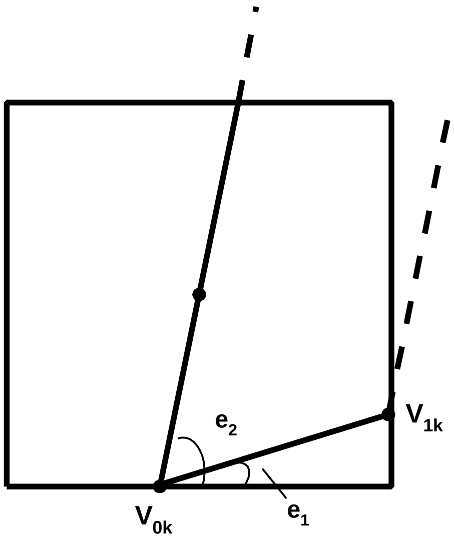


Figure 7.11. Position of sampling parallelogram that determines the maximum length of $V_{0k}V_{1k}$. $V_{0k}V_{2k}$ is longer than 1 / sin e_2.

$$V_{0k}V_{1k} = \frac{\frac{1}{2} + \frac{1}{2\tan e_2}}{\cos e_1}$$

Equation 7.12. Maximum length for $V_{0k}V_{1k}$ if $V_{0k}V_{2k}$ is longer than 1 / sin e_2.

Using **Figure 7.12**, one can derive the maximum length of $V_{0k}V_{2k}$ (**Equation 7.13**).

$$V_{0k}V_{2k} = \frac{1 + V_{0k}V_{1k}\sin e_1}{\sin e_2}$$

$$V_{0k}V_{2k} = \frac{1 + \sin e_1 \frac{\frac{1}{2} + \frac{1}{2\tan e_2}}{\cos e_1}}{\sin e_2}$$

$$V_{0k}V_{2k} = \frac{2\tan e_2 + \tan e_1 \tan e_2 + \tan e_1}{2\sin e_2 \tan e_2}$$

Equation 7.13. Maximum length for $V_{0k}V_{2k}$.

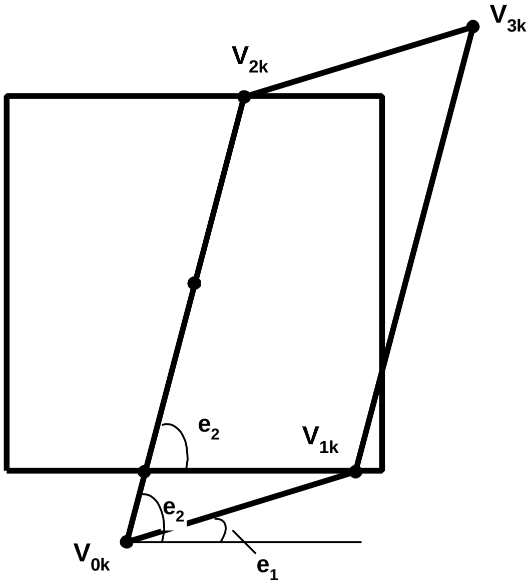


Figure 7.12. Position of sampling parallelogram for the maximum length of $V_{0k}V_{2k}$.

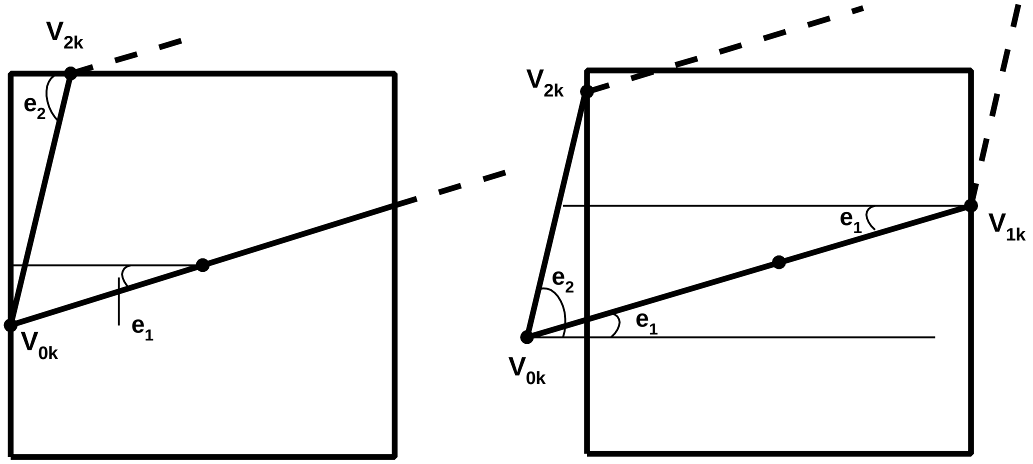


Figure 7.13. Determination of maximum lengths of sides of sampling parallelogram in case 2. If $V_{0k}V_{1k}$ is longer than 1 / cos e_1, $V_{0k}V_{2k}$ must be shorter than 1 / sin e_2. The two cases shown are used to determine the lengths of the two sides of the sampling parallelogram.

Similarly, $V_{0k}V_{1k}$ can be lengthened while $V_{0k}V_{2k}$ is shortened. Using **Figure 7.13**, one can determine the lengths of the sides of the sampling parallelogram (**Equation 7.14**).

$$V_{0k}V_{2k} = \frac{\frac{1}{2}+\frac{\tan e_1}{2}}{\sin e_2}$$

$$V_{0k}V_{1k} = \frac{1}{\cos e_1} + \frac{V_{0k}V_{2k}\cos e_2}{\cos e_1} = \frac{1+\frac{\frac{1}{2}+\frac{\tan e_1}{2}}{\sin e_2}\cos e_2}{\cos e_1} = \frac{1+\frac{1+\tan e_1}{2\tan e_2}}{\cos e_1} = \frac{1+2\tan e_2+\tan e_1}{2\cos e_1\,\tan e_2}$$

Equation 7.14. Lengths of the sides of the sampling parallelogram when $V_{0k}V_{1k}$ is longer than 1 / cos e_1.

I derived the lengths of the sides of the sampling parallelogram for the three extreme cases that can occur (shown in **Figure 7.10**, **Figure 7.12**, **Figure 7.13**; from here on I will call these cases 2.a, 2.b and 2.c). One has to choose the sampling parallelogram such that the product between the lengths of the sides is maximized. Any of the three cases can generate the maximum product as can be seen in **Figure 7.14**. Consequently one has to evaluate the expressions for $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ for the three cases and choose accordingly.

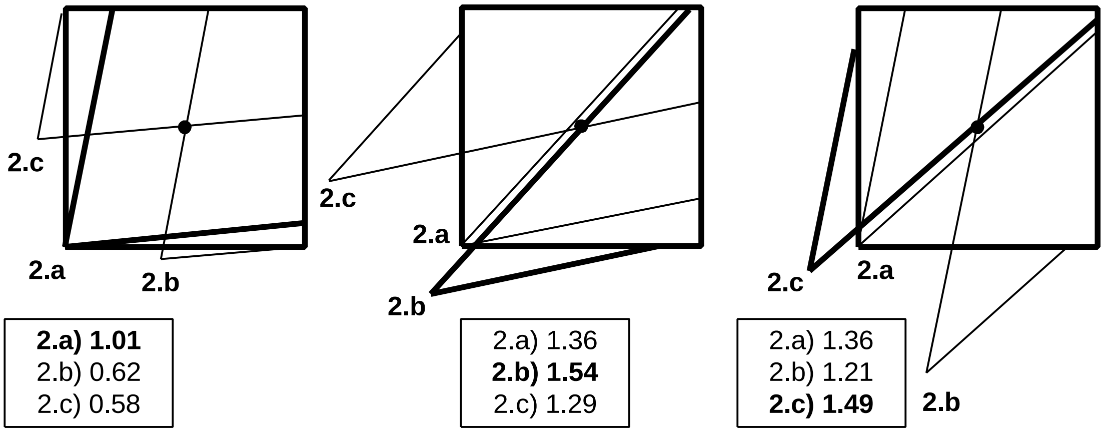


Figure 7.14. Analysis of case 2. Any of the cases 2.a, 2.b, and 2.c can generate the minimum number of samples, according to the orientations of the edges. Three possible edge orientations are shown. For each edge orientation, the sampling-parallelogram is built according to all three cases 2.a, 2.b and 2.c. The case that generates the best interpolation factor is shown with thicker lines. The product between the lengths of the sides of the sampling parallelogram are also given in square pixels.

7.1.2.3 Case 3

In this case, the sampling parallelogram can be inscribed in a pixel (**Figure 7.15**).

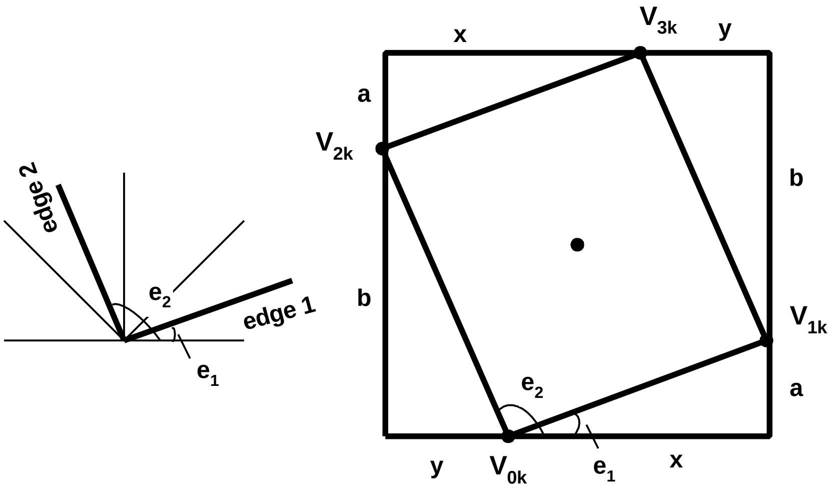


Figure 7.15. Case 3: 0° < e_1 < 45° < 90° < e_2 < 135°.

Computing the lengths of the edges according to **Figure 7.15** ensures one sampling-parallelogram vertex per pixel. I will compute these lengths and then show that for any combination of edge-orientations in case 3, the sampling parallelogram *can* be inscribed in a pixel. This will give a conservative interpolation factor for any edge orientations in case 3. However, the interpolation factor might be over-conservative. If one of the sides of the sampling parallelogram is longer than shown in **Figure 7.15**, the other side must be shorter in order to guarantee a vertex per pixel. For this case (3), the determination of the *minimal* interpolation factor is left as future work.

Using **Figure 7.15** $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ can be computed as follows (**Equation 7.15**).

$$\tan e_1 = \frac{a}{x}, a = x\tan e_1$$

$$-\tan e_2 = \frac{b}{y}, b = -y\tan e_2$$

$$a + b = 1, x\tan e_1 - y\tan e_2 = 1$$

$$x + y = 1, y = 1 - x$$

$$x\tan e_1 - (1 - x)\tan e_2 = 1$$

$$x(\tan e_1 + \tan e_2) = 1 + \tan e_2$$

$$x = \frac{1 + \tan e_2}{\tan e_1 + \tan e_2}$$

$$y = \frac{\tan e_1 - 1}{\tan e_1 + \tan e_2}$$

$$V_{0k}V_{1k} = \frac{x}{\cos e_1} = \frac{1 + \tan e_2}{(\tan e_1 + \tan e_2)\cos e_1}$$

$$V_{0k}V_{2k} = -\frac{y}{\cos e_2} = \frac{-\tan e_1 + 1}{(\tan e_1 + \tan e_2)\cos e_2}$$

Equation 7.15. Computation of the lengths of the sides of the inscribed sampling parallelogram.

The resulting interpolation factor is given in **Equation 7.16**.

$$if_1 = \left\lceil \frac{x_1 - x_0}{\dfrac{1 + \tan e_2}{\tan e_1 + \tan e_2}} \right\rceil$$

$$if_2 = \left\lceil \frac{-(x_2 - x_0)}{\dfrac{\tan e_1 - 1}{\tan e_1 + \tan e_2}} \right\rceil$$

Equation 7.16. Interpolation factor for the inscribed sampling parallelogram. x_0, $x_1$and x_2 are the *x* coordinates of the vertices of the triangle to be rasterized.

In still need to show that there exists an inscribed parallelogram for every combination of edge orientations in case 3; to do so I will show that one can always be constructed using **Equation 7.15**.

Starting from the bottom left corner of the pixel, V_{0k} is selected at a distance *y* (see **Figure 7.15**) along the bottom edge of the pixel. I must show that V_{0k} is inside the bottom pixel edge, regardless of the edge-orientations e_1 and e_2 (**Equation 7.17**).

$$\begin{aligned}
&\textit{given}\\
&0 \le e_1 \le 45^\circ < 90^\circ \le e_2 \le 135^\circ\\
&\textit{show}\\
&0 \le y \le 1\\
&0 \le \frac{\tan e_1 - 1}{\tan e_1 + \tan e_2} \le 1
\end{aligned}$$

Equation 7.17.

The proof is straightforward (**Equation 7.18**).

$$\begin{aligned}
&0 \le \tan e_1 \le 1, \tan e_1 - 1 \le 0\\
&-\infty < \tan e_2 \le -1, \tan e_1 + \tan e_2 \le 0\\
&\textit{so}\\
&0 \le \frac{\tan e_1 - 1}{\tan e_1 + \tan e_2}\\
&\textit{and}\\
&\frac{\tan e_1 - 1}{\tan e_1 + \tan e_2} \le 1
\end{aligned}$$

Equation 7.18.

Starting from the bottom-right corner of the pixel (**Figure 7.15**), V_{1k} is placed on the right edge of the pixel at distance *a*. Again, we must make sure that V_{1k} is within the right edge of the pixel (**Equation 7.19**).

$$\begin{aligned}&\textit{given}\\&0 \le e_1 \le 45^\circ < 90^\circ \le e_2 \le 135^\circ\\&\textit{show}\\&0 \le a \le 1\\&0 \le \frac{(1+\tan e_2)\tan e_1}{\tan e_1 + \tan e_2} \le 1\end{aligned}$$

Equation 7.19

The proof is straightforward. V_{3k} is placed on the top edge at y distance from the top-right corner of the pixel. Since I have just shown that $0 < y < 1$, we know it will be within the top-left and top-right corners of the pixel. Similarly, V_{2k} is placed at distance a below the top-left corner. This concludes the proof that for any combination of edge-orientations, and inscribed sampling parallelogram can be built.

As stated above, the inscribed parallelogram is possibly over-conservative. Note however that it is always possible to choose a different pair of edges such that they are not oriented as required by case 3. Considering **Figure 7.3**, any triangle will have at least two edges in the same quadrant. That pair of edges is known not to be in case 3, which requires the edges to be in different quadrants.

7.1.2.4 Case 4

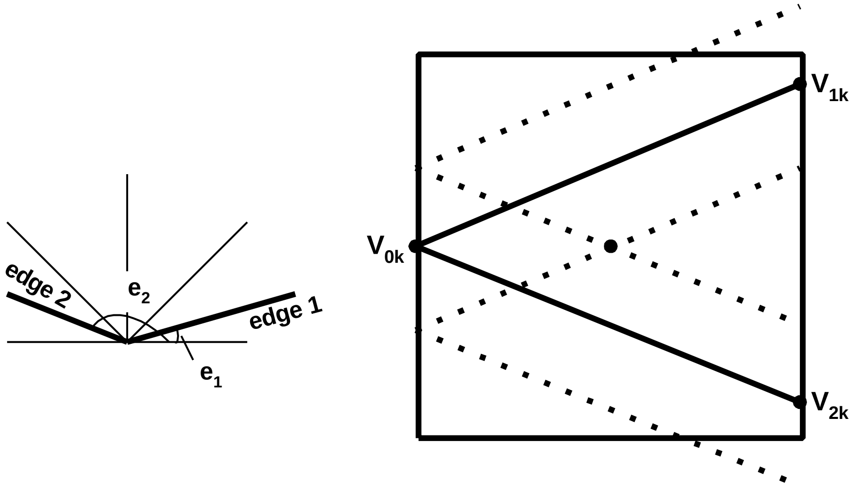


Figure 7.16. Case 4: 0° < e_1 < 45° < 135° < e_2 < 180°.

A conservative choice for the lengths of the sides of the sampling parallelogram is shown in **Figure 7.16**. No matter where the sampling parallelogram is shifted (the only condition is to contain the center of the pixel), at least one vertex will be inside the pixel. The two extreme cases are shown with dotted lines. Since e_1 is less than 45°, even when the parallelogram is shifted as low as possible, both V_{0k} and V_{1k} will be inside the pixel. The argument is similar for the extreme case when the parallelogram is shifted as high as possible. The lengths of the sides and the corresponding interpolation factors are given in **Equation 7.20**.

$$
\begin{aligned}
V_{0k}V_{1k} &= \frac{1}{\cos e_1} \\
V_{0k}V_{2k} &= \frac{1}{\cos e_2} \\
if_1 &= \lceil x_1 - x_0 \rceil \\
if_2 &= \lceil x_2 - x_0 \rceil
\end{aligned}
$$

Equation 7.20. Conservative interpolation factor for case shown in **Figure 7.16**.

The interpolation factor computed as above is possibly over-conservative; although $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ cannot be both longer than the values given in **Equation 7.20**, one of them could possibly be longer. Determining the interpolation factor for minimal sampling is left as future work for this case.

However it can happen that the angles e_1 and e_2 are such that both V_{1k} and V_{2k} are outside the right edge of the pixel (**Figure 7.17**). One has to choose lengths for $V_{0k}V_{1k}$ and $V_{0k}V_{2k}$ that are shorter than the ones prescribed by **Equation 7.20**.

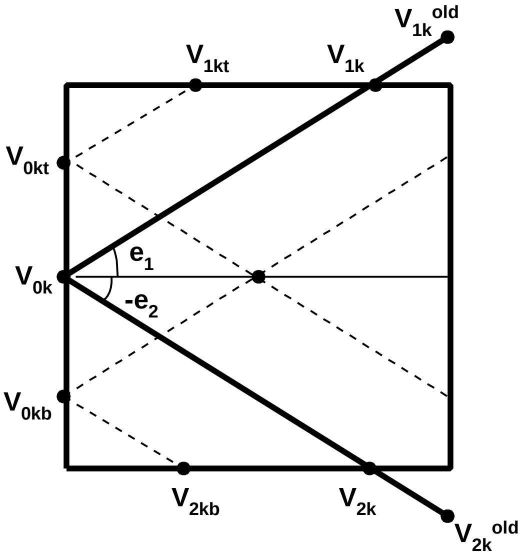


Figure 7.17. Conservative computation of lengths of sampling parallelogram sides for case 4. In the case shown, choosing V_{1k} and V_{2k} according to **Equation 7.20** (V_{1k}^{old} and V_{2k}^{old}) is incorrect. Indeed, if V_{0k} slides to the left, none of the vertices of the sampling-parallelogram will be inside the pixel anymore. A conservative sampling parallelogram is given by limiting the sides to the top and bottom edges of the pixel (V_{1k}, V_{2k}).

For choosing the interpolation factor, V_{0k} can be anywhere between V_{0kt} and V_{0kb}, and one has to choose the position that maximizes the product of the interpolation factors. I leave determining the exact position for future work. **Equation 7.21** derives the interpolation factor for V_{0k} situated at the middle of the left edge of the pixel (as it is in the case shown). This is a reasonable choice since the extreme positions V_{0kt} and V_{0kb} are not favorable (if both edges are at 45 degrees, the length of the "short" side is 0).

$$V_{0k}V_{1k} = \frac{1}{2\sin e_1}, V_{0k}V_{2k} = -\frac{1}{2\sin e_2}$$
$$if_1 = \lceil 2(y_1 - y_0) \rceil, if_2 = \lceil 2(y_0 - y_2) \rceil$$

Equation 7.21. Interpolation factor for case shown in **Figure 7.17**.

In conclusion, the interpolation factor for each edge should be chosen as the maximum of the expressions given in **Equation 7.20** and **Equation 7.21**.

7.1.2.5 Cases 5, 8 and 10

The analysis of these cases is similar to the analysis of case 1.

7.1.2.6 Case 6

The analysis of this case is similar to the analysis of case 4.

7.1.2.7 Case 7

The analysis of this case is similar to the analysis of case 3.

7.1.2.8 Case 9

The analysis of this case is similar to the analysis of case 2.

7.1.3 Implementation considerations and examples

Section 7.1.2 showed that there are essentially four major cases:

- *same half-quadrant* (covering cases 1, 5, 8, 10)
- *same quadrant* (covering cases 2 and 9)
- *inscribed parallelogram* (covering cases 3 and 7)
- *consecutive half-quadrants* (covering cases 4 and 6).

For the same half-quadrant and same-quadrant case I derived the coarsest interpolation factor that guarantees minimal sampling. For the two remaining cases I gave only a *sufficient* interpolation factor, not also necessary. In other words, the interpolation factor is sometimes over-conservative. Determining the best interpolation factor in these last two cases is interesting future work and the analysis should be similar to the analysis of the other cases.

From a practical standpoint however, it might not pay off using the absolute coarsest interpolation factors since their evaluation requires quite a bit of computation[33] and one of the major reasons I propose forward rasterization instead of the classic method is reduced operation count at setup. Using the *sufficient* interpolation factors (**Equation 7.22**) requires only a small amount of computation. Note that no square-root computation is required since no Euclidean distance is needed

[33] as can be seen in **Equation 7.13** and **Equation 7.14** for example

$$
\begin{aligned}
&same-quadrant-case\\
&same-half-quadrant-case\\
&if_1=\lceil \max(|x_0-x_1|,|y_0-y_1|)\rceil\\
&if_2=\lceil \max(|x_0-x_2|,|y_0-y_2|)\rceil\\
&cons.-half-quadrants-case\\
&if_1=\lceil \min(\max(2|x_0-x_1|,|y_0-y_1|),\max(|x_0-x_1|,2|y_0-y_1|))\rceil\\
&if_2=\lceil \min(\max(2|x_0-x_2|,|y_0-y_2|),\max(|x_0-x_2|,2|y_0-y_2|))\rceil
\end{aligned}
$$

Equation 7.22. Sufficient interpolation factor for all cases except inscribed parallelogram. The baricentric interpolation is done parallel to edges V_0V_1 and V_0V_2.

Recall that the inscribed-parallelogram case can be avoided. So all an implementation has to do is to compute the interpolation factor for each edge pair and then choose the pair with the largest product, while making sure that that pair is not in the inscribed-parallelogram case. Moreover, most of the differences in **Equation 7.22** are required by the rasterization itself so they are not an additional burden.

Figure 7.18, **Figure 7.19**, and

Figure 7.20 show a triangle forward rasterized using the **Equation 7.22**.

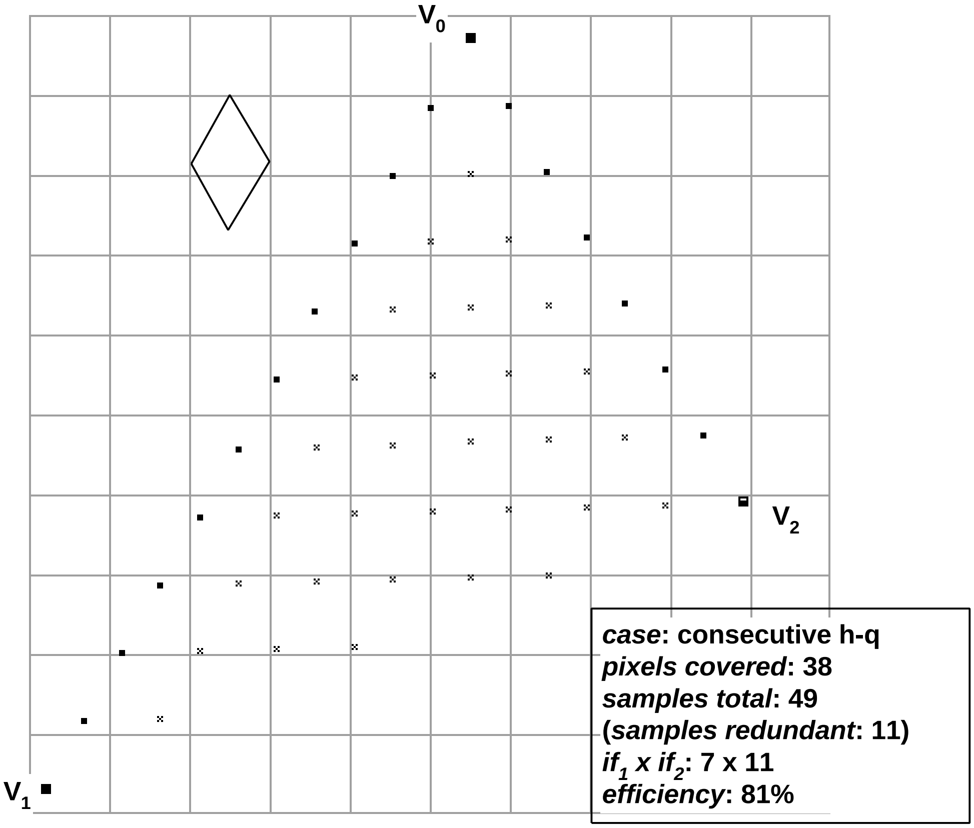


Figure 7.18. Example of forward rasterization: consecutive half-quadrants case. The pair of edges that gives the smallest number of samples is (V_0V_1, V_0V_2). The pair is in the consecutive-half-quadrants case. The efficiency is computed as the area of the sampling parallelogram, in pixels.

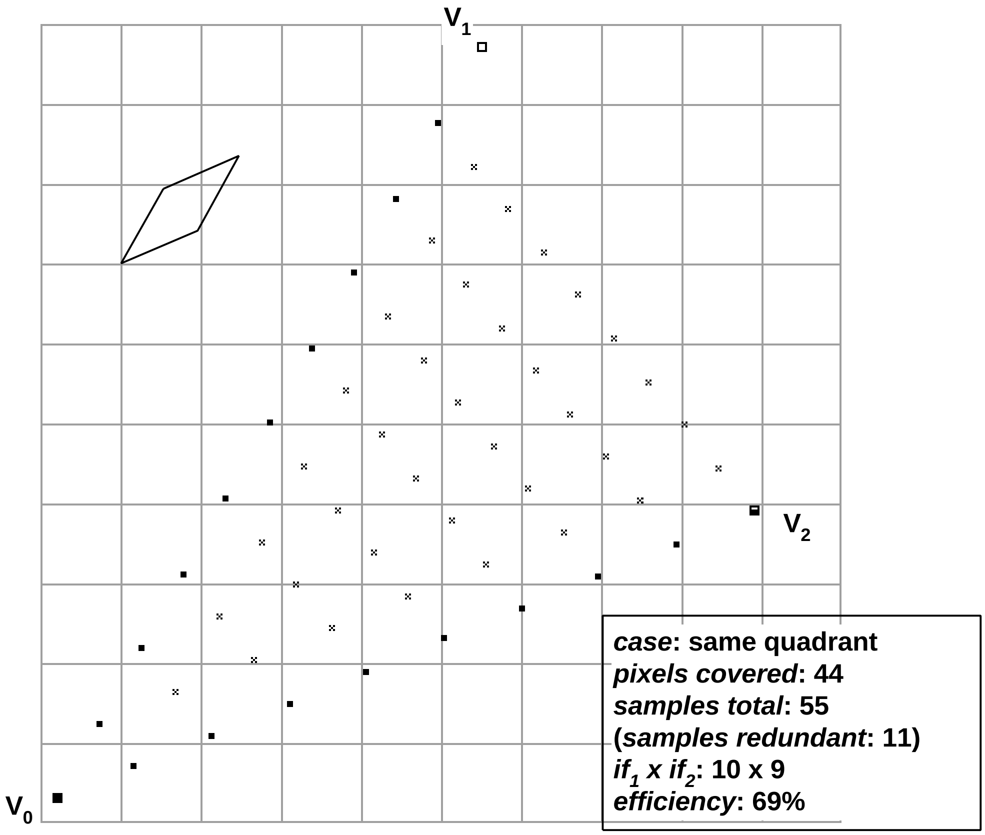


Figure 7.19. Example of forward rasterization: same-quadrant case. The three vertices of the sampling-parallelogram shown are not exactly on the edges of the pixel. This is due to the ceiling operation used when computing the interpolation factor. Consequently, the sides of the sampling parallelogram are correctly a little bit shorter than the absolute maximums prescribed by the equations derived assuming the vertices *on* the pixel boundaries.

7.1.4 Early discarding of redundant samples

A simple and effective way of further reducing the redundant samples is to save the pixel to which the previous sample landed and not to generate a new sample unless it lands in a new pixel. In order to increase the efficiency of the method, one should choose in the inner loop the interpolation direction that generates the most redundancy. If the next sample is to land to the same pixel, it is more likely to happen if the step from one sample to the next is shorter.

In baricentric interpolation there are four interpolation directions possible: one for each edge direction and one for each diagonal direction of the sampling parallelogram. Generating samples on scan-lines parallel to the diagonal is not more expensive computationally, and can be useful since the shortest diagonal might be shorter than the shortest side, which allows the early discarding of more samples.

Out of the various cases that determine the sampling parallelogram, same quadrant and same half-quadrant are the most appropriate for this early discarding mechanism. In the case consecutive half-quadrants it can happen that both sides are too short to span an entire pixel (**Figure 7.17**). The inscribed parallelogram case is obviously worse.

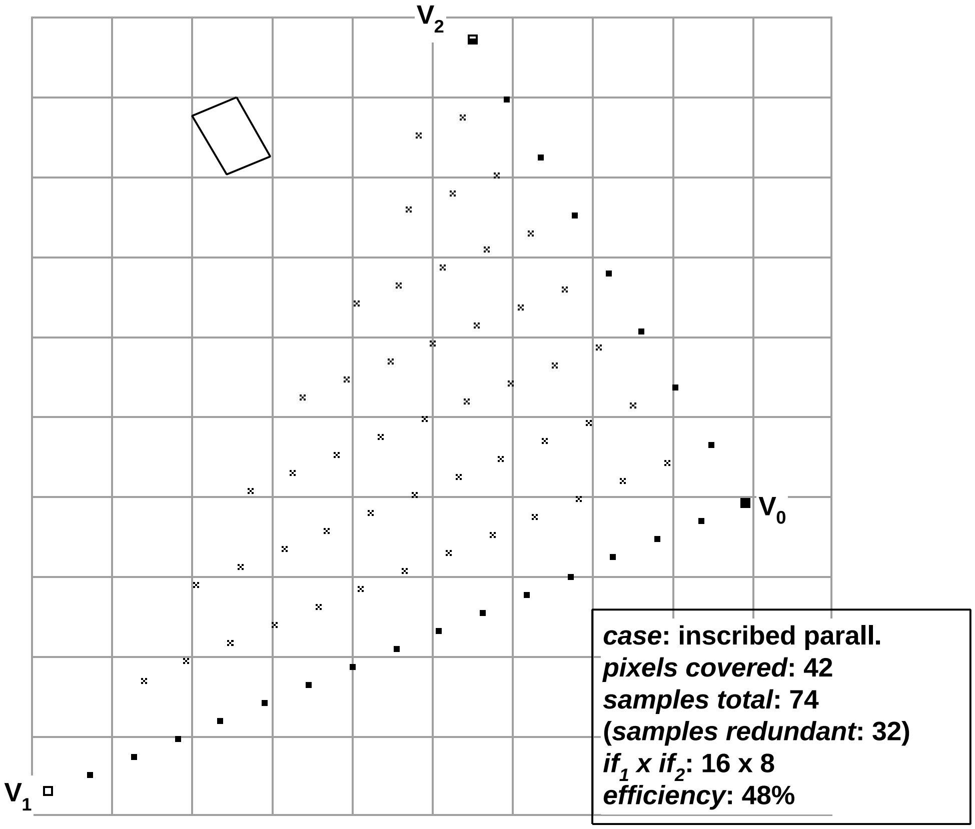


Figure 7.20. Example of forward rasterization: inscribed-parallelogram case. A large number of samples are generated, since the area of the inscribed parallelogram is less than half a pixel. Fortunately there is always a pair of edges that is not classified in this case.

In the same half-quadrant or same quadrant cases, the sides of the sampling parallelogram span one pixel (**Figure 7.21**). As sample $S_{i+1,\ j+1}$ is generated, the only sample with whom it could possibly compete for the same pixel is $S_{i+1,j}$. Sample $S_{i,j}$ is $1\text{-}\varepsilon$ pixels away in the y-direction by the way the interpolation factor was chosen and so it cannot land at the same pixel. Similarly, sample $S_{i,j+1}$ is $1\text{-}\varepsilon$ pixels away in the x-direction and sample $S_{i,j+2}$ is more than one pixel away in the x-direction.

As I pointed out before, ε is due to the ceiling operation used in computing the interpolation factor. Not using the ceiling operation would have the advantage of early discarding all redundant samples. However it has two disadvantages:

- vertices V_1 and V_2 will not be hit, and,
- the inverses of the interpolation factors are more costly since they involve real numbers (not small integers).

Since even with the ceiling operation the number of redundant samples that escape early discarding is extremely low, I believe that it does not pay off to use real numbers for the interpolation factor.

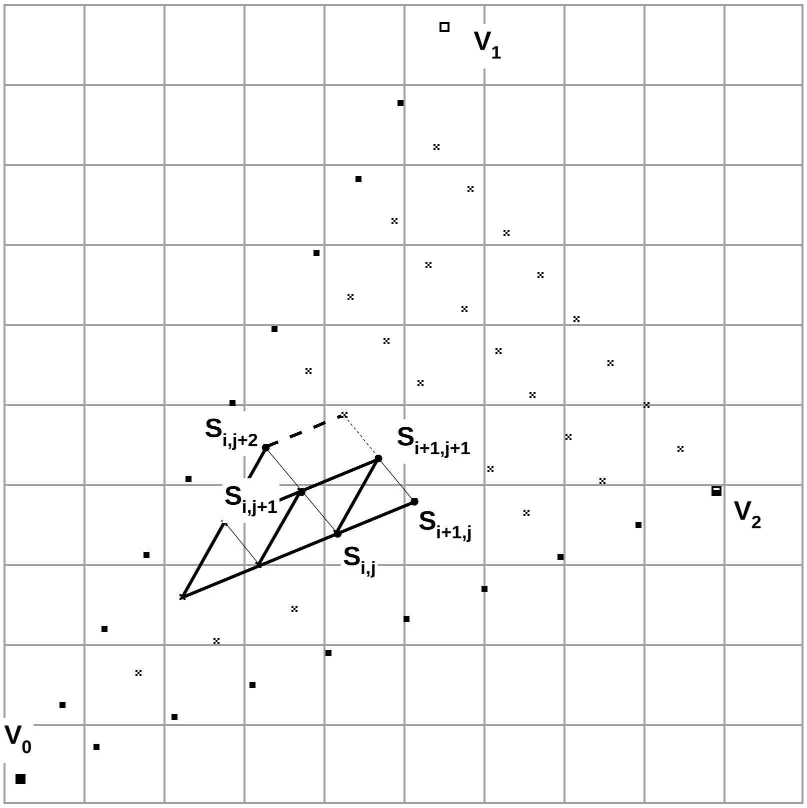


Figure 7.21. Early discarding of redundant samples. In the same quadrant case, generating samples on a direction parallel to the short diagonal of the sampling parallelogram enables early discarding of virtually all redundant samples.

Finally, every triangle has at least a pair of edges that is in the same quadrant or same-half-quadrant case. This is obvious since at least one of the two y+ quadrants has to take two or more of the three edges of the triangle.

In conclusion, there are virtually no redundant samples if:

- the pair of edges in the same-quadrant or same-half-quadrant case are used,
- the samples are generated on the small diagonal of the sampling parallelogram (**Figure 7.22**),
- a new sample is generated only if it lands at a different pixel than the previous sample.

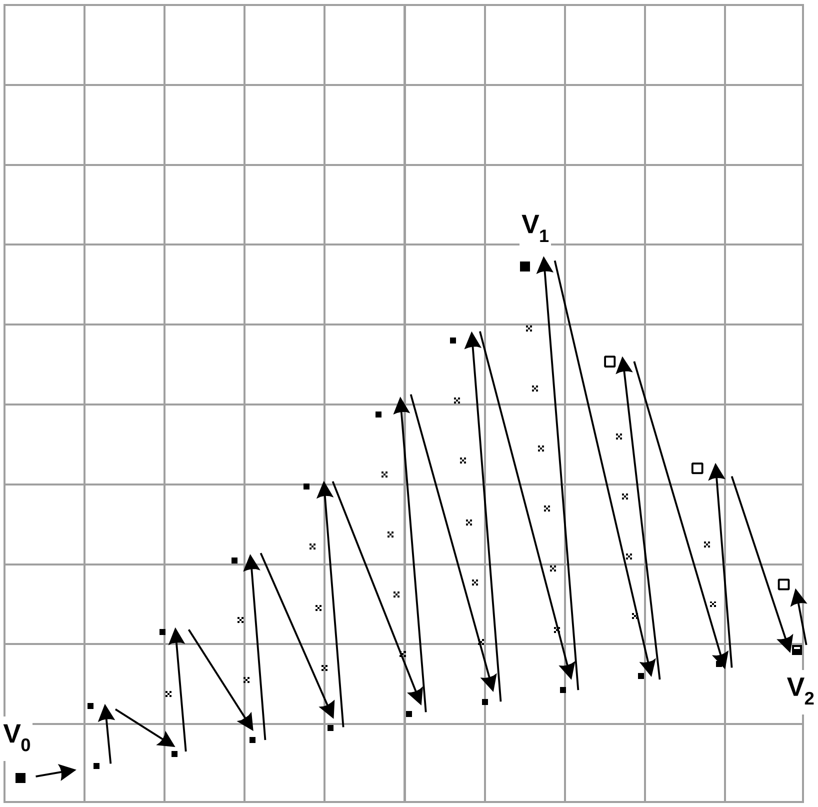


Figure 7.22. Example of interpolation along the short diagonal of the sampling parallelogram. The interpolation factor is 7x10. The interpolation starts from V_0 and all subsequent scan-lines start from V_0V_2. For the first 8 scan-lines (counting V_0 also), the number of samples per scan-line increases from 1 to 8. The 8th sample of the 8th scan-line is V_1. For the remaining 3 scan-lines, a third-edge (V_1V_2) sidedness test is needed to know when to stop the scan-line (the little squares are samples for which the sidedness test failed).

Chapter 8 Discussion and Future Work

8.1 Summary and discussion

8.1.1 Summary of proof of thesis statement

In this section I will review the research results that helped prove the thesis statement of this dissertation:

> ***Image-based rendering by warping using forward rasterization, which means parameter-space interpolation and offset reconstruction, produces high-quality images and can be efficiently implemented in hardware.***

8.1.1.1 Forward rasterization for IBRW

The first part of the proof of the thesis statement is to prove that the *forward rasterization algorithm* produces high-quality images. The dissertation describes the algorithm in detail.

First, the algorithm correctly resolves visibility. Unlike previous forward-mapping reconstruction algorithms, surface continuity is maintained, which guarantees that after the visibility stage only samples of *visible* surfaces persist in the warpbuffer.

Then, the final image is reconstructed from the visible samples. *2x2* supersampling of the warpbuffer provides four samples per output pixel. Truncation errors inherent to forward-mapping techniques are reduced below noticeable level by using an inexpensive pair of two-bit offsets that accurately locate the sample within the visibility cell, which is the warpbuffer location. The offset-reconstruction method relies on the observation that in order to reduce the aliasing due to insufficient precision at locating the samples, it is not necessary to resort to very expensive finer resolutions.

The final image is reconstructed from enough guaranteed-visible, accurately-located samples and in the field of computer graphics this is known to produce high-quality results. The expectations were empirically confirmed as the images produced by the algorithm were judged of high-quality by several researchers in the field.

8.1.1.2 Hardware implementation: the WarpEngine

The second claim in the thesis statement is that the algorithm can be efficiently implemented in hardware. One aspect of the efficiency is a reduced operation count. The forward-rasterization algorithm does not require expensive setup computations as no backward-mapping from image plane to parameter space is needed. When compared to the mesh IBRW-reconstruction method, the setup cost is reduced four times.

The dissertation presents the WarpEngine, which is a hardware architecture based on the forward-rasterization algorithm. The WarpEngine promises high-resolution output images at interactive rates and it does so efficiently since it takes advantage of the locality characteristic to depth images.

Locality is used to inexpensively and accurately predict the screen regions to which groups of primitives (depth-and-color samples) map, enabling sort-first high-level parallelism. Sort-first allows for on-chip warpbuffer, making the bandwidth requirements manageable. In retained mode, sort-first has the advantage of frame-to frame coherence.

Locality also implies that nearby samples can be interpolated with the same interpolation factor, enabling SIMD low-level parallelism.

8.1.1.3 The vacuum-buffer sample-selection algorithm

The thesis statement indirectly implies that IBRW *can* produce high-quality images, making abstraction of the algorithm used. The first problem that must be solved is sample selection, which implies determining for every desired frame a sufficient and reasonably-sized set of input color-and-depth samples that when processed by the IBRW algorithm produce a high-quality image.

The dissertation presents the *vacuum-buffer algorithm*, which, unlike previous sample-selection methods, provides a measure of whether visible surfaces could have potentially been missed. The algorithm uses the current view, thus must be executed at every frame. The vacuum-buffer algorithm is too laborious to run in real-time on general-purpose processors. However, it could run on polygon-rendering hardware with an enhanced z-buffer. Although the algorithm is not practical in absence of such hardware, I believe it is important since it is the first that attempts to guarantee that no surfaces are missed.

This concludes the summary of the proof of the thesis statement. Before I summarize the other research results presented in the dissertation, I would like to answer a higher level question: is there any alternative to IBRW?

8.1.2 Mesh-simplification: a possible alternative to IBRW

The depth-images are in fact triangle meshes. Rasterizing all these triangles conventionally is expensive but a possibility is to simplify the triangle meshes; this reduces the polygon count, and the (ever-increasing) performance limits of conventional graphics hardware would not be exceeded. This is a viable alternative and other members of our UNC graphics group are investigating it.

However, simplifying these meshes is not a trivial task, especially in the presence of acquisition errors. Also, simplified meshes are less well suited to real-time depth updates because of the pre-processing required. One could argue that simplification could be done in real-time with the help of hardware but I speculate that the algorithms involved are less suited for hardware implementation.

The essence of the problem of rendering from samples is that conventional rasterization through backward mapping is very inefficient since the three samples defining the interpolation are close together. There are two possible solutions:

- go back to primitive sizes for which backward mapping is efficient, and this is exactly what mesh-simplification does, or,
- avoid backward mapping altogether by forward-mapping the samples, and this is what the splatting methods were trying to achieve.

The splatting methods do not produce completely satisfying results, but I believe that the forward-mapping rendering method introduced by this dissertation overcomes most of the problems associated with splatting. In conclusion, I think that the advantage of mesh-simplification has been reduced to the benefit of readily available hardware.

8.1.3 Forward rasterization for polygon rendering

Since the triangles in polygonal models become smaller and smaller as the complexity of the scenes increase, polygon rendering could also benefit from reducing the setup cost. In the last chapter of the dissertation I presented my first efforts in this direction.

For polygon-rendering, surface shading of samples is expensive, frequently involving one or several texture lookups per sample. Thus it is crucial to reduce the number of redundant samples[34] generated by forward-rasterization. I sketched an algorithm designed around the requirement of avoiding redundant samples. The samples are generated by baricentric interpolation, which generates samples parallel to two of the edges of the triangle. The quads formed by neighboring samples are identical parallelograms and with this constraint a coarser interpolation factor can be used while all pixels covered by the triangle are still hit. Moreover, if the right pair of edges is chosen, virtually all redundant samples can be discarded early (before shading). Many of the algorithm details are yet to be worked out, including a rigorous operation count, but the algorithm seems promising. A possible advantage to this method that must be fully explored is better temporal antialiasing.

[34] samples that land at the same pixel

8.2 Future work

8.2.1 WarpEngine

There is still work that must be done before chip-layout of WarpEngine can begin. Several parts of the architecture must be specified or specified in more detail. These include, but are not limited to, the Network Interface and the Reconstruction Buffer.

For my dissertation research I concentrated on static scenes. The implications of objects moving in the scene are not explored. The second-generation WarpEngine should also probably support some form of re-lighting. The Warp Array could, in theory, be programmed to do it, but at the cost of longer processing times for tiles. From our estimates, the Warp Array requires an insignificant fraction of the total chip area. This suggests that having several Warp Arrays per WarpEngine node does not imply a substantial additional cost, from the chip-area point of view.

8.2.2 Vacuum-buffer algorithm

8.2.2.1 Better sample-selection

In order to make the vacuum-buffer algorithm practical, it must be accelerated in hardware. Although such hardware is similar to current polygon-rendering hardware, an additional capability is required, namely the ability to store and update several z values per z-buffer location.

There are some improvements that could be made to the algorithm. The current version of the algorithm processes nearby reference images in the order of increasing distance from the desired camera position. The residual amount of vacuum should decrease more rapidly if the algorithm:

- first processes the reference image that sees the desired view location and has a view direction closest to the desired image view (this eliminates all the vacuum close to the desired view camera), and
- then uses the vacuum accounting tree (VAT) to localize the remaining vacuum and then select for processing reference images that see that vacuum.

This reference image ordering might allow spacing the reference images even farther apart.

The vacuum-buffer algorithm offers a convenient way of analyzing which subvolumes of the current view frustum are already determined by the reference images that have been processed. This could be useful in another application than sample selection.

8.2.2.2 A different use of the vacuum buffer: determining sampling locations

An obvious goal in computing the locations where the reference images should be placed is to acquire as few images as possible while minimizing disocclusion errors. This is the well known *next-best-view* problem ([Connoly85], [Maver93] and many others).

I speculate on an iterative process based on the vacuum-buffer algorithm that automatically produces a set of candidate sampling locations.

First, the scene is subdivided according to a regular 3D grid. A cell of this grid would be a cube of side 0.5m for a room or 20m for a town. Complete panoramas (6 faces of a cube) should be acquired at each grid node. Then, for each cell, the vacuum-buffer algorithm is executed at locations that form a more refined grid, with nodes that are let's say .1 m apart. The reference images used are the reference images acquired at the corners of the current cell and the cells adjacent to the current cell. The node of the refined grid at which the most vacuum remains undetermined is a new sampling location. The residual vacuum at the refined grid locations is computed again. Again the location with the most undetermined vacuum becomes a new sampling location. The process stops when the budget of sampling locations per cell is reached or when the maximum residual vacuum value dips below a tolerable threshold.

The greedy approach described above does not provide the optimal solution but it is fully automatic and the solution might be good enough.

8.2.3 Forward-rasterization for polygon-rendering

When using baricentric interpolation, samples are generated parallel to two edges of the triangle to be rasterized. The expression of the coarsest conservative interpolation factor depends on the orientation of the two edges. I did not derive the expressions for the absolute best interpolation factor for every case. As I pointed out, deriving these expressions could be just a theoretical exercise since they might be too costly to evaluate. The only way to find out is to derive them and perform a careful count of number of operations required.

Future work that also must be done is an analysis of the sampling of the third edge. The last sample of the scan-line will not land on the third edge since the increment is constant from scan-line to scan-line. The end of a scan-line is detected by performing a sidedness test with respect to the third edge. One concern is that the third edge is jaggy but a more serious concern is that surface continuity is not maintained between adjacent triangles. A possible solution is probably to sample the third edge with the finest of the two interpolation factors, as implied by the triangles that share it.

Also one must analyze the behavior of the forward-rasterization algorithm for the case of degenerate triangles.

Small triangles are a serious challenge for classic backward-mapped rasterization because of numerical error. The parameter-plane equations involve a division by the area of the triangle and it can create large errors if the area is very small. The PixelFlow setup code tests for the area and if too small, it discards the triangle altogether. The forward rasterization algorithm does not require any division[35], which implies better numerical stability. It seems that in the case of tiny triangles, the forward algorithm would generate three samples corresponding to the three vertices of the triangle, that converge to one point as the size of the triangle decreases. Of course there is the problem of redundant samples since a triangle will generate at least three samples, regardless of how small it is.

[35] the inverses of the two interpolation factors do not count since the interpolation factors are integers greater or equal to one

Slim triangles, that is triangles with a small angle, are also rasterized differently by the two methods. The classic method generates a dotted line, as centers of pixels are or are not included in the slim triangle. The forward method generates a solid line, which might be a better result. Again there is the problem of redundant samples, especially if the pair of edges used in the baricentric interpolation is composed of the edges with the small angle.

A possible problem of the forward-rasterization algorithm is the problem of potential visibility artifacts due to not testing for visibility along the exact same ray. This potential problem was reported for the IBRW case, but because of the specifics of modeling with images, it is extremely unlikely to occur in that case. For polygon rendering the problem seems more likely to occur. Tests must be conducted on complex scenes to assess the likelihood of the visibility artifacts. If the need exists, the problem can be solved by computing z the classic way. Computing the parameters of the z-plane allows for testing for visibility along the same ray, avoiding the potential visibility artifacts. The cost is of course a slightly more expensive setup. If the number of parameters is large, the additional cost might be negligible.

8.2.4 Other future work

A depth camera that returns accurate depth maps in real time would enable the extraordinary application of 3D television, when coupled with a WarpEngine system. This makes developing such a "magic" depth camera very exciting future work. Maybe existing depth-extraction techniques can be sufficiently accelerated using near future computing technology. Maybe radically new depth-extraction methods need to be developed but it doesn't seem overoptimistic to hope to see such a depth camera in the not too distant future.

REFERENCES

[Adelson91] Adelson E., Bergen J., "The Plenoptic Function and the Elements of Early Vision", *Computation Models of Visual Processing*, Landy M. and Movshon J. editors, MIT Press, Cambridge, ch. 1, (1991).

[Akeley93] Akeley K., "RealityEngine Graphics", *Proc. of SIGGRAPH '93*, 109-116 (1993).

[Aliaga97] Aliaga D., and Lastra A, "Architectural Walkthroughs Using Portal Textures". *Proc. of IEEE Visualization '97*, 355-362 (1997).

[Aliaga99] Aliaga D. and Lastra A., "Automatic Image Placement to Provide a Guaranteed Frame Rate", *Proc. of SIGGRAPH '99*, 307-316 (1999).

[Bastos97] Bastos R., Goslin M., and Zhang H., "Efficient Radiosity Rendering using Textures and Bicubic Reconstruction". *ACM-SIGGRAPH Symposium on Interactive 3D Graphics*, 27-30 (1997).

[Beraldin92] Beraldin J.-A., Rioux M., Blais F., Domey J., and Cournoyer L., "Registered Range and Intensity Imaging at 10-Mega Samples per Second", *Opt. Eng.*, **31**(1): 88-94 (1992).

[Blinn76] Blinn J. F., and Newell M. E., "Texture and Reflection in Computer Generated Images", *CACM, 19(10)*, 542-547 (1976).

[Chang99] Chang C., Bishop G., Lastra A., "LDI Tree: A Hierarchical Representation for Image-based Rendering", *Proc. of SIGGRAPH '99*, 291-298, (1999).

[Chen95] Chen S., "Quicktime VR - An Image-Based Approach to Virtual Environment Navigation", *Proc. of SIGGRAPH '95*, 29-38 (1995).

[Clark82] Clark J., "The Geometry Engine: A VLSI Geometry System for Graphics", *Computer Graphics, 16(3), (Proceedings of SIGGRAPH '82)*, 127-133 (1982).

[Connolly85] Connolly, "The determination of next best views", *Proc. of EEE International Conference on Robotics and Automation*, pages 432-435, (1985).

[Cyra] The Cyrax System, in http://www.cyra.com/.

[Debevec96] Debevec P., Taylor C., and Malik J., "Modeling and Rendering Architecture from Photographs: A Hybrid Geometry and Image-Based Approach", *Proc. of SIGGRAPH '96*, 11-20 (1996).

[Debevec98] Debevec P., "Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography", *Proc. of SIGGRAPH '98*, 189-198 (1998).

[Demetrescu85] Demetrescu S., "High Speed Image Rasterization Using Scan Line Access Memories", *Proc. of the 1985 Chapel Hill Conference on VLSI, Rockville, MD*, Computer Science Press, 221-243 (1985).

[Ellsworth97] Ellsworth D., "Polygon Rendering for Interactive Visualization on Multicomputers", *Doctoral Dissertation, CS UNC Chapel Hill*, (1997).

[Eyles97] Eyles J., Molnar S., Poulton J., Greer T., Lastra A., England N., and Westover L., "PixelFlow: The Realization," *Proc. of the 1997 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 57-68 (1997).

[Fuchs89] Fuchs H., Poulton J., Eyles J., Greer T., Goldfeather J., Ellsworth D., Molnar S., Turk G., Tebbs B., and Israel L., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Proc of SIGGRAPH '89*, 79-88 (1989).

[Gortler96] Gortler S., Grzeszczuk R., Szeliski R., and Cohen M., "The Lumigraph", *Proc. of SIGGRAPH '96*, 43-54 (1996).

[Grossman98] Grossman J., and Dally W., "Point Sample Rendering", In *Rendering Techniques '98*, 181-1921 (1998).

[Haeberli90] Haeberli P., "Paint by Numbers: Abstract Image Representations", *Proc. of SIGGRAPH '90*, 207-214, (1990).

[Kanade99] Kanade T., Rander P., Vedula S., and Saito H., “Virtualized Reality: Digitizing a 3D Time-Varying Event As Is and in Real Time”, *Mixed Reality, Merging Real and Virtual Worlds*, Y. Ohta and H. Tamura, Editors. Springer-Verlag. p. 41-57 (1999).

[K2T] *Scene Modeler*, http://www.k2t.com/.

[Larson98] Larson G., "The Holodeck: A Parallel Ray-Caching System", *Proc. of Eurographics Workshop on Parallel Graphics and Visualization*, (1998).

[Lengyel97] Lengyel J., and Snyder J., "Rendering with Coherent Layers", *Proc. of SIGGRAPH '97*, 233-242 (1997).

[Levoy85] Levoy M., and Whitted T., "The Use of Points as a Display Primitive", *UNC Computer Science Technichal Report TR85-022*, (1985).

[Levoy96] Levoy M. and Hanrahan P., “Light Field Rendering”, *Proc. of SIGGRAPH ’96*, 31-42 (1996).

[Maciel95] Maciel P, and Shirley P., "Visual Navigation of Large Environments Using Textured Clusters", *Proc. of Symposium on Interactive Graphics*, 95-102 (1995).

[Mark97] Mark W., McMillan L., and Bishop G., "Post-Rendering 3D Warping", *1997 Symposium on Interactive 3D Graphics*, 7-16 (1997).

[Mark99] Mark W., *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*, PhD thesis, University of North Carolina at Chapel Hill, 1999.

[Maver93] Maver and Bajcsy, “Occlusions as a guide for planning the next view”, *IEEE Transactions on Pattern Analysis and Mmachine Intelligence*, 15(5):417-433 (1993).

[McAllister99] McAllister, D., Nyland L., Popescu V., Lastra A., McCue C., "Real-Time Rendering of Real-World Environments", *Rendering Techniques '99, Proc. Eurographics Workshop on Rendering*, 145-160, (1999).

[McMillan95] McMillan L. and Bishop G., “Plenoptic Modeling: An Image-Based Rendering System”, *Proc. of SIGGRAPH ’95*, 39-46 (1995).

[McMillan96] McMillan L., "Computing Visibility Without Depth", *UNC Computer Science Technical Report TR95-047*, University of North Carolina, (1995).

[McMillan97] McMillan L., *An Image-Based Approach to Three-Dimensional Computer Graphics*, PhD thesis, University of North Carolina at Chapel Hill, 1997.

[Minolta] *Minolta 3D 1500*, in http://www.minolta3d.com/.

[Molnar91] Molnar, S. "Efficient Supersampling Antialiasing for High-performance Architectures," *UNC-CS Technical Report TR91-023*, (1991).

[Molnar92] Molnar S., Eyles J., and Poulton J., “PixelFlow: High-speed Rendering using Image Composition”, *Proc. of SIGGRAPH ’92*, 231-240 (1992).

[Molnar94] Molnar S., Cox M., Ellsworth D., and Fuchs H., "A Sorting Classification of Parallel Rendering", *IEEE Computer Graphics and Aplications*, 14(4), 23-32 (1994)

[Montrym97] Montrym J., Baum D., Dignam D., and Migdal C., “InfiniteReality: A Real-Time Graphics System”, *Proc. of SIGGRAPH ’97*, 293-302 (1997).

[Mueller95] Mueller C., “The Sort-First Rendering Architecture for High-Performance Graphics”, *1995 Symposium on Interactive 3D Graphics*, 75-84 (1995).

[Mueller97] Mueller C., “Hierarchical Graphics Databases in Sort-First”, *1997 Parallel Rendering Symposium*, 49-57 (1997).

[Mueller00] Mueller C., “The Sort-First Architecture for Real-Time Image Generation”, *Doctoral Dissertation, CS UNC Chapel Hill*, (2000).

[Nyland99] Nyland L., McAllister D., Popescu V., McCue C., Lastra A., Rademacher P., Oliveira M., Bishop G., Meenakshisundaram G., Cutts M., and Fuchs H, "The Impact of Dense Range Data on Computer Graphics", *Proc. of Multi-View Modeling and Analysis Workshop (MVIEW99)*, 1999.

[Olano98] Olano M. and Lastra A., “A Shading Language on Graphics Hardware: The PixelFlow Shading System”, *Proc. of SIGGRAPH '98*, (1998).

[Oliveira00] Oliveira M., Bishop G., and McAllister D., "Relief Texture Mapping", *Proc. of SIGGRAPH '00*, 359-368 (2000).

[Pfister00] Pfister H., Zwicker M., Van Baar J., and Gross M., "Surfels: Surface Elements as Rendering Primitives", *Proc. of SIGGRAPH '98*, 335-342 (2000).

[Popescu98] Popescu V., Lastra A., Aliaga D., Oliveira M., "Efficient Warping for Architectural Walkthroughs using Layered Depth Images", *Proc. of IEEE Visualization '98*, 211 - 215 (1998).

[Popescu99] Popescu V., and Lastra A., "High Quality 3D Image Warping by Separating Visibility from Reconstruction", *UNC Computer Science Technical Report TR99-017*, University of North Carolina, (1999).

[Popescu00] Popescu V., Eyles J., Lastra A., Steinhurst J., England N., Nyland L., "The WarpEngine: An Architecture for the Post-Polygonal Age", *Proc. of SIGGRAPH '00*, 433-442 (2000).

[Rafferty98] Rafferty M., Aliaga D., and Lastra A., "3D Image Warping in Architectural Walkthroughs", *Proceedings of VRAIS '98*, 228-233 (1998).

[Regan99] Regan M., Miller G., Rubin S., and Kogelnik C., "A Real Time Low-Latency Hardware Light-Field Renderer", *Proc. of SIGGRAPH '99*, 287-290 (1999).

[Rusinkiewicz00] Rusinkiewicz S., and Levoy M., "QSplat: A Multiresolution Point Rendering System for Large Meshes", *Proc. of SIGGRAPH '00*, 343-352 (2000).

[Schaufler96] Schaufler G., and Stürzlinger W., "A Three Dimensional Image Cache for Virtual Reality", *Proceedings of EUROGRAPHICS '96*, 227-236 (1996).

[Schilling91] Schilling, A., "A New Simple And Efficient Antialiasing With Subpixel Masks", *Proc. of SIGGRAPH '91*, 133-141 (1991).

[Seitz96] Seitz S. and Dyer C., "View Morphing: Synthesizing 3D Metamorphoses Using Image Transforms", *Proc. of SIGGRAPH '96*, 21-30 (1996).

[Shade98] Shade J., Gortler S., He L., and Szeliski R., "Layered Depth Images", *Proc. of SIGGRAPH '98*, 231-242 (1998).

[Torborg96] Torborg J. and Kajiya J., "Talisman: Commodity Real-time 3D Graphics for the PC", *Proc. of SIGGRAPH '96*, 353-364 (1996).

[Westover90] Westover L., "Footprint Evaluation for Volume Rendering", *Proc. of SIGGRAPH '90*, 367-376 (1990).

[Williams83] Williams L., "Pyramidal Parametrics", *Proc. of SIGGRAPH '83*, 1-11 (1983).

[Yu98] Yu Y., and Malik J., "Recovering Photometric Properties of Architectural Scenes from Photographs", *Proc. of SIGGRAPH '98*, 207-218 (1998).