Diplomarbeit

Efficient Compression and Rendering in a Client-Server setting

Fabian Giesen

Mai 2008

Vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, und sämtliche Zitate markiert habe.

Königswinter, den 23.05.2008

Contents

Introduction 5

1. Previous work 9
- 1.1. Image-based Rendering . . . 9
- 1.2. Warping of planar depth images . . . 9
- 1.3. Warping-based client-server rendering . . . 10
- 1.4. Warping and compression . . . 13
- 1.5. Video coding . . . 15

2. Warping of Depth Images for Video Compression 19
- 2.1. Warping of depth images . . . 20
  - 2.1.1. The warping equation . . . 20
  - 2.1.2. Limitations of warping . . . 22
  - 2.1.3. Reconstruction . . . 23
  - 2.1.4. Inverse warping . . . 24
- 2.2. Fast warping for prediction in video coding . . . 24
- 2.3. Evaluating the efficiency of warping as a prediction model . . . 29
  - 2.3.1. Prediction efficiency in ideal circumstances . . . 29
  - 2.3.2. Relative cost of disparity data and motion vectors . . . 35

3. Improving video compression and rendering 39
- 3.1. Warping-based motion estimation . . . 39
  - 3.1.1. Implementation . . . 41
  - 3.1.2. Results . . . 43
  - 3.1.3. Conclusions . . . 49
- 3.2. Accelerated rendering using warping and related methods . . . 49
  - 3.2.1. Per-block confidence information . . . 51
  - 3.2.2. Relevance to server-side rendering . . . 53

Conclusions and future work 55

A. Source code 57
- A.1. Build process . . . 58

B. Complete results for warping-based motion estimation 59

Bibliography **63**

Index **67**

Introduction

Over the course of the last decade, 3D hardware has become cheap and commonplace; an average new PC has better graphics capabilities than dedicated workstations had 10 years ago. As a result, applications that make extensive use of 3D graphics are becoming increasingly widespread and popular; 3D games in particular have been one of the main reasons for the rapid development of 3D graphics hardware during the last few years. It also means that standards have risen considerably: even relatively cheap hardware is able to render scenes with millions of visible triangles and hundreds of megabytes of texture data at interactive rates. At the same time, 3D display of one sort or another has appeared even in relatively weak embedded and mobile devices; an example is car navigation systems, which by now typically show a (relatively crude) 3D rendition of the area surrounding the car.

But while 3D rendering is starting to become a commodity, there is a huge variation in the available levels of performance and quality. On PCs, high-end graphics cards not only provide a far larger featureset than integrated graphics chips, they are also between 1 or 2 *orders of magnitude* faster—with a similar difference in price and power consumption, to be sure. For embedded and mobile devices, the differences are even bigger, ranging from graphics chips that only provide a framebuffer with no hardware-accelerated rendering at all through hardware support for 2D vector graphics to fully-fledged 3D chipsets roughly on par with high-end PC rendering hardware around 2001.

This creates a big problem for application developers: the only way to achieve consistent quality and performance over a wide range of different target machines is to either have separately tuned datasets and renderers for different configurations, which is very expensive to develop, or to aim for the lowest common denominator, which means that the added capabilities of newer hardware don't get used at all.

Another problem is the acquisition or creation of content: owners of high-end graphics cards expect applications to use them, but acquiring or creating geometry and materials at a high level of detail is a costly process. Still, the resulting data is needed for rendering, so it is made available to every user—and with systems such as *Google Earth*, the potential userbase is everyone with access to the Internet. This is a problem for providers of GIS (geographical information system) datasets: this data is quite costly to obtain, and making it available to virtually everyone free of charge is not in their best interest.

Finally, just as development of 3D hardware does not stand still, neither does rendering and geometry/material scanning technology, with the result that data formats go in and out of fashion every few years, always being replaced by a representation that is more suitable for the current state of the art. All attempts at interchange formats are at best

limited (again trying to match some least common denominator, or only focusing on a quite specific application) and at worst obsolete by the time their specification is finished. This is a big problem when developing an application that is supposed to have a lifetime of at least a few years, since it complicates the choice of delivery format considerably: because high-resolution 3D data is quite big, changing the data format along the way—possibly having to convert a huge server-side database, and then redistribute several hundred megabytes worth of data to each and every user—is prohibitively expensive, so backwards compatibility is necessary, with all the resulting complexities.

In short, writing 3D applications, a quite complex endeavour by itself, is made even more complicated because the environment is very "hostile" to developers, requiring them to design for a huge range of hardware capabilities at the same time (and—a problem often ignored in publications—work around hardware and driver bugs). Any means of simplifying this task is thus greatly appreciated. In a networked environment, an obvious approach is to try and offload some of the work to the server. Martin [Mar00] describes three primary ways of performing 3D rendering in a client-server environment:

- *Client-side methods*, where the server only supplies a description of the 3D scene (model/geometry data, textures etc.) to the clients, which have to perform the rendering themselves. This is the approach taken by the majority of current networked 3D applications; it is relatively simple on the server side (serving static data is comperatively easy by itself, and there is lots of practical experience), and a single server can easily process a large number of clients. However, as explained above, rendering on the client causes significant problems for implementors and content providers.

- *Server-side methods* let the server peform all the rendering. The finished image is then transmitted to the client, usually in compressed form. Due to network and rendering latency, such methods are usually not suitable for interactive applications; also, the maximum number of concurrent clients per server is usually several orders of magnitude lower than for client-side methods, because the server workload per client is much higher. On the plus side, because all rendering is performed on machines with known configuration, compatibility issues on the client mostly disappear, and the same quality is available to all clients no matter how powerful their hardware is. Also, it is easy to use different scene representations, or different renderers for that matter: all it takes is installing a new application on the server(s), which is completely transparent to all clients.

- *Hybrid-side methods* perform rendering on both the server and client sides, in various ways; one such way is to render "background" geometry on the server and "foreground" geometry on the client (which is which depends on the application at hand). Another way is to render a low-resolution version of the scene on the client and a high-resolution version on the server, only transmitting the difference between the two, effectively using the low-resolution image as prediction of the high-resolution one. The problem is that such schemes inherit weaknesses from

both client- and server-based methods: rendering overhead is incurred on both sides, but scene data still needs to be transmitted to clients, and all the rendering problems stay the same, because high-quality, high-detail rendering makes a visible difference *especially* for foreground objects and when using high resolutions. Thus, while a valid option, a hybrid-side renderer is not very interesting in practice, at least for the applications mentioned.

In this text, I will assume a client-server system with server-side rendering. While not a panacea, server-side rendering certainly has serious practical advantages (at least for applications that require cutting-edge rendering quality but only a modest number of clients) and has received a disproportionately small amount of attention so far. There are two main issues that need to be considered for a practical server-side rendering system:

1. Each server needs to render frames for several clients, an expensive operation. If possible, one would like to reduce the total workload by reducing the amount of per-client work, by exploiting temporal coherency between the frames produced for a single client and similarities between the viewpoints of different clients.

2. After rendering, the server also needs to compress the outgoing data for each of the clients. Again, this is quite expensive; however, since the images are rendered by the server, there is quite precise information about what changed from one frame to the next; current video coders do not make use of such information, which might incur a cost in both bitrate (and thus network bandwidth) and performance.

My main contributions are as follows: Chapter 2 will first look at depth image warping, an image-based rendering technique particularly useful when there are frame-to-frame coherencies, and investigate its usefulness when applied to video compression—the hope being that such a system would provide high quality while requiring less resources on the server side than "normal" video compression would, with reasonable cost on the client side.

Chapter 3 describes a method that is more useful to improve compression ratio and speed in practice: warping is used to estimate motion vectors for conventional motion compensation-based video codecs, the advantage of such codecs being that they are standardized already, and that decoder software and hardware is available virtually everywhere. Finally, it also briefly talks about ways of using warping and related methods to accelerate rendering.

1. Previous work

1.1. Image-based Rendering

One way to render complex 3D scenes with relatively low computational cost is using *image-based rendering* (IBR) techniques. Instead of transmitting an explicit representation of a 3D scene (by directly describing the geometry of objects, material properties, light sources etc.) and rendering 2D viewpoints on the client side, the scene is described *implicitly* by one or more images supplemented with additional information. This typically includes at least a description of the camera type, position, viewing direction, and field of view, but may also include harder to obtain information such as per-pixel depth values. The theoretical foundation for image-based rendering is the *plenoptic function* as defined in [AB91]:

$$P = P(\theta, \phi, \lambda, t, V_x, V_y, V_z)$$

measures the incoming radiance from the direction (θ, ϕ) as observed by an idealized point-shaped viewer at position (V_x, V_y, V_z), for every wavelength λ and at every time t. Different image-based rendering methods correspond to different ways of *sampling* the plenoptic function (typically involving quite severe dimensionality reduction) and later *reconstructing* an image—itself a sampled 2D representation of the plenoptic function for fixed viewer position, fixed time and a small set of representative wavelengths—from the sampled values.

There's a big variety of image-based rendering methods available; a relatively recent survey is [ZC04]. Probably the most suitable method for the target application is warping of planar depth images. Planar depth images are very easy to generate with both rasterization- and ray tracing-based renderers at neglegible to no extra cost compared to a regular image, there are no arbitrary restrictions to viewer movement, and planar warping is able to synthesize views quickly from an input dataset that is not much bigger than the destination image—in contrast to IBR methods that use a high-dimensional scene representation. This makes it very useful in a client-server setting, where network bandwidth is a limited resource and rendering should be fast and benefit from dedicated harware support if available. All in all, it is a very natural fit, so I will focus on warping and related methods for most of this chapter.

1.2. Warping of planar depth images

Needless to say, a prerequisite for warping-based video compression is that (efficient) warping algorithms exist. McMillan's dissertation [McM97] lays the necessary ground-

work for various camera models (including, but not limited to, normal planar pinhole cameras). He describes the original (forward) warping algorithm, shows how occlusions can be resolved by simply processing pixels in the right order, introduces different reconstruction methods to turn a set of warped points back into an image, and explains the problems that can occur during warping. He also derives the *inverse warping* algorithm. His main results are summarized at the beginning of chapter 2.

The basic warping algorithm has several limitations (cf. section 2.1.2) that manifest themselves as different kinds of errors in the warped images. While most of these errors are direct results of incorrect assumptions being made either during the warping or reconstruction process, *exposure errors* are not; they occur because a single depth image can't describe general 3D scenes, as everything in the shadow of the foreground objects is necessarily invisible. Shade et al. [SGHS98] introduce *layered depth images* (LDIs) to solve this problem. As the name suggests, they allow several (depth) layers to be stored per pixel, so that a layered depth image can also contain information from the area behind foreground objects. LDIs can be displayed using a variant of McMillans warping algorithm, without a z-buffer. However, they are relatively hard to generate; even when it is possible to efficiently obtain several depth layers per pixel (this is a lot easier with a ray tracer than it is with a z-buffer based rasterizer), doing so increases rendering cost notably. Otherwise, a single LDI is generated by combining several "normal" depth images, also an expensive process. This limits the usefulness of layered depth images for dynamic rendering, although they are still useful for storage, since storing a single LDI is smaller than storing several very similar depth images.

Marks PhD thesis [Mar99] on "Post-Rendering 3D Image Warping" describes techniques more suitable in a real-time environment. The general idea is to use warping to speed up a conventional (rasterization or ray tracing-based) renderer by only generating a few images per second and using warping to interpolate between them. Such a system obviously needs to be faster than the original renderer to be worthwhile, so there is a focus on fast reconstruction and hole-filling algorithms, including some that have comperatively simple hardware implementations. Another chapter deals with general implementation issues for warping hardware, including a derivation of precision requirements for a warping implementation using fixed-point arithmetic and a detailed analysis of the memory access patterns of warping. This analysis also results in a more cache-friendly reference image traversal strategy.

1.3. Warping-based client-server rendering

I am not the first to suggest using warping-based methods in a client-server rendering environment. Warping a reference depth image to a new viewpoint typically produces a quite dense image, apart from holes that appear due to occlusion or exposure errors; this behavior can be exploited on the server side by only rendering those areas where no information is available, and for compression by only transmitting new data pixels where such errors occur (this is known to both the server and client sides, so their

locations don't need to be transmitted). In [YN00], Yoon and Neumann describe their IBRAC system (Image-Based Rendering Acceleration and Compression) which is based on this idea. The first frame is transferred losslessly and including depth information and camera parameters. Subsequent frames first transmit the new camera position; this is used to synthesize the new image from a reference frame, usually the previous one, with inverse warping (cf. Section 2.1.4). The warping process is done both at the server and the client sides. While warping, destination pixels are classified into one of three different categories:

1. A *definite hit* occurs when the current ray transitions from in-front-of to behind a surface.

2. A *definite miss* occurs when it "hits" the background (i.e. it doesn't leave the view frustum but failed to hit any surfaces).

3. Finally, a *possible hit* occurs when the ray leaves the view frustum without hitting any surfaces, when it passes behind a surface without ever having been in front of it, or when it hits a pixel close to a depth discontinuity (heuristically detected by testing whether adjacent pixel's depth values differ by more than a given threshold).

1 and 2 are both deemed high-confidence estimations and predict the color at the pixel "hit" or the background color, respectively. Only pixels that fall into category 3 get re-rendered at the server side, then transmitted to the client. Since both server and client perform the same inverse warping process, positions of possible hits do not have to be transmitted. Finally, the stream of color and depth values is compressed using `gzip`.

The authors compare their algorithm against MPEG-2 and claim "a per frame compression ratio of 2 to more than 10 times better than MPEG2 encoding" with their test sequences (flights around objects of various complexity in front of a black background), at quite good objective quality (consistently above 35 dB PSNR). However, the algorithm is purely warping-based and has no means of replacing colors or depth values for pixels that didn't fall into the "possible hit" category; for surfaces that are not perfectly diffuse, this obviously poses a problem. Even worse are invisible occluder errors, since there is by definition no information contained in the reference image that suggests their presence. In short, when such situations occur in a scene, the generated view can end up very different from an actual rendered view, without ever been corrected or even noticed by the system. That means that even though the reported results may be good, the algorithm achieves them at the cost of oversimplification; the invisible occluder problem especially is very serious and likely to be a "show-stopper" in complex scenes.

Less ambitious and also less problematic is a client-server system briefly described in chapter 6 ("Real-time remote display") of Marks aforementioned PhD thesis [Mar99]. The client transfers the current camera position to the server, which periodically renders a new view and transmits it back to the client. Warping is used for two purposes: to

compensate for network latency, and to have the client render at a higher frame rate than server-side rendering speed and network bandwidth allows. For each reference frame, the server renders 4 views corresponding to the sides of a cube centered at the viewer without the top and bottom faces, for a total 360° horizontal and 90° vertical field of view (largely avoiding invisible occluder errors). Other than that, the system is again purely warping-based and thus subject to occlusion and exposure errors. No compression is used for the communication between client and server. Hudson and Mark [HM99] update this system to use 3 reference image sets with different centers of projection. Together with an algorithm to select "good" reference viewpoints, this notably reduces occlusion and exposure errors in typical walkthrough scenarios. The lack of compression makes the system impractical for use over the Internet, though. Furthermore, it has quite high CPU requirements: the client uses three MIPS R4400 processors clocked at 200MHz to produce images with a resolution of 320×240 pixels at 7 frames/second!

Chang and Ger [CG02] describe a similar system with significantly lower CPU usage, using PDAs as clients. They only use single planar reference images (which makes the system susceptible to invisible occluder errors) but do support layered depth images [SGHS98] to reduce occlusion errors. The client-side warper as described in the paper just plots warped pixels at their destination position without a reconstruction or hole-filling pass; this seems to be mainly due to limited computational resources on the target platform. The frame rate is reported at about 6 frames/second on a test system using a 206MHz StrongARM processor, while rendering at a resolution of 240×180 pixels—still relatively slow, but proving that simple warping variants can render at interactive rates even on very low-powered platforms.

Another client-server system using a later genereation of PDAs as clients is described by Thomas, Point and Bouatouch [TPB05], using a somewhat different approach. Aiming at urban walkthroughs, they try to optimize placement of reference cameras so that a complete scene can be described exclusively using a relatively small number of reference images—in contrast to the previously described schemes, which always place cameras along the path the user takes, irrespective of the underlying scene geometry. The main contribution of the paper is a camera placement algorithm for urban scenes; it works on a simplified 2D description of the street network and is not applicable to general 3D scenes. The server doesn't need to constantly render new camera views; rather, new views are only rendered when the user moves to enter an area not well covered by the currently active reference views. This reduces network bandwidth requirements significantly; reference views are additionally compressed using `zlib`. To make use of the multiple reference images, the clients need to perform several warping operations per frame (typically 2 to 4).

On the target PDA, using an Intel PXA263 processor at 400 MHz and a target resolution of 320×240 pixels, frame rates are reported as below 4 frames per second when using 4 reference images—without hole filling and using a fixed point implementation of the warping algorithm that, judging by the images in the paper, has very poor rendering quality. They also report frame rates on PCs (the paper doesn't state how fast

the machines are), presumably using the floating-point variant of the warping algorithm and with hole filling, though it isn't explicitly stated; the frame rate there is around 20 fps when only one reference image is used and drops to about 8 fps with four reference images. These results are quite disappointing; while the camera selection algorithm makes it unlikely that any significant errors are introduced during the warping process, it does so only after severely constraining the scene geometry and viewer; with all the restrictions on scene geometry, a simple 2D raycaster would probably suffice to render the scenes, and do so smoothly on a PDA. In any case, the algorithm is very specialized and of little to no use for more general scenes.

1.4. Warping and compression

Apart from IBRAC, the systems mentioned in the previous section are all designed to work over a local network with relatively high transfer rates and low latencies; as a result, they do not invest much effort on compression, since it would cost a lot of CPU time and have little to no practical benefits. For usage over the Internet, available bandwidth from the server to the client is more constrained (and a cost factor!), which makes compression a lot more important.

Aliaga et al. [ARPC06] present a method designed for architectural walkthroughs of real-world buildings, where floor plans (and thus a coarse description of the underlying geometry) are available and photographs are fairly easy to obtain; warping is then used mainly as prediction between similar images, to reduce the amount of data that has to be stored. This differs from the previously mentioned methods in significant ways: first, the system is not interactive; all images are known beforehand. In fact, photos are the primary scene description. To capture a scene accurately, a lot of photos must be taken; the used test datasets use several thousand images, with the centers of projection between neighbored reference cameras being 2.2 inches (or less) from each other. Second, because the images are not rendered from a (more or less) exact scene description, but real photos, several assumptions inherent in warping are not met: materials are not perfectly diffuse, the camera is not an ideal pinhole camera, and there is noise present in both image and registration data. To deal with these imperfections, the algorithm doesn't assume that warping produces a perfect image, not even where sufficient data is available; instead, the difference between warped and actual image is coded. As a side effect, this also allows invisible occluder errors to be handled properly. Third and last, accurate depth data is not available; instead, the floor plan is used as proxy geometry to provide approximate per-pixel depth information.

All images are known beforehand, and it is to be expected that the order in which images are coded has an influence on the compression ratio. The authors propose a hierarchical structure, where images are organized in a binary tree, with each node being predicted from its parent. They also present a relatively simple tree construction algorithm that approximately minimizes overall coding cost for the complete tree. Nodes on deep levels in the tree require a fairly long sequence of warps to obtain the actual

image data; to reduce this access time, the authors propose storing *I-nodes* (nodes that are not predicted from their parent node, conceptually similar to MPEG I-frames) every few levels, reducing compression ratio slightly but providing much more uniform access times. At a compression ratio of roughly 85 : 1, the visual quality of decoded images is reported as significantly better than an MPEG-2 encoded video of a linearization of the image database (it is not explicitly stated how images are ordered for MPEG-2 encoding). No objective measurements are performed, however, so it is difficult to quantify the quality improvement obtained by using warping. It is not mentioned whether the proxy geometry is stored along with the compressed data; for the scenes in question, where the proxy geometry is very simple, the impact on compression ratio is probably negligible, but for scenes with complex geometry this could be a notable cost factor.

The paper also doesn't use a regular pinhole camera model; instead, the *Depth Discontinuity Occlusion Camera* model as introduced in [PA06] is used. In short, rays that pass close to object silhouettes are "bent around the object" to include information from behind. This reduces exposure errors at the expense of somewhat lower resolution near silhouettes (to "make space" for the behind-the-object samples). Also, since this distortion is geometry-dependent, either the geometry or the resulting distortion map needs to be stored along with the image (which is not an issue for this particular application, since it already uses proxy geometry to estimate per-pixel depth for warping). The impact of this camera model is measured in a separate section, where it is reported that at "low to medium compression ratios (35 : 1 to 83 : 1)", the occlusion camera yields difference images that are "up to 3.3% smaller and 4.4% more compact" (the former apparently meaning difference image energy and the latter compression ratio). However, "the overall average improvement we saw in images containing disocclusions was only about one percent" (all quotes from [ARPC06], p. 22). Considering that the new camera model needs information about the geometry and results in a more complex warping algorithm, these results are rather disappointing.

Overall, the paper contributes several promising ideas and shows good results; however, the algorithm as described is not applicable to interactive applications, where the images to be compressed are not known beforehand.

A coding algorithm *for* depth images—as opposed to coding normal images using warping—is presented by Duan and Li [DL03], who discuss compression of layered depth images; since a LDI is a generalized depth image, the same algorithm can be used for normal (single-layered) depth images with trivial changes. First, an image is generated that contains the number of layers for each pixel; this is coded losslessly using JPEG-LS [WSS00]. The per-layer color data is then converted to the YCbCr color space. While the first layer is normally stored completely, later layers are increasingly sparse. To aid decorrelation, horizontal "holes" (spans of pixels not coded in that layer) are removed before encoding, effectively grouping all stored pixels at the left side of the image. This is then coded (componentwise) using the video object wavelet codec [XLLZ01]. The paper also describes a simple bitrate allocation method to minimize visible distortion; care must be taken especially with the depth (disparity) components, since coding artifacts there can cause objects to tear apart at the boundaries. It is reported that compression of

LDIs with a ratio of up to 17 : 1 is possible with "minimal visual distortion", although the objective (PSNR) quality measurements are relatively bad. However, quality is measured by distortion in rendered images, not the actual coded LDI; a bad choice of reconstruction filters in the renderer (or optimizations that sacrifice visual quality for speed) can amplify small errors significantly, without necessarily being very visible. For example, if the disparity components have a small bias, large objects may end up at a slightly different position. While mostly invisible to a human observer, such errors can cause significant distortion in the PSNR sense.

1.5. Video coding

As an alternative to using warping for compression, existing video codecs can be used as well, the main advantage being that there is already widespread hardware and software support for encoding and decoding with these methods. Since section 2.3 compares a warping-based prediction model with the prediction used in the Dirac codec and chapter 3 uses warping to accelerate H.264 encoding, this section will be somewhat more detailed than the preceding ones, to give a detailed overview about how these codecs work, and also to establish terminology.

Most state of the art video codecs are quite similar in basic structure; in fact, the general design hasn't changed much since the introduction of the original MPEG video standard [ISO93]. I will review three different algorithms: MPEG-4 part 2 [ISO01], H.264/MPEG-4 AVC [ITU05], and Dirac [BBC08]. The first two are well-established international standards and already being used for HDTV, media formats such as HD-DVD and Blu-Ray, and also smaller embedded devices (e.g. MP3 players that support video playback). Dirac is a somewhat newer and more experimental design; nonetheless, the intra-coding part is already standardized as SMPTE VC-2, and the BBC intends to start using Dirac for streaming video within the next few years.

All these video codecs support two basic modes: intra and inter coding. Intra coded frames are self-contained (like an image file), while inter frames use data from other (previously coded) frames to exploit temporal coherency between frames. The intra part of any video codec is just a still image coder; inter coding and the additional redundancies it can exploit are the reason why current video formats are able to achieve significantly lower bitrate at the same quality, compared to coding each frame individually—at the cost of complicating seeking.

In fact, there is a three-class distinction between frame types: *I-frames* (called L0 frames in Dirac, and often referred to as key frames) are intra-coded frames, and serve as synchronization points for seeking. *P-frames* (L1 frames in Dirac) are inter-coded and predicted from previous I or P frames. Finally, *B-frames* (L2 frames in Dirac) are *bidrectional* inter frames, meaning they can refer to the I and P frames both immediately before and immediately after them; they are not themselves used as source frames for prediction—which can be beneficial, as an encoder can heavily compress a B-frame to save bits without penalizing the following frames.

B-frames may refer to frames "in the future", but the encoder still has to know the referred frames to be able to decode a B-frame properly; hence, when B frames are used, frames are reordered during encoding, so that the sequence of frames received by the decoder is "causal". Both encoders and decoders need to buffer frames to be able to perform this reordering, which consumes additional memory and introduces extra delay when videos are being coded in real time. The sequence of frames from one I-frame up to (but not including) the next is referred to as a group of pictures, or GOP.

All three codecs perform the inter prediction step using different variants of *motion compensation*: the destination image is partitioned into rectangular blocks of pixels. For each block, a single 2D vector (m_x, m_y) is stored, which is an offset relative to the block's position that specifies where the source data for the respective block is located in the reference frame (that is, it points from the destination position to the source position, opposite the motion direction). Conventional (block) motion compensation just copies the respective pixels over, and is used by both MPEG-4 variants; Dirac uses overlapped block motion compensation, which uses larger destination blocks that overlap (as the name suggests). There, each pixel is predicted as a weighted sum of the predictions from the different destination blocks that overlap that pixel.

The problem of finding these motion vectors on the encoder side is called *motion estimation*. Formally, it is an instance of the optical flow problem in computer vision, but there are no regularity constraints on the motion vectors, and the objective function is coding cost (i.e. size of the coded motion vector plus resulting residual data in bits) instead of a direct image similarity metric. As a result, codec implementations typically don't use direct optical flow algorithms such as the Lucas-Kanade algorithm [LK81]; specialized methods, usually based on explicit search with early termination heuristics, are more common. This motion estimation process is quite costly (even with optimized algorithms), and typically a significant fraction of the total video encoding time is spent on it.

Motion estimation (and motion compensation) is performed on blocks of pixels. In MPEG/H.264 parlance, these blocks are called *macroblocks* and always have a size of 16×16 pixels, while Dirac calls them *superblocks* and supports variable sizes. The original MPEG standard only had one motion vector per macroblock; all three newer algorithms allow subdividing a macroblock into smaller *partitions* with different motion vectors, which is useful for macroblocks that contain object boundaries. Motion vectors are typically specified up to fractional (half- or quarter-pixel) precision; the reconstruction filters used are part of the codec specification.

The image obtained from the motion compensation process is an approximation of the target frame; this approximation is subtracted from the actual image data, yielding the *residual image*. The way this image is coded is where the algorithms differ the most, but the ideas are closely related. Dirac uses an integer wavelet transform to decorrelate the luminance and chrominance difference images, H.264 uses a custom-designed integer transform on 4×4 pixel blocks,[1] and MPEG-4 uses the 2D DCT-II with a block size of

[1] Streams using the newer "High" profile can also use a 8×8 pixel transform in smooth areas.

8×8 pixels; all three methods use the same transform for luminance and chrominance components.[2] The MPEG standards up to MPEG-4 do not specify the exact inverse DCT to be used, but instead require that compliant decoders do not exceed certain error bounds; this was meant to increase implementation flexibility, but causes encoders and decoders to go out of sync over time when they use different IDCT implementations ("IDCT mismatch")—a problem that causes quite visible artifacts. This is usually worked around by inserting I-frames at regular intervals (once or twice per second), but the newer codecs (Dirac and H.264) specify inverse transforms exactly to avoid this problem.

Computing these various transforms yields *transform coefficients* which are then quantized, the main lossy step in video coding. Since they are the result of a decorrelation process, those coefficients can be quantized individually; this is done using uniform quantizers with or without dead zone. The quantization step size used can be adapted over the course of the frame to meet a bitrate target, and often also depends on the particular coefficient being coded (for example, low-frequency coefficients typically get assigned lower step sizes than high-frequency ones do). The resulting quantized coefficients are then combined with the motion vector data and entropy coded. MPEG-4 uses a run-length coding scheme and predetermined Huffman tables, while Dirac employs an adaptive arithmetic coder. H.264 supports specialized variants of both, dubbing the CAVLC (context-adaptive variable length coding) and CABAC (context-adaptive binary arithmetic coding), respectively.

Practical applications also usually have limits on how many bits can be transferred or decoded per second; encoders have to monitor how many bits are spent on individual frames, and correct quantization settings to meet the target bitrate if necessary. This process is called *rate control*. To this end, frames are typically analyzed by encoders before any actual data is encoded, to determine which areas are visually important and to get a rough estimate of how many bits are going to be spent on motion vectors; this is then used to choose quantization step sizes for the frame. For films and other "static material", two-pass encoding is used: Each video is encoded twice. During the first pass, statistics are gathered, while the final output is determined in the second pass. This allows encoders to look ahead in time, coding "easy" frames with a lower number of bits to save them for "difficult" frames. The resulting improved bitrate allocation typically increases perceived quality substantially, but is not possible for live broadcast and streaming applications for obvious reasons.

[2] This is the reason for the 16×16 pixel macroblock size, since chrominance information is downsampled by a factor of 2 in both dimensions.

2. Warping of Depth Images for Video Compression

Warping of Depth Images was first described by McMillan in his dissertation [McM97]. It is based on the observation that an image with per-pixel depth information together with a description of the camera and viewpoint can be viewed as a scene representation; one can then later change the camera position or orientation and use the original image to synthesize a new image corresponding to the changed viewpoint. This leads to a very simple image-based rendering algorithm that doesn't restrict the movement of the viewer in any way; as it is able to produce approximations of new images (with a changed viewpoint) given previous images, it is also useful for compression. Section 2.1 explains the basic warping algorithm and some important variations.

Most of the existing work on using warping for compression (as reviewed in chapter 1) assumes that warping does, in principle, produce a "perfect" rendering of the image—except in areas where no suitable data is present in the source image or images, of course. Section 2.1.2 explains why this is not, in general, the case. However, even when the assumptions inherent in the warping process do not hold (for example, when surfaces are not ideal lambertian reflectors), warping is likely to produce a good *approximation* of the correct image as long as no drastic camera movements are involved. This suggests a less radical approach to warping-based compression: instead of only filling holes that occur due to exposure errors, the difference between the image produced by warping and the actual rendered scene is coded—in short, warping is employed as a *predictor*.

This results in a codec structure very similar to that used by modern video codecs (their basic components are reviewed in section 1.5). In short, while most current video codecs employ motion compensation to obtain an approximation of the current frame to be coded from previously codec reference frames, warping can adequately perform the same task (for static 3D scenes with known depth data, at least). However, warping as it is typically used requires a relatively expensive reconstruction step, which is undesirable for a method intended to be able to play back video at rates of ≥ 20 frames/second, preferably even on embedded devices; section 2.2 develops a somewhat simpler method that trades simplicity and speed for a slight decrease in visual quality and coding efficiency.

Finally, section 2.3 concludes this chapter by comparing various warping variants with motion compensation, judging them by the quality of the predicted images they produce and the amount of data that needs to be stored to produce these predictions: motion vectors for motion compensation and per-pixel disparity information for warping.

2.1. Warping of depth images

2.1.1. The warping equation

This section summarizes chapter 3 of [McM97], mainly to introduce the setting and establish terminology. Throughout this chapter, the camera will be described by an idealized planar-pinhole model: The camera occupies a single point in space, the *center-of-projection* (COP), and measures incoming radiance along rays from the COP to a bounded planar region some distance away from the camera, the *image plane*.

Assuming a fixed 3-dimensional *world coordinate system* is given, one can map a position (u, v) on the image plane (given in the *image coordinate system*) to the direction of a corresponding ray from the COP that passes through it with a linear system:

$$\mathbf{d} = u\,\mathbf{a} + v\,\mathbf{b} + \mathbf{c} = \begin{bmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} =: \mathbf{P}\mathbf{x} \tag{2.1}$$

In this expression, $\mathbf{a}$ and $\mathbf{b}$ are (world space) basis vectors of the image coordinate system, $\mathbf{c}$ points from the COP to the origin of the image coordinate system (the top-left corner of the image plane, with $\mathbf{a}$ pointing to the right and $\mathbf{b}$ towards the bottom) and $\mathbf{P}$ is the *projection matrix*. We also normalize $\mathbf{a}$ and $\mathbf{b}$ to represent the "width" and "height" of one pixel in the sampled image, respectively. Finally, $\dot{\mathbf{C}}$ denotes the position of the camera's center-of-projection. The camera can then be completely described using $\mathbf{P}$ and $\dot{\mathbf{C}}$; this is visualized in figure 2.1.

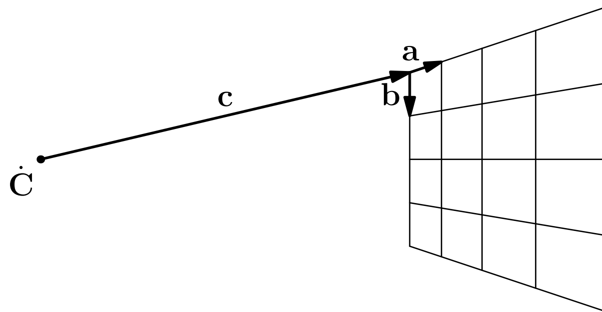


Figure 2.1.: Camera coordinate system.

Now assume there are two different cameras given by $\mathbf{P_1}, \dot{\mathbf{C}}_\mathbf{1}$ and $\mathbf{P_2}, \dot{\mathbf{C}}_\mathbf{2}$ that both see the point $\dot{\mathbf{X}}$. It follows that there are positions $\mathbf{x_1}, \mathbf{x_2}$ on the image planes of the cameras[1] such that the corresponding ray passes through $\dot{\mathbf{X}}$, as shown in figure 2.2. By

[1] The image plane is 2-dimensional, so $\mathbf{x_1}$ and $\mathbf{x_2}$ should be too; however, in the following, I assume that the two are in fact 3-vectors with the third component set to one. This is the usual homogenous notation for points in the projective plane.

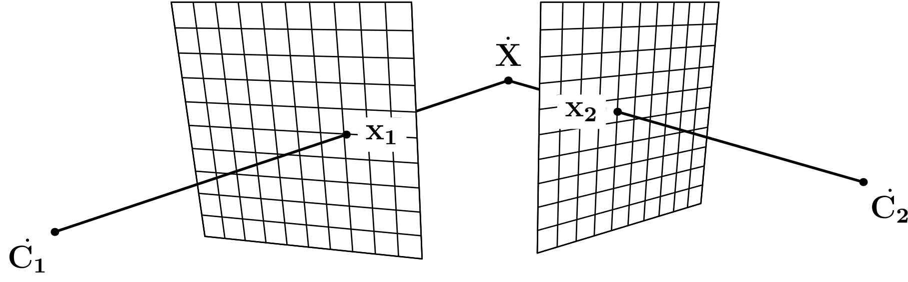


Figure 2.2.: Two cameras that see the point $\dot{\mathbf{X}}$.

construction, the two rays intersect, so there are scalars t_1, t_2 for which

$$\dot{\mathbf{X}} = \dot{\mathbf{C}}_\mathbf{1} + t_1\mathbf{P_1x_1} = \dot{\mathbf{C}}_\mathbf{2} + t_2\mathbf{P_2x_2}.$$

Regrouping terms yields

$$\frac{t_2}{t_1}\mathbf{P_2x_2} = \mathbf{P_1x_1} + \frac{1}{t_1}(\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})$$

which can in turn be simplified by defining $\delta(\mathbf{x_1}) := \frac{1}{t_1}$ (this value is called *generalized disparity* because it is proportional to *stereo disparity*, the quantity that is measured to estimate depth from aligned stereo image pairs) and using the symbol $\doteq$ to denote that the left- and right-hand sides are identical up to a scalar factor:

$$\mathbf{P_2x_2} \doteq \mathbf{P_1x_1} + \delta(\mathbf{x_1})(\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})$$

Finally, multiplying with $\mathbf{P_2^{-1}}$ from the left yields the *planar image-warping equation* (or just *warping equation* for short, since I only consider planar pinhole cameras here).

$$\mathbf{x_2} \doteq \mathbf{P_2^{-1}}(\mathbf{P_1x_1} + \delta(\mathbf{x_1})(\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})) \tag{2.2}$$

This equation describes where a point $\mathbf{x_1}$ in a reference image (as seen by camera 1) ends up on the image plane of camera 2, only depending on the parameters of both cameras and the generalized disparity at the source point $\mathbf{x_1}$. Since no information dependent on the destination point is involved, equation 2.2 can be used with *any* destination camera (provided that it still is a planar pinhole camera, of course). In practice, it is useful to rewrite equation 2.2 as a matrix equation depending only on u_1, v_1 and $\delta(u_1, v_1)$:

$$\begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} \doteq \underbrace{\begin{bmatrix} \mathbf{a_1}\cdot(\mathbf{b_2}\times\mathbf{c_2}) & \mathbf{b_1}\cdot(\mathbf{b_2}\times\mathbf{c_2}) & \mathbf{c_1}\cdot(\mathbf{b_2}\times\mathbf{c_2}) & (\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})\cdot(\mathbf{b_2}\times\mathbf{c_2}) \\ \mathbf{a_1}\cdot(\mathbf{c_2}\times\mathbf{a_2}) & \mathbf{b_1}\cdot(\mathbf{c_2}\times\mathbf{a_2}) & \mathbf{c_1}\cdot(\mathbf{c_2}\times\mathbf{a_2}) & (\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})\cdot(\mathbf{c_2}\times\mathbf{a_2}) \\ \mathbf{a_1}\cdot(\mathbf{a_2}\times\mathbf{b_2}) & \mathbf{b_1}\cdot(\mathbf{a_2}\times\mathbf{b_2}) & \mathbf{c_1}\cdot(\mathbf{a_2}\times\mathbf{b_2}) & (\dot{\mathbf{C}}_\mathbf{1} - \dot{\mathbf{C}}_\mathbf{2})\cdot(\mathbf{a_2}\times\mathbf{b_2}) \end{bmatrix}}_{=:\mathbf{W}} \begin{bmatrix} u_1 \\ v_1 \\ 1 \\ \delta(u_1, v_1) \end{bmatrix}$$

For the short derivation, refer to chapter 3 of [McM97]. $\mathbf{W} = (w_{ij})$ is the *warping matrix*. Solving for u_2 and v_2 then yields the following rational expressions:

$$\begin{bmatrix} r(u_1, v_1) \\ s(u_1, v_1) \\ t(u_1, v_1) \end{bmatrix} := \mathbf{W} \begin{bmatrix} u_1 \\ v_1 \\ 1 \\ \delta(u_1, v_1) \end{bmatrix}$$
$$u_2 = \frac{r(u_1, v_1)}{t(u_1, v_1)}$$
$$v_2 = \frac{s(u_1, v_1)}{t(u_1, v_1)} \tag{2.3}$$

This version of the warping equation can be implemented in a straightforward fashion and leads to a reasonably efficient image warper with very little code. Care must be taken to ensure that visibility order is preserved, that is, "front" pixels get drawn in front of "back" pixels —[McM97] shows that this can be achieved by processing source pixels in the correct order. Also, integer pixel coordinates in the source image may (and usually will) be mapped to noninteger coordinates in the destination image, and the warping process distorts the relative area of source pixels; this necessitates a *reconstruction step* after the warping has been performed. I will return to this subject in section 2.1.3.

2.1.2. Limitations of warping

It is important to mention that, when used for image synthesis, planar warping has a few notable shortcomings:

- The derivation for warping is purely geometrical and assumes that a point "looks the same" from all directions. This is wrong in general; it only holds for perfectly diffuse (lambertian) materials.

- Similarly, it is assumed that surfaces in the source image are "solid": translucent and refractive surfaces are not considered, and there is no proper way to assign either a single color or disparity to a point that lies on such a surface.

- Warping of depth images is subject to *occlusion errors* and *exposure errors*. Exposure errors occur whenever a destination pixel lies in the "shadow" of an object in the source image — there is no information in the source image to determine the color of the destination pixel. This can be solved by using *layered depth images* [SGHS98] which are able to store several colors and disparity values per source image pixel, but they are relatively complex to generate (mainly because neither photos nor rendered images contain the required information) and form a rather unwieldy data structure, at least compared to the simple 2-dimensional array representation of regular depth images. Occlusion errors, on the other hand, occur

when an object in the synthesized image incorrectly occludes another one; this can happen with some types of reconstruction kernels.

- Finally, *invisible occluder errors* can occur when an occluder in the original 3D scene would be seen by the destination camera, but isn't inside the frustum of the source camera and hence won't be reproduced by warping. While the previous problems are common to all warping-based schemes, this particular issue is caused by the limited field of view of planar pinhole cameras and disappears when panoramic views are used.

2.1.3. Reconstruction

While the mapping from source to destination image coordinates as expressed by the warping equation is quite straightforward, turning the resulting set of (x, y, color) tuples into an image is anything but. Warping can distort relative area significantly: points that get closer to the viewer will occupy more area in the destination image (and hence be spaced further apart), while points moving away from the viewer tend to form clusters. Both effects can appear in different regions of the same destination image, and even worse, depth discontinuities will cause gaps in the destination image that won't disappear (or even get smaller) with increased sampling resolution.

One could try to use general algorithms for *scattered data interpolation* to turn the set of points back into a regularly sampled image; however, such algorithms are quite slow and not necessarily well-suited to the problem at hand (for example, some of the mapped points might end up between other points defining a surface closer to the viewer; one would like such "hidden" points to be ignored). There are specific reconstruction algorithms for warping that are both more efficient and of higher visual quality than this generic approach.

McMillan [McM97] presents two reconstruction methods, which I will call the "gaussian cloud" and "bilinear patch" methods, respectively. The gaussian cloud method views each pixel in the source image as a small spherical splat. It then uses the Jacobian of the warping equation at the source position to determine how the original spherical shape should be distorted; the resulting deformed shape is then splatted onto the frame buffer. This accounts for the area distortion induced by the warping equation, but holes remain in areas that weren't visible in the source image. The bilinear patch method explicitly builds a mesh of quadriliterals from the source points; the points are then transformed to their respective positions in the destination image, and the resulting distorted quadriliterals are rasterized, interpolating colors bilinearly. This process also implicitly closes holes by interpolating over the gaps that appear due to depth discontinuities; while incorrect, this is visually far less distracting than simply leaving holes with the background color.

Mark [Mar99] presents two improvements of the bilinear patch method; one simply triangulates the quadriliteral mesh as formed by the "normal" bilinear patch and interpolates colors linearly over the resulting triangles, so that existing triangle rasterization

hardware can be used. It also explicitly detects depth continuities and uses a specialized hole-filling algorithm that tries to improve on the simple interpolation method. The second method is aimed towards hardware implementations that use supersampling and rasterizes axis-aligned quadriliterals with subpixel resolution.

All these algorithms, however, are quite expensive; they require drawing hundreds of thousands of triangles or splats even at quite modest resolutions. This is impractical for real-time processing on embedded devices unless specialized hardware is present.

An alternative is simply not to perform any explicit reconstruction. The frame buffer is cleared to a background color, and warped pixels are plotted into the frame buffer as they are produced. This is quite fast, but gaps due to depth discontinuities will be left open (like with gaussian cloud reconstruction), and due to the limited sampling resolution holes may appear even in regions without any depth discontinuities where the area distortion of the warping equation causes pixels to be pushed too far apart. A simple workaround is to use small squares instead of single pixels, with the diameter determined by the Jacobian of the warping equation; this is significantly faster than full gaussian cloud reconstruction and fills most non-depth-discontinuity gaps.

2.1.4. Inverse warping

To avoid the quite laborious reconstruction process, it seems tempting to try to reverse the mapping direction: instead of projecting source pixels into the destination image, start from the destination pixel and search a source coordinate that maps to that point. Since the source image is regularly sampled, obtaining a color for a given source position is simple, using for example nearest-neighbor or bilinear filtering. The main problem is that the destination position depends on the depth value at the source position, and the mapping isn't generally bijective, as evidenced by occlusion and exposure errors: a single destination pixel might be "hit" by more than one source pixel, or not hit at all. For each destination pixel, inverse warping needs to search for a matching depth value along a line in the source image, quite a time-consuming process. It is possible to speed up this search in the average case by using a hierarchical representation of the source depth buffer [Mar98], but in the worst case, it may still be necessary to traverse a full line in the source image. For source and destination images with $n \times n$ pixels, this amounts to $O(n^3)$ steps of work (i.e. $O(n)$ steps per destination pixel). Forward warping only needs $O(n^2)$ steps (which is $O(1)$ per pixel)—way more attractive for real-time processing, especially when the source image changes frequently, because each such change would require the hierarchical disparity structure used for inverse warping to be recomputed.

2.2. Fast warping for prediction in video coding

The most important consequence of using warping as a predictor is that errors made during the prediction process can later be corrected; hence even systematic errors can be tolerated when doing so reduces encoder/decoder complexity, or increases performance

or coding efficiency. The original MPEG codec uses a very simple model of 16×16 macroblocks of pixels being translated, which is only a very coarse approximation of motions happening in "real" videos, but quite compact to transmit (one 2D translation vector every 256 pixels) and easy to decode. Warping can similarly make some assumptions (like treating objects as perfectly diffuse) and take "shortcuts" in some places, at the sole cost of a certain loss in coding efficiency; the benefit of lower decoder complexity outweighs this cost if embedded devices are the target.

As explained in detail in section 2.1.3, the reconstruction step involved in warping is typically the most involved and most difficult one. "Normal" warpers need to do well here, since the reconstructed image is their final output and directly visible to the user. For the sake of prediction, however, a very simple reconstruction method is sufficient, as long as errors are either uncommon or typically inexpensive to code.

For videos with a frame rate of ≥ 25 frames/second and continuous camera motion, even the trivial "plot single pixels" reconstruction filter without any hole-filling produces quite reasonable results, as shown in figure 2.3. The images were obtained by recording a depth image sequence for a flight over a 3D city dataset, and generating each new frame by warping the frame before it. As is clearly visible, mostly individual rows and columns of pixels are missing; in the right image, some typical disocclusions are also visible (the "shadows" behind the buildings). But even these holes are, for the most part, relatively small. This suggests a very simple method: instead of using a more complicated reconstruction filter, simply keep plotting individual pixels and later fill both types of holes (those due to a limited number of warped pixels and those due to disocclusions) with one simple and fast screen-space algorithm. It is to be expected that errors are larger than they would be with e.g. gaussian cloud or bilinear patch reconstruction; the exact cost will be determined later, in section 2.3.1.

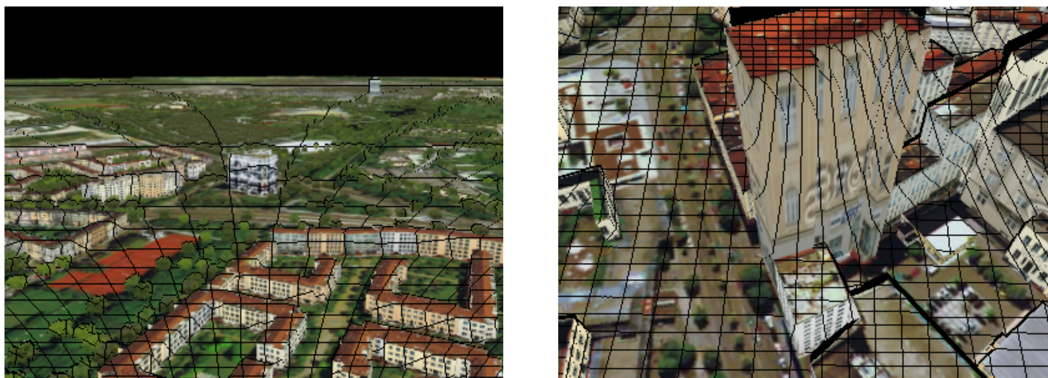


Figure 2.3.: Holes that appear when omitting the reconstruction step altogether.

The proposed algorithm is straightforward: First, a normal warping operation is performed; source pixel coordinates are warped into the destination image, and the resulting coordinates are rounded to integers. If these coordinates fall within the destination image, color and disparity of the pixel in the destination image are written at that location.

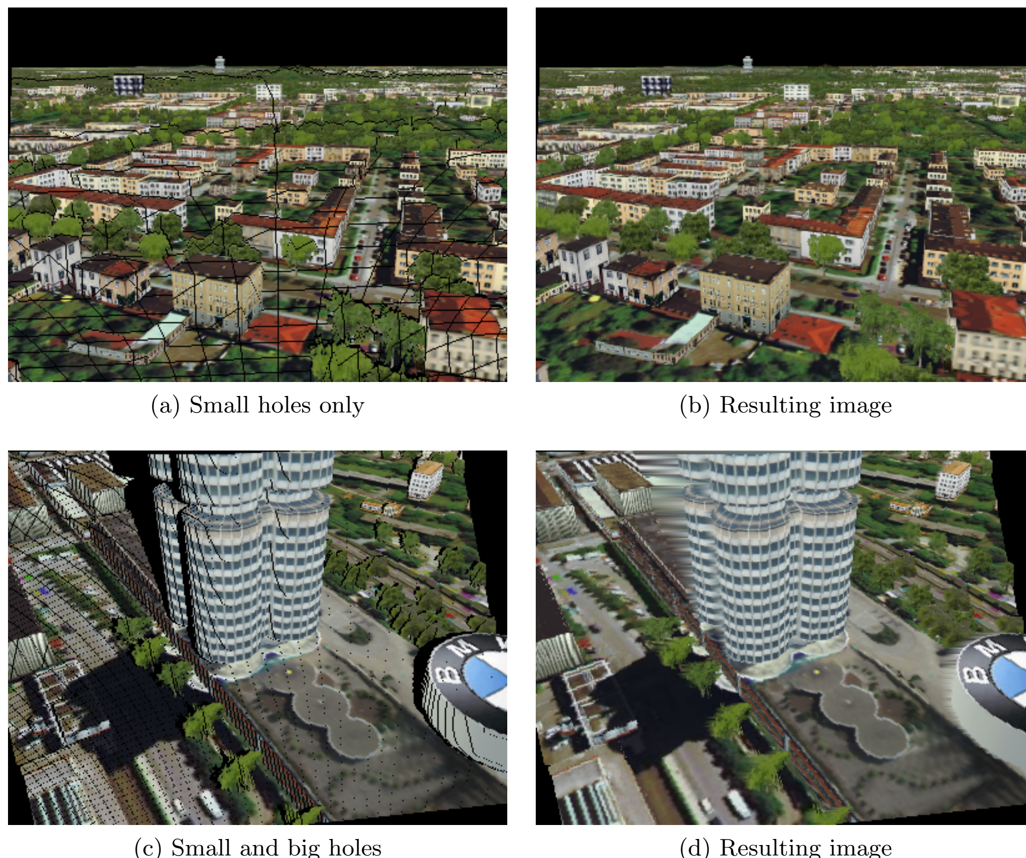
The hole-filling itself proceeds in two stages: First, single "missing rows" are filled. This is accomplished by simple looping over all pixels: if a pixel has not been set but both its top and bottom neighbors have, its color and disparity are determined as the average of the values for those two pixels. Second, horizontal spans of unwritten pixels (corresponding to one or more "missing columns") are filled. This is accomplished by looking for two pixels at the left and right corner of a run of unwritten pixels, and interpolating their color and depth values linearly to fill the pixels inbetween. Pseudocode for the whole process is given as algorithm 1 (Note that the two stages are performed simultaneously, so only one pass over the destination image is necessary). The loop body is executed exactly once for each pixel, and the number of pixels filled by the horizontal span interpolation is bounded by the number of unwritten pixels, which is again bounded by the number n of pixels in the destination image; hence the algorithm runs in $\Theta(n)$ time.

Algorithm 1 Fill holes after warping

for $y = 0$ to $height - 1$ **do**
  $s \Leftarrow width$ // Make sure first written pixel does not produce a valid span.
  for $x = 0$ to $width - 1$ **do**
    if $written(x, y-1) \wedge written(x, y+1) \wedge \neg written(x, y)$ **then**
      $color(x, y) \Leftarrow (color(x, y-1) + color(x, y+1))/2$
      $disparity(x, y) \Leftarrow (disparity(x, y-1) + disparity(x, y+1))/2$
      $written(x, y) \Leftarrow$ **true**
    end if
    if $written(x, y)$ **then**
      if $s < x - 1$ **then**
        Fill horizontal span between (s, y) and (x, y) (exclusive), linearly interpolating color and disparity.
      end if
      $s \Leftarrow x$
    end if
  end for
end for

Figure 2.4 shows the results of applying this algorithm to two sample frames from the aformentioned test sequence. 2.4a has only small holes and disocclusions, and the resulting image 2.4b has decent quality. 2.4c has large disocclusion holes, and 2.4d has very visible artifacts as a result, but the overall output quality is still acceptable given the algorithms' simplicity and speed.

However, using warping like this for prediction has one serious drawback: Snapping warped pixel coordinates to the integer grid effectively introduces jittering into the sampling process, which manifests as noise in the signal, especially near edges and other high-frequency components in the image. As a result, the difference between predicted and actual image typically contains a fair amount of noisy high-frequency content, which



(a) Small holes only

(b) Resulting image

(c) Small and big holes

(d) Resulting image

Figure 2.4.: Two warped sample frames before and after hole-filling using algorithm 1.

is quite expensive when using wavelets or block-based transforms. It also causes a staircasing effect that is especially visible near round objects, e.g. the BMW logo and the outlines of the buildings on the left side of it in figure 2.4d.

Since these problems are primarily caused by forcing pixel positions to lie on an integer grid, the obvious solution would be to introduce subpixel precision; however, this would require either a more expensive reconstruction filter that can make use of subpixel position information directly (e.g. splatting with precomputed splats for various subpixel offsets) or a larger framebuffer and additional filtering (effectively rendering in subpixel resolution, then downsampling afterwards). Both methods are more expensive and quite unattractive. I propose a simpler solution: As before, let (u_1, v_1) be the source pixel coordinates, and (u_2, v_2) the destination coordinates obtained by warping. After warping the source pixel, the Jacobian $\mathbf{J}$ of the warping equation at (u_1, v_1) is computed. Then the source image gets sampled (using a simple bilinear filter) at the position

$$\begin{pmatrix} u_1 \\ v_1 \end{pmatrix} + \mathbf{J}^{-1} \begin{pmatrix} \mathrm{round}(u_2) - u_2 \\ \mathrm{round}(v_2) - v_2 \end{pmatrix} \tag{2.4}$$

and the resulting color is used to plot the destination pixel. In effect, the difference between the computed destination position and the one that is actually used is approximately back-projected into the source image, which has a regular representation and is easy to sample at fractional coordinates.

Assuming that $\delta(u, v)$ is constant in a neighborhood around (u_1, v_1), and writing $r(u_1, v_1)$, $s(u_1, v_1)$ and $t(u_1, v_1)$ as r, s and t respectively for brevity, $\mathbf{J}$ can be determined directly from equation 2.3 using the quotient rule:

$$\mathbf{J} = t^{-2} \begin{pmatrix} w_{11}\,t - w_{31}\,r & w_{12}\,t - w_{32}\,r \\ w_{21}\,t - w_{31}\,s & w_{22}\,t - w_{32}\,s \end{pmatrix}$$

The determinant of $\mathbf{J}$ is given by

$$\det(\mathbf{J}) = t^{-4}\ [(w_{11}\,t - w_{31}\,r)(w_{22}\,t - w_{32}\,s) - (w_{12}\,t - w_{32}\,r)(w_{21}\,t - w_{31}\,s)]$$

which simplifies to

$$\det(\mathbf{J}) = t^{-3} \begin{vmatrix} w_{11} & w_{12} & r \\ w_{21} & w_{22} & s \\ w_{31} & w_{32} & t \end{vmatrix} = t^{-3} \begin{vmatrix} w_{11} & w_{12} & w_{11}\,u_1 + w_{12}\,v_1 + w_{13} + w_{14}\,\delta(u_1, v_1) \\ w_{21} & w_{22} & w_{21}\,u_1 + w_{22}\,v_1 + w_{23} + w_{24}\,\delta(u_1, v_1) \\ w_{31} & w_{32} & w_{31}\,u_1 + w_{32}\,v_1 + w_{33} + w_{34}\,\delta(u_1, v_1) \end{vmatrix}$$

and application of determinant identities leads to

$$\det(\mathbf{J}) = t^{-3}\ (\det(H) + \delta(u_1, v_1)\det(G))$$

where

$$\mathbf{H} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix} \quad \text{and} \quad \mathbf{G} = \begin{pmatrix} w_{11} & w_{12} & w_{14} \\ w_{21} & w_{22} & w_{24} \\ w_{31} & w_{32} & w_{34} \end{pmatrix}.$$

The inverse of $\mathbf{J}$ can thus be explicitly given as

$$\mathbf{J}^{-1} = \frac{t}{\det(H) + \delta(u_1, v_1)\det(G)} \begin{pmatrix} w_{22}\,t - w_{32}\,s & w_{32}\,r - w_{12}\,t \\ w_{31}\,s - w_{21}\,t & w_{11}\,t - w_{31}\,r \end{pmatrix} \tag{2.5}$$

$\mathbf{G}$ and $\mathbf{H}$ solely depend on the warping matrix $\mathbf{W}$, so their determinants only need to be computed once per frame, and r, s, and t are already computed to determine u_2 and v_2 anyway. Hence the cost of evaluating equation 2.4 per pixel is about as high as the warping equation itself, but no additional passes or memory accesses are required. Compared with the cost of warping at subpixel resolution, warping more pixels to get less holes, or using splatting for reconstruction, this is still quite reasonable.

The modified sampling process (simply referred to as "warping with sample shifting" in the following) works well for colors and notably reduces high-frequency spikes near edges without unreasonable blurring of the depth image. For that very reason, however, it is not suitable to process disparity values, since sharp depth discontinuities are important features for warping that should be preserved if possible; hence, disparity values

are best point-sampled. To prevent jittering noise from accumulating, a simple blur filter can be used. To prevent edges from getting smoothed out, pixels are not blurred if the partial derivative of δ in the u_2 or v_2 direction is above a threshold. All this is only relevant if the image is to be used as a reference image for warping later, since the depth image is never directly visible to the user.

Results using the different warping methods are shown in figure 2.5 on page 30. (a) shows the actual target frame to be predicted, and (b) is the previous frame, which is the source frame for warping; (c) shows the result obtained by normal warping, (d) is the difference to the actual frame. Subfigures (e) and (f) show the same using warping with sample shifting. Finally, (g) and (h) were generated using bilinear patch reconstruction and are included for reference. As is clearly visible, using sample shifting improves the prediction accuracy notably, as edges in the difference images are less pronounced and the overall intensity level is lower. By contrast, the additional improvement from using bilinear patch reconstruction is relatively minor, and comes at the expense of significantly higher CPU cost. An objective evaluation of the different warping variants will be given in the following section.

2.3. Evaluating the efficiency of warping as a prediction model

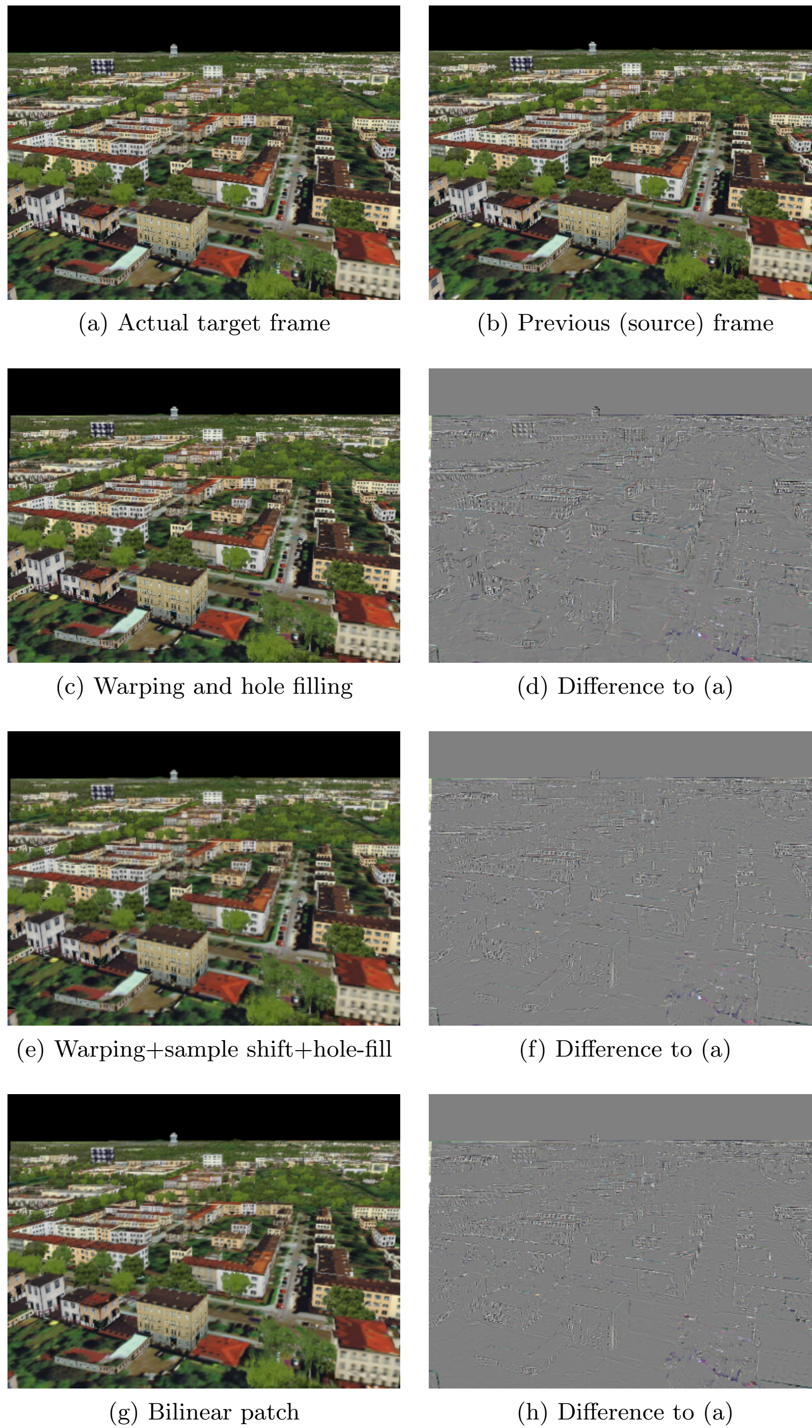
The main question when replacing motion compensation with warping for video coding is simply, *does warping-based prediction provide improved coding efficiency*, that is, does the extra CPU time spent doing warping[2] instead of simply performing motion compensation pay off? Since warping is much slower on the decoder side, it has to deliver a significant gain in quality per bit (i.e. better quality at the same bit rate, or the same quality at a lower bit rate) to be seriously pursued.

To answer this question, two experiments will be performed: In the first one, the quality of predicted images given perfect reference data (that is, without any lossy coding) will be compared. It will also show what kind of quality improvement, if any, can be expected over normal motion compensation—ignoring how much information needs to be encoded to get that quality. In the second experiment, the amount of "extra data" (that is, non-color data) will be compared. That is, it will be calculated how many bits are needed to store motion vectors, and how many bits can be expected to be spent on the per-pixel depth (disparity) information required for warping.

2.3.1. Prediction efficiency in ideal circumstances

For both motion compensation and warping, the difference images to be encoded look very similar, namely like the difference images in figure 2.5: particularly, energy is mostly concentrated near edges in the target image, with very little low-frequency content. Since there are no noticable structural differences between the difference images obtained using

[2]This is obviously true for decoders; but even for encoding, modern CPUs have special instructions to speed up motion estimation, while there is no such hardware support for warping.



(a) Actual target frame

(b) Previous (source) frame

(c) Warping and hole filling

(d) Difference to (a)

(e) Warping+sample shift+hole-fill

(f) Difference to (a)

(g) Bilinear patch

(h) Difference to (a)

Figure 2.5.: Comparison between warping variants.

the two algorithms, I use the PSNR (peak signal-to-noise ratio) between the actual and predicted luminance channels as quality metric. It is defined as:

$$PSNR = 10 \cdot \log_{10}\left(\frac{w \cdot h \cdot M_I^2}{\|I - P\|_2^2}\right) \quad \text{[dB]} \tag{2.6}$$

where w and h are the width and height of the images, respectively, M_I is the maximum possible pixel value in the source data (255 when using 8 bit per color channel), and I and P are vectors containing the per-pixel luminance values of the actual and predicted images, respectively.

A higher PSNR means that the predicted image approximates the actual image more faithfully; hence, less residual information need to be coded to achieve the desired target quality. Only luminance is compared because the video codec used for comparison, Dirac [BBC08], always subsamples chrominance channels; subsampling is less attractive for warping, because the relatively expensive warping and reconstruction steps would have to be done twice, once in full resolution for luminance data, and again in subsampled resolution for chrominance.[3] In any case, both Dirac and the warping-based methods treat luminance and chrominance the same way (apart from the subsampling), so the results are still assumed to be representative.

The test setup is as follows: The various warping variants operate on sequences of umcompressed 400×300 pixel frames with 8 bit per RGB color channel and 16 bit depth information per pixel. Position, orientation and all necessary parameters for the planar pinhole camera model (as necessary for warping) are stored alongside each frame. The results for motion compensation are obtained using Dirac, which actually implements overlapped block motion compensation (OBMC). As the name suggests, this scheme uses larger blocks that overlap each other; a weighting function is used such that for each pixel in the destination image, the weights for all the blocks covering that pixel sum to 1. OBMC has the advantage that there are no artificial discontinuities at block boundaries, which would otherwise introduce a bias into the comparison. For encoding, the Dirac reference encoder [BBC] is used in lossless mode (so that all reference frames are "perfect") and with L2 frames (corresponding to MPEG B-frames) disabled, making sure that each frame is predicted from its immediate predecessor, as with the warping-based methods. For Dirac encoding, frames were converted to the YCbCr color space using the tools supplied with the reference encoder; the original RGB input frames and the warper output frames are converted into the same color space and the Y channels are compared. The Dirac encoder was modified to output the results of the motion compensation process. Finally, keyframes and frames without motion compared to the previous frame are omitted from the evaluation; since all methods have lossless reference data, there is no difference between actual and "predicted" image data in those cases, which would result in an infinite PSNR.

[3] Alternatively, one could do the warping with non-subsampled RGB/YCbCr, upsampling chroma information before warping then downsampling it again afterwards; apart from the extra overhead, such a process would also tend to accumulate resampling errors over time.

The first test sequence is a flight over a (completely diffuse) 3D dataset of Munich generated from aerial images and land register data provided by RSS GmbH, rendered with the current version of the SCARPED terrain rendering engine as first described in [WMD+04]. The sequence is captured from a short interactive session with the system, 823 frames long, and inteded to be played at 30 frames/second. Because of its interactive nature, there are a few pauses while the user switches control modes or repositions the mouse and the image stays the same for a few frames. These frames are ignored in this comparison. In general, the sequence consists of short fully interactive segments where the user zooms or repositions the camera, and continuous flight segments where the user has double-clicked on a point of interest and the system performs a flight towards that point.

Test sequence 2 is a 900 frame long (again, 30 frames/second) camera flight through the Utah Fairy Forest scene (a standard test scene for real-time raytracing, available at `http://www.sci.utah.edu/~wald/animrep/`), or more precisely its first frame; the camera motion is defined as a spline, and only the diffuse textures were used, with both specular effects and bump mapping disabled. This was done intentionally, to provide optimal input for a warping-based method, since the purpose of this evaluation is to find out how well warping-based prediction can perform given perfect conditions.

The different methods tested are:

OBMC: overlapped block motion compensation as implemented by Dirac.

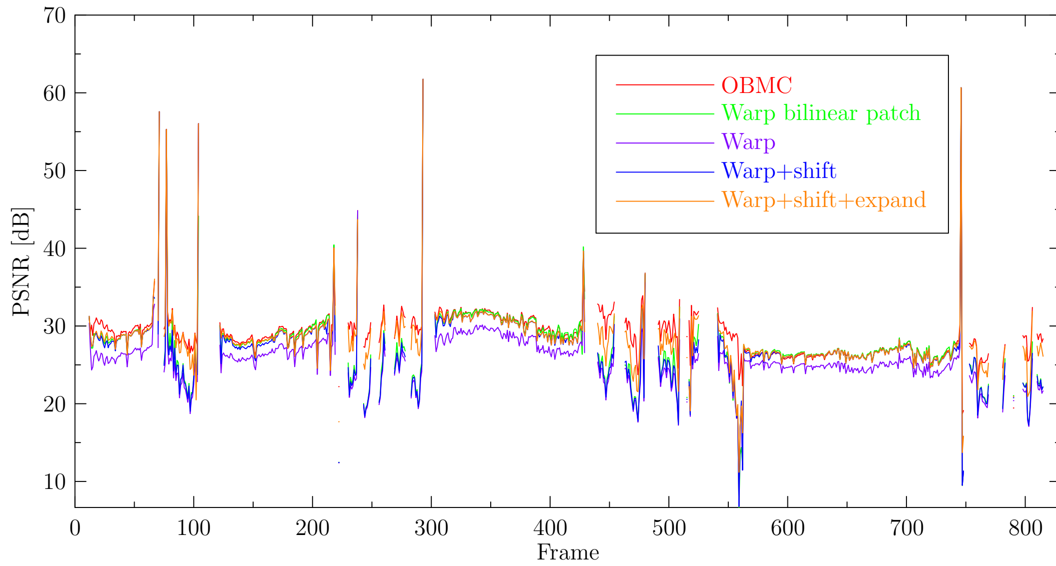
Warp bilinear patch: warping with bilinear patch reconstruction (actually, not proper bilinear patch; the warped quadriliterals are split into two triangles, and colors are linearly interpolated within triangles. This is how e.g. an implementation using 3D hardware would work).

Warp: warping with single-pixel plotting and hole-filling using algorithm 1.

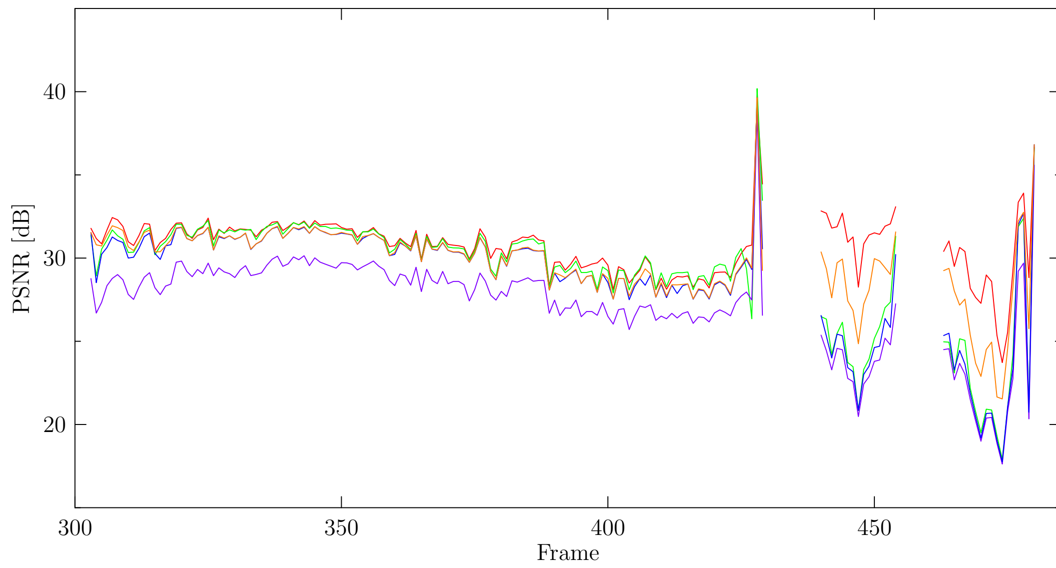
Warp+shift: same as above, but including the Jacobian-based sample shifting described in section 2.2.

Warp+shift+expand extends the first written pixel in each scanline towards the left and the last written pixel towards the right; also, the top- and bottommost scanlines containing written pixels are repeated above and below, respectively. This provides more reasonable "default" colors near the screen edges than leaving everything black. This was included because motion compensation based methods never default to black pixels; the black pixels produced by warping when no data is available reduce PSNR notably. The simple expansion process greatly improves PSNR on high-motion frames and is very cheap.

The results of the comparison using test sequence 1 are shown in figure 2.6. Subfigure (a) shows PSNRs for the whole sequence, while (b) shows frames 303–480 (which are typical for the whole sequence) in more detail. It is immediately obvious that most methods are at a similar level of quality for most of the time, except for "Warp" (direct



(a) Complete sequence



(b) Zoomed view: frames 303–480.

Figure 2.6.: Results obtained with test sequence 1.

warping without sample shifting), which loses about 2–2.5 dB compared to the other methods for nearly all frames. This confirms the observation in section 2.2 that the sample shifting process improves quality significantly. The mean difference between the PSNR obtained by bilinear patch reconstruction and the PSNR obtained by the simpler warping with sample shifting process is 0.27108 dB, with a standard deviation of about 0.67228 dB—mainly caused by a few outliers; removing samples more than 3 standard deviations from the mean results in a new mean of 0.29947 dB with a standard deviation of 0.32645 dB. This is, again, in line with the expectations: warping with sample shifting does produce worse quality than full bilinear patch reconstruction; but warping with sample shifting and hole filling "only" approximately doubles the work per warped point and does an extra pass over the image. Bilinear patch as typically implemented not only needs to evaluate the warping equation for all $w \times h$ pixels, it also has to rasterize $2(w-1)(h-1)$ triangles. The combined hole-filling/sample shifting method thus delivers reasonable quality at far lower computational cost, as intended.

The results also show, however, that the overlapped block motion compensation-based prediction wins outright most of the time, and is very close to the best warping-based methods in the cases where it doesn't. This is especially obvious in the fully interactive parts (e.g. the right part of figure 2.6b), where the camera moves quite rapidly. As is clearly visible, the "Warp+shift+expand" method significantly outperforms all other warping-based methods here, despite being roughly on par with "Warp+shift" or bilinear patch reconstruction for smooth motions. As the only difference between "Warp+shift" and "Warp+shift+expand" is changed behavior for pixels where no information is available from warping, this highlights a fundamental weakness of warping when compared with motion compensation: there is no sensible default behavior for areas that simply weren't visible in the reference frame. While motion compensation still has the option of simply using a block in the reference image that looks similar (even though the corresponding motion may not be physically plausible), all warping-based methods are forced to a default behavior of extrapolating—in some way or another—from the information available. While it is certainly possible to improve on the very simple "expand" technique, any improved technique that does not store extra information for this purpose will still be "flying blind", while motion compensation-based encoders can actively look for a good match in existing image data—without needing additional motion vectors. Thus, motion compensation has a fundamental advantage over warping when the camera angle changes drastically.

In any case, warping does not produce significantly better reference images than obtainable with simple motion compensation. While any of the methods given can certainly be improved, no small change is likely to result in a dramatic increase in quality; in fact, it seems doubtful that it would even be possible to match the quality of motion compensation when fast camera motions are involved. Since all of the warping-based methods also have significantly higher computational demands on the decoder side, designing a warping-based codec is simply not attractive.

Figure 2.7 deals with test sequence 2 and shows similar results: again, the warping-based methods fail to provide any significant improvement over motion compensation,

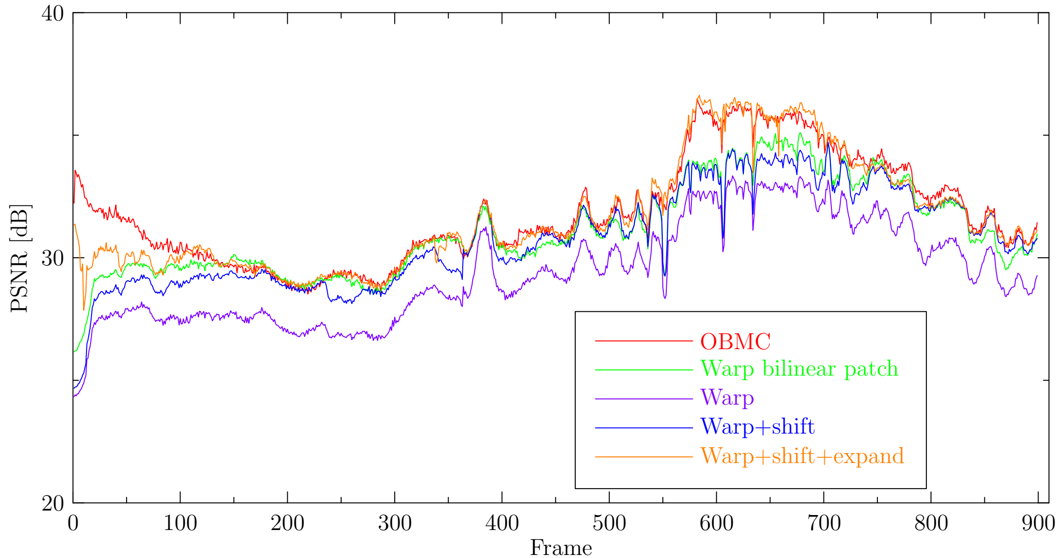


Figure 2.7.: Results obtained with test sequence 2.

and are in fact noticably worse near the beginning. Since test sequences 1 and 2 are very different in structure, and test sequence 2 was even modified to better match the expectations of the warping model (specular highlights and reflections disabled, bump mapping disabled), this provides additional evidence that apparently, there are no big gains to be expected from using warping for video coding—not for high bitrates anyway, where the data is dominated by residual coefficients already, and any increase in their magnitude costs dearly.

2.3.2. Relative cost of disparity data and motion vectors

However, the question of how much storage is needed for disparity information remains; if this is low compared to the amount of bits spent on motion vectors in a motion compensation based codec, warping is still interesting for low bitrate applications, where an increase in computation time is typically tolerable as long as it produces reduced distortion at the same bit rate.

To investigate this question, I implemented a very basic warping-based video codec. Intra coding is a simplified version of Dirac; disparity information is stored as a separate plane, just as luminance and chrominance information. Colors are transformed into the YCoCg color space for coding, a relatively recent variant on the YCbCr/YUV color spaces typically used for image/video coding; YCoCg is used because the transforms RGB $\leftrightarrow$ YCoCg are very simple, requiring only integer additions and shifts, and because it results in better coding gain than other YUV variants for natural images [MS03].

Inter coding uses the same methods, but subtracts the warped version of the previous (reference) frame from the frame to be coded first. The biorthogonal LeGall 5/3 wavelet

is used for all frames; Dirac uses the Cohen-Daubechies-Feauveau 9/7 wavelet for intra coding, which is advantageous since keyframes are typically smoother (the residuals in inter-coded frames consist mostly of high-frequency components, and smoother lowpass filters don't help much; hence the smaller-support and thus faster to evaluate wavelets are preferred there). Since only one keyframe (the first frame) per sequence is ever coded for this test, the warping-based coder uses LeGall 5/3 wavelets everywhere for simplicity.

Simple uniform scalar quantization is used, and the quantization factors are constant for the whole frame and fixed beforehand (proper rate control and rate/distortion optimization would have involved a significant extra amount of work – probably overkill for this test). The quantized values are then passed to an arithmetic coder, where the model used is again based on Dirac. Test sequences 1 and 2 are used again; both were encoded twice with Dirac, with target bit rates of 1500 and 300 kbit/second, respectively. The sequences were also encoded with the simple warping-based video coder just described, and the quantization settings adjusted to (very) roughly match the size of the Dirac-encoded files; disparity quantization is set up to minimize the total size of coded color and disparity information. What is compared is simply the percentage of "payload" (i.e. non-header etc.) data devoted to coding motion vectors (for Dirac) or disparity/camera information (for the warping-based coder). The visual quality of the videos is very different—the warping-based codec produces results much worse than Dirac—no doubt in large part due to the missing rate control in the warping-based codec, which results in poor overall bitrate allocation. However, the absolute amount of bits spent on motion vectors and disparity information grows quite slowly for increasing bitrates with both codecs; it is assumed that the reported ratios are unlikely to change significantly with a better rate control strategy. The results are shown in table 2.1.

	Test sequence 1		Test sequence 2	
Codec	1500kbit/s	300kbit/s	1500kbit/s	300kbit/s
Dirac	9.726%	32.140%	10.219%	26.204%
Warping-based	5.384%	14.990%	3.050%	12.357%

Table 2.1.: Percentage of payload data spent on motion information for Dirac and the warping-based codec.

.

So, for the given test sequences and under the simplifying assumptions made earlier, warping does seem to require notably less storage for motion information than motion compensation does, even taking the bits used for disparity information into account – a reduction by a factor of more than 1.8 for all the test scenes. These results are to be taken with a grain of salt, since there were several big assumptions made; however, a proper evaluation is virtually impossible without implementing a complete warping-based video codec (including somewhat more complex coding and rate control) and comparing it

"on equal terms" with a state-of-the art motion compensation based codec. The data does, at least, provide some hints that such a codec would be feasible, and might even outperform conventional video codecs in some cases.

In any case, the main result of this section remains that warping is not inherently superior to motion compensation as a model, not even when perfectly diffuse source data is used; hence, paying the higher per-pixel cost of warping is probably overkill. More importantly, a codec based on warping is inherently limited to static 3D scenes, while motion compensation easily deals with animation. Any codec with such limited field of application is unlikely to succeed if it only provides an incremental improvement over existing (and widely implemented) general-purpose solutions.

3. Improving video compression and rendering

The previous chapter talked about warping and one specific way to apply it to the problem of encoding videos of a single camera moving through a static 3D scene. This chapter will take a step back and look at the “bigger picture” of server-side rendering: to be feasible in practice, any server-side rendering system must be able to handle a decent numnber of clients, and do so with predictable (and preferably low) cost in network bandwidth and CPU/GPU time.

Network bandwidth requirements for a server-side rendering application are necessarily relatively high: even with modest resolutions (e.g. 320×240 pixels) and low frame rates (20 frames/second), video data needs over 200 kbit/second to display high-motion scenes with acceptable visual quality (and virtually any camera movement results in a lot of motion, because moving the camera changes the whole frame). Furthermore, state of the art codecs have to be used, because older ones perform significantly worse (in terms of resulting distortion) for low-bitrate applications.

Also, rendering several views per frame on the server side (one for every client) is expensive in terms of CPU/GPU time; having to additionally encode several compressed streams at the same time adds to the computational load, and video encoding has high cost all by itself already.

Barring significant improvements in video coding technology, there is little to be done about the network bandwith requirements of server-side rendering; as section 2.3 has shown, existing codec technology is already well-suited to the task, and revolutionary breakthroughs are unlikely. As such, the bandwith requirements are a cost factor that simply has to be taken into account when evaluating the suitability of server-side rendering to a certain application.

The computational costs are not quite as fixed, however, and it seems worthwhile to check whether there might be ways to improve cost of both compression and rendering using the knowledge that the videos in question were generated from flights through 3D scenes. This chapter will focus on compression and one particular way of speeding up rendering.

3.1. Warping-based motion estimation

The results in section 2.3 show that a direct warping-based codec is unlikely to produce a significant improvement over already existing methods, and might indeed turn out much worse depending on the circumstances. Conventional video codecs are well-tuned

and have widespread hardware and software support; in particular, even some otherwise very weak devices such as MP3 players can play back videos because they have hardware decoders.

This section will explore ways of improving conventional video encoding, assuming that per-pixel depth information and a description of the camera setup is available to the encoder. Both are readily available in our setting: the camera parameters are passed as input to the renderer anyway, and rasterizers and raytracers can both easily output depth information without performing additional work.

The main idea is to improve video compression quality and, possibly, speed by using warping to determine motion vectors, instead of performing conventional motion estimation. The video format used for comparision is H.264 [ITU05], because it is both an ISO and ITU standard and quite popular in applications.

H.264 allows having more than one motion vector per macroblock: the 16×16 pixel blocks can be subdivided further, down to sixteen blocks of 4×4 pixels each if necessary. The resulting sub-blocks are called *partitions*. The optimal partitioning for each macroblock is determined during encoding. Instead of performing the default motion search procedure, the midpoint of each partition is warped to obtain the corresponding point in the reference image.[1] The difference between the position of the midpoint in the destination image and the position in the reference image is used as the motion vector; since warping typically results in fractional coordinates anyway, this motion vector can be determined with subpixel precision.

The per-partition cost of this procedure is quite low: one evaluation of the warping equation (11 multiplies, 9 additions, 1 division), two floating point to integer conversions, and two integer subtractions to turn the reference frame coordinates into motion vectors relative to the destination block midpoint.

Using the obtained motion vectors directly is possible, but it still makes sense to give the encoder a little more freedom: using a motion vector that is off by some subpixels may reduce blurring, use less bits with no significant quality difference, or be otherwise beneficial. Thus it still makes sense to try a few motion vectors in the direct neighborhood of the calculated motion vector, and pick the best one.

To do this efficiently, a very simple trick is used: care is taken to make sure that rounding errors made when converting the fractional motion vectors to integers are one-sided. This is done by always rounding downwards. The obtained motion vector (m_x, m_y) will thus tend to be slightly too small in both components. The code then tries all four members of the set $\{(m_x + i, m_y + j) \mid i, j \in \{0, 1\}\}$ as candidate motion vectors, and the best one is used. This requires less than half as many tests as the more obvious procedure of rounding motion vectors towards the closest integer and then trying candidate motion vectors of the form $\{(m_x + i, m_y + j) \mid i, j \in \{-1, 0, 1\}\}$, but results in very similar quality.

[1] There might not be such a point, if the 3D position of the destination point is in front of the reference camera's near plane; in that case, the fallback solution is to perform the regular motion search procedure on that partition.

It is still possible to perform regular subpixel refinement (like one would do with motion vectors obtained from a direct search procedure) on the resulting motion vectors; just as with normal motion estimation, this increases quality by allowing the codec to make a better rate/distortion tradeoff, at the expense of increased runtime. But even without subpixel refinement, the obtained motion vectors have good quality.

3.1.1. Implementation

I implemented the method as described above by modifying `x264` [AM+07], an open-source H.264 codec library that also performs quite well in H.264 encoder comparisions. I also modified the supplied standalone encoder application to make the warping-based motion estimation procedure available using a commandline option. The changes fall into several categories; I will describe each of them in turn.

Support for per-pixel depth data and camera setup information in `x264`

To use warping-based motion estimation, camera and per-pixel depth information has to be available to the motion estimation code. `x264` does not normally deal with such data, so support for it had to be added in the relevant places.

The application supplies images to the codec by passing it a `x264_picture_t` structure. Its definition was updated to include a description of the camera position and parameters (described by the `x264_camera_t` structure), and by adding a pointer to per-pixel depth information to `x264_image_t`, which contains a description of the actual image data.

Once images have been submitted to the encoder, they become frames in the video sequence and are described using `x264_frame_t`; support for camera and depth information had to be added here as well. Finally, the functions that allocate pictures, convert pictures to frames or allocate, copy, and free frames had to be updated to handle the new fields correctly.

Reading depth image sequences

To get depth image sequences into the encoder in the first place, a custom file format was necessary, because none of the standard video formats support per-pixel depth data (or any non-color data, for that matter). I used the `dvid` format, which was developed earlier to export image sequences from the SCARPED renderer.[2] The format consists of a short header which describes the width and height of individual images as well as the frame rate, followed by the raw data for every frame: 24 bit RGB color data for each pixel, followed by 16 bit depth data for each pixel, followed by a description of the camera setup.

A `dvid` reader was added to the command-line encoder to work with these depth image sequences. The codec expects data to be submitted as subsampled YCbCr color data; the conversion from RGB is performed on the fly, and the resulting pictures (including camera description and per-pixel depth data) are submitted to the encoder.

[2]Specifically, the test sequences in section 2.3 were recorded this way.

Adding warping-based motion estimation

Warping has two main components: calculation of the warping matrix and evaluation of the warping equation. The computation of the warping matrix only has to be performed once per frame, or more precisely, once for every pair of reference and destination frame. I added the module `me_warping.[ch]` to `x264`, which encapsulates the warping details. The function `x264_pointwarp_init` computes the warping matrix for a given pair of frames; it is called from `x264_reference_build_list`, where the list of reference frames for a given target frame is determined.

The warping equation is evaluated by the function `x264_pointwarp_block`: it determines the motion vector for a given partition (which is described by the x, y coordinates of its top-left pixel and its width and height) by warping the midpoint, as described in section 3.1. It is called from `x264_me_search_ref`, the main motion estimation function in `x264`, when warping-based motion estimation is enabled. If the warping process yields a motion vector,[3] its cost and that of other "close" motion vectors are computed, again as described in section 3.1—the cost function is a weighted sum of the number of bits required to encode a motion vector and a vector norm of the difference between the reference block and the block to be coded. H.264 predicts the motion vector for each partition from the motion vectors in adjacent partitions that have already been coded; this predicted motion vector and the null motion vector are cheaper to encode than regular motion vectors, so their cost is also evaluated. The overall best motion vector is used.

If the warping process was not successful, normal motion search is performed as a fallback solution; this only occurs very rarely in practice, however. Finally, additional subpixel refinement of the obtained motion vectors can be performed, if desired. This improves quality but comes at an extra expense in CPU time.

Rate control issues

Prior to actually encoding each block, `x264` makes a first pass over a subsampled version of the input frame to estimate the amount of motion and the magnitude of residual data. This is used to decide which frame type to use (for example, after a scene change, an I-frame is typically cheaper than a P-frame) and to determine the quantization factors for different blocks in the image so that the target bitrate is met.

Since the rate control pass works with a subsampled version of the image, it is necessary for the warping-based motion estimation code to know when it is being called from the rate control code: Input coordinates have to be scaled to access the corresponding pixel in the disparity image, and the resulting motion vectors have to be divided by the subsampling factor. If this is not done properly, the actual cost of each frame will be quite far from the estimated one; this causes, among other things, large frame-by-frame variations in bitrate (when a frame does not use its allocated budget, the next few frames will be given correspondingly more or less bits), and a notable decrease in visual quality, because the assignment of quantization factors uses incorrect motion information.

[3] As mentioned beforehand, warping may result in a point that is in front of the reference camera near plane; such points do not result in meaningful motion vectors.

Codec interface

Since the warping-based motion estimation code still may fall back to a conventional motion estimation procedure,[4] the use of the warping-based code is controlled by a binary flag, instead of adding it as an alternative motion estimation method. Thus, even when the user chooses to enable warping-based motion estimation, it is still possible to select the fallback motion estimation method separately. The modified command-line encoder enables the warping-based code when the command line option `--warpme` is given; all other motion estimation options (like the choice of ME method, or the quality of subpixel refinement if desired) can be controlled independently.

3.1.2. Results

The modified version of `x264` was tested on several different sequences and with different encoder parameters (target bitrate, number of reference frames per target frame, and accuracy of subpixel motion estimation) to evaluate the efficiency of the proposed warping-based motion estimation process.

The test procedure is as follows: First, the input sequence is read once in full, to make sure it is in the filesystem cache so that I/O bandwidth does not affect the results. Then, the encoder application is run with the specified settings, encoding from the input file to the null device. After completion, several statistics about the encoded sequence are output, including objective quality metrics and the speed of encoding as measured in encoded frames per second. To account for random variations due to background processes, each run is repeated three times, and the median speed is used. The quality of the encoded sequence is measured both using the PSNR of the luminance channel, defined on page 31, and the structural similarity index (SSIM) between the luminance images. The latter is introduced in [WBSS04] and tries to take perceptual effects of the human visual system into account to produce an objective quality metric that has a stronger correlation to perceived similarity than the mean square error (MSE) and derived metrics such as the PSNR do. SSIM indices range between 0 and 1, where 0 would indicate that the two images are completely uncorrelated, while 1 means that they are identical.

All tests are run on a notebook with an Intel Core2Duo T7500 processor (clocked at 2.2 GHz) and 3 GB of RAM. In addition to the variable parameters, the encoder is always run using the `--threads 2`, `--partitions all` and `-b 0` commandline options: the first one to make use of the dual-core processor, the second to enable all partition types (by default, lesser-used partition types are not considered) and the last one disables B-frames, since using them actually decreased overall quality in these tests.[5]

“Terrain” sequence

The first test sequence, “terrain”, is the same as test sequence 1 in section 2.3 (cf. page 31). The test scene is a 3D model of Munich; the geometry for individual houses

[4] This is very rare in practice, and doesn’t occur at all with most of the sequences tested below.
[5] They generally improve quality when two-pass encoding can be used.

is quite simple (mostly extruded 2D paths), but since there is often a large number of houses visible at the same time, the overall amount of geometric detail visible in a typical frame is relatively high. The camera motion is a mixture of user interaction and computer-generated smooth flights between different points of interest.

Results are shown in table 3.1. “Bitrate” is the target bitrate passes to the encoder, “Ref” is the number of reference frames to use (higher numbers improve quality but cost extra CPU time), “WarpME” indicates whether warping-based motion estimation was used or not and “SubME level” selects the quality of subpixel motion estimation: 1 disables it altogether, 2 performs a few subpixel refinement iterations, and successive levels add a higher number (and better accuracy) of refinement steps, up to 5 which is the default. Levels 6 and 7 perform full rate-distortion optimization instead of minimizing the heuristic cost function; this improves quality but significantly increases CPU usage and is probably impractical for real-time encoding. The columns “SSIM-Y”, “PSNR-Y” and “Frames/s” report the results obtained with the given parameter set.

A first surprise is that turning on warping-based motion estimation, all other parameters being equal, does not improve speed. Further experimentation revealed that the difference is caused by evaluating the warping equation: This cost is nearly constant and paid for every partition, whereas the normal motion estimation search patterns use predicted motion vectors and early-outs to minimize average-case runtime. The actual time spent computing the cost for candidate motion vectors is very similar in both cases, but the warping-based method has higher overhead because the warping equation needs to be evaluated.

However, warping-based motion estimation does produce a notable improvement in PSNR in *all* tests, and in SSIM for all but the 300 kbit/s tests. In fact, for all tests, enabling WarpME yields better results (in terms of PSNR) than those produced without warping and using the *next higher* listed level of subpixel refinement. SSIM results are not quite as spectacular, but enabling warping still produces significant improvements with bitrates of 500 and 1000 kbit/s: while not surpassing the SSIM indices obtained using the “next higher level” of subpixel refinement, they are nevertheless quite close.

These results are very consistent over the quite large range of different parameters given, indicating that warping results in either notably improved quality for very little extra CPU time, or matches a given target quality with substantially lower CPU cost—for this test sequence, at least.

“Fairy” sequence

The second test sequence, “fairy”, corresponds to test sequence 2 in section 2.3. It shows a model of a fairy in front of a forest backdrop (which includes modelled mushrooms, grass, and trees). There are large variations in the size of geometric features: for example, individual grass blades are represented as geometry. The camera plays back a motion along a spline that was created in a 3D modelling application.

Corresponding results are shown in table 3.2. Only the results when using one reference frame are reported in the following; in all of the sequences, and both with and without warping, using multiple reference frames results in better quality at the cost

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.7949808	26.140	133.34
300	1	yes	1	0.7918640	26.480	131.01
300	1	no	2	0.8017598	26.307	112.31
300	1	yes	2	0.7955168	26.589	111.13
300	1	no	5	0.8116736	26.586	78.73
300	1	yes	5	0.8098858	26.877	79.33
300	3	no	1	0.7926901	26.103	125.71
300	3	yes	1	0.7921847	26.485	119.71
300	3	no	2	0.8003088	26.308	103.90
300	3	yes	2	0.7942310	26.597	100.91
300	3	no	5	0.8111303	26.592	69.39
300	3	yes	5	0.8099403	26.898	68.85
500	1	no	1	0.8570027	27.950	126.00
500	1	yes	1	0.8629337	28.630	123.35
500	1	no	2	0.8638736	28.196	104.71
500	1	yes	2	0.8671046	28.783	101.88
500	1	no	5	0.8708137	28.466	70.60
500	1	yes	5	0.8728295	28.982	70.13
500	3	no	1	0.8553014	27.914	119.17
500	3	yes	1	0.8630835	28.655	111.13
500	3	no	2	0.8636627	28.218	95.59
500	3	yes	2	0.8671956	28.816	90.03
500	3	no	5	0.8714704	28.512	62.85
500	3	yes	5	0.8736568	29.044	61.24
1000	1	no	1	0.9221869	30.991	115.77
1000	1	yes	1	0.9269663	31.948	109.73
1000	1	no	2	0.9272521	31.292	92.73
1000	1	yes	2	0.9300071	32.137	88.09
1000	1	no	5	0.9308854	31.571	61.11
1000	1	yes	5	0.9323950	32.299	60.47
1000	3	no	1	0.9213025	30.956	108.83
1000	3	yes	1	0.9277248	32.011	100.33
1000	3	no	2	0.9277198	31.341	86.78
1000	3	yes	2	0.9310868	32.219	80.54
1000	3	no	5	0.9318547	31.658	55.56
1000	3	yes	5	0.9334796	32.395	54.70

Table 3.1.: Encoding results for the “terrain” sequence.

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.8544471	31.999	146.94
300	1	yes	1	0.8543651	31.967	147.32
300	1	no	2	0.8593798	32.159	126.32
300	1	yes	2	0.8589095	32.119	127.15
300	1	no	5	0.8695813	32.459	89.44
300	1	yes	5	0.8686421	32.387	90.57
500	1	no	1	0.8924078	33.747	136.82
500	1	yes	1	0.8927671	33.723	135.22
500	1	no	2	0.8974561	33.973	114.52
500	1	yes	2	0.8974329	33.935	114.97
500	1	no	5	0.9040100	34.256	79.45
500	1	yes	5	0.9029872	34.184	80.11
1000	1	no	1	0.9377240	36.744	120.76
1000	1	yes	1	0.9378214	36.714	120.50
1000	1	no	2	0.9419348	37.056	98.47
1000	1	yes	2	0.9414331	36.991	97.96
1000	1	no	5	0.9457670	37.369	66.44
1000	1	yes	5	0.9449789	37.291	66.36

Table 3.2.: Encoding results for the “fairy” sequence.

of slightly higher encoding time, and the behavior of warping-based motion estimation was very similar between one-reference-frame and multiple-reference-frame tests. For the curious, the full results are available in appendix B.

The numbers themselves are quite different than those obtained using the “terrain” sequence. Here, warping is actually slightly faster in a large number of cases, but produces slightly worse results in general, though the difference is small: always lower than 0.075 dB for the PSNR ratings—contrast with the consistent improvement of over 0.25 dB for *all* tests run on the terrain dataset, with warping producing a gain exceeding 0.9dB several times for the higher bitrates. The SSIM indices are, similarly, quite close. So while the warping-based motion estimation yields no improvement for this scene, it does not make the results significantly worse, either. In general, the camera motions in this test sequence are quite smooth and slow, which benefits conventional motion estimation, since a search procedure is likely to find good motion vectors quickly.

“Interactive fairy” sequence

To test whether this indeed makes a difference, the same scene was rendered using a different camera motion, this time recorded from an interactive session. As a result, mo-

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.8940561	34.118	144.35
300	1	yes	1	0.8976926	34.656	141.18
300	1	no	2	0.8983479	34.358	127.15
300	1	yes	2	0.9011825	34.860	123.34
300	1	no	5	0.9061646	34.734	90.42
300	1	yes	5	0.9133149	35.303	89.16
500	1	no	1	0.9310123	36.642	133.95
500	1	yes	1	0.9349554	37.420	130.61
500	1	no	2	0.9352065	36.969	115.65
500	1	yes	2	0.9385026	37.715	112.50
500	1	no	5	0.9401657	37.339	80.00
500	1	yes	5	0.9448482	38.109	79.23
1000	1	no	1	0.9685249	40.839	119.25
1000	1	yes	1	0.9703983	41.906	115.21
1000	1	no	2	0.9713022	41.251	101.78
1000	1	yes	2	0.9723293	42.198	96.32
1000	1	no	5	0.9731321	41.572	68.17
1000	1	yes	5	0.9744583	42.550	67.68

Table 3.3.: Encoding results for the “interactive fairy” sequence.

tions are jerkier in general, and include short burts of very high-motion frames whenever the viewer “looks around”, turning the camera rapidly in the process. Rendering a video with the new camera path resulted in the “interactive fairy” sequence. Results are shown in table 3.3.

Here, observations are similar to what was already described for the “terrain” sequence: warping-based motion estimation delivers a notable gain in both PSNR and SSIM for all tests, requiring very modest amounts of extra CPU time to do so—far less than the cost of better motion estimation methods. This seems to confirm the conjecture that warping improves on conventional motion estimation mainly by determining fast motions accurately; for slow motions (in the single-pixel or subpixel range), a search-based procedure has a good chance to find local rate-distortion minima, while warping always results in a motion vector close to the “correct” one, which may not be optimal in rate-distortion terms.

“CS_Italy” sequence

To confirm this theory, another interactive test sequence is tested. This fourth and last test sequence, “cs_italy”, uses the map of the same name from the game Coun-

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.6353905	25.557	131.20
300	1	yes	1	0.6679018	26.709	129.44
300	1	no	2	0.6397960	25.668	113.39
300	1	yes	2	0.6722945	26.828	112.71
300	1	no	5	0.6493255	25.838	81.71
300	1	yes	5	0.6833704	27.024	81.24
500	1	no	1	0.7036136	26.687	120.24
500	1	yes	1	0.7314405	27.981	119.74
500	1	no	2	0.7087490	26.836	102.31
500	1	yes	2	0.7372080	28.129	100.52
500	1	no	5	0.7161530	26.995	72.09
500	1	yes	5	0.7444381	28.296	72.18
1000	1	no	1	0.7982175	28.812	106.67
1000	1	yes	1	0.8173899	30.186	105.68
1000	1	no	2	0.8038153	28.988	88.75
1000	1	yes	2	0.8229522	30.386	87.41
1000	1	no	5	0.8115458	29.204	60.63
1000	1	yes	5	0.8285797	30.566	60.00

Table 3.4.: Encoding results for the “cs_italy” sequence.

ter-Strike as its 3D scene (it was kindly converted to the Wavefront `.obj` format by Christopher Schwartz). Due to limitations of the raytracer used, the included diffuse light maps were not used—resulting in lower visual quality, but not making a substantial difference for video coding, since lighting is still completely diffuse and the lightmaps only contribute quite low-frequency information. The scene has some amount of variation in geometric scale—including houses, a marketplace, with the merchandise of individual stands represented as 3D geometry—but less so than the “fairy” scene does. The camera path was obtained from an interactive session by taking a 30-second long segment from the middle.

Results for this scene are shown in table 3.4. Here, warping significantly improves quality in all cases, both as measured by PSNR (with an increase exceeding 1 dB in all cases) and the structural similarity index. In particular, the results obtained using warping without subpixel refinement are significantly better than those obtained without warping and level 5 subpixel refinement, even though the former runs faster by a factor of 1.68 on average (geometric mean over all the tests, including some only shown in appendix B).

A similar comparison is done for the interactive sequences in table 3.5. It shows the SSIM index and speed obtained using warping without subpixel refinement; this is compared with the level of subpixel refinement that achieves the closest match in SSIM when warping-based motion estimation is disabled. “Speedup” is the ratio between the two encoding frame rates. As can be clearly seen, warping provides a notable speedup for the interactive scenes in all but one case.[6] The bitrate used was appended to the sequence names.

	Without warping			Warping (SubME=1)		
Sequence	SubME	SSIM-Y	Frames/s	SSIM-Y	Frames/s	Speedup
Terrain–300	1	0.7949808	133.34	0.7918640	131.01	0.9825
Terrain–500	2	0.8638736	112.31	0.8629337	123.35	1.0983
Int. Fairy–300	2	0.8983479	127.15	0.8976926	141.18	1.1103
Int. Fairy–500	2	0.9352065	115.65	0.9349554	130.61	1.1294
CS_Italy–300	6	0.6541142	67.84	0.6679018	129.44	1.9080
CS_Italy–500	6	0.7191762	58.36	0.7314405	119.74	2.0517

Table 3.5.: Time spent to reach a given SSIM index with and without warping.

3.1.3. Conclusions

The results reported in this section show that the modified `x264` encoder using warping-based motion estimation is slightly worse than conventional motion estimation for scenes with a relatively low amount of motion, but provides significantly increased quality for sequences interactively rendered from user input, which is the the typical use case in a client-server setting using server-side rendering. This increased quality is attractive to the user, but the main advantage in practice is probably that the same level of quality can be delivered to the user at lower bit rates (and thus, saving on network bandwidth costs). This notable increase in quality or, equivalently, compression performance is achieved at very low cost in terms of CPU time, and can be accomplished using comperatively simple modifications to the video encoder.

3.2. Accelerated rendering using warping and related methods

However, the problem of rendering video streams for several clients on a server remains: one would like for one server to be able to process as many clients as possible at the same time, to minimize the number of servers necessary—an important cost factor for a server-side rendering system.

[6] The increase in PSNR due to warping is way more pronounced than the increase in SSIM; a comparison based on PSNR values yields far higher speedup values, but since SSIM has higher correlation with perceived image quality, it was used instead.

An obvious approach is to use warping on the server side, and only call the renderer to fill the “missing pixels”. The idea is not new, and several such approaches were reviewed in section 1.3. Such a system can only help if three conditions are met: first, the renderer must be able to efficiently return results for individual pixels. This is the case for raytracers, but not for rasterizers. Second, the scenes have to be suitable for warping—in particular, this means they have to be static and only contain diffuse objects. Third, the combined cost of warping and rendering the missing pixels must be lower than the cost of rendering a complete image (in the average case at least), or such a system will end up *costing* time instead of saving it. For the following, it is assumed that these criteria are met.

Warping also introduces some new problems of its own: as mentioned earlier, in an interactive system, it is quite common for the user to rotate the camera rapidly to orient himself. Imagine a user “looking to the left”: the camera will first turn leftwards, so that little of the original frame will be visible after a few frames: most of the image has to be generated by the renderer. When the camera returns to its original heading, the same thing happens in reverse, even though the final camera position is the exact same as just a few frames earlier, before the user “turned his head”. This situation can be improved by using multiple reference frames or by rendering reference images with a higher field of view than the actual viewer camera uses [Mar99]. Both of those methods result in increased costs, however: the first one performs several warps for each frame, multiplying costs in the process, while the second one results in more time spent doing both rendering and warping, since the reference images are bigger. Even worse is that, when using warping to generate each new frame from the previous one, large parts of the image end up being repeatedly warped and reconstructed, accumulating resampling errors on the way, which causes very blurry results over time. Finally, high-performance hardware rasterizers are readily available, and hardware raytracers are at least an active research area; warping, on the other hand, has to be implemented as software. While it is possible to accelerate warping using a GPU (for example by evaluating the warping equation in a vertex shader and performing bilinear patch reconstruction by rendering quadriliterals), doing so does not use the hardware very efficiently and can end up being more expensive than rendering the original scene.

A more robust approach, the Render Cache, is described by Walter, Drettakis, and Parker in [WDP99]. The main idea is to record the ray hit points, including color, as a point cloud. This point cloud is then rendered using z-Buffering and a relatively simple 2D reconstruction filter. This involves somewhat more work than warping does, because each point has to be reprojected independently whereas the warping equation can be computed incrementally, but has the big advantage that there is no gradual quality degradation when samples get reused over the course of multiple frames. Also, the size of the point cloud is independent of the image resolution—using a bigger cloud results in a larger amount of work per frame, but also allows more than one full-resolution depth image’s worth of data to be kept for several frames. This helps in the “looking to the left” case and similar scenarios where the user performs rapid camera movements for a short while.

In addition to position and color data, an "age" value is kept for all points, and incremented with each frame. During rendering, the age values are written to a separate image. New samples are requested for areas for which there are only relatively old points, or no points at all; the newly rendered points replace the oldest ones in the point cloud. If a new point has a very different color from all older points in the neighborhood, the older points are aged even further; this ensures that areas showing reflections, refractions and specular highlights get updated regularly.

In [WDG02], Walter, Drettakis and Greenberg suggest several modifications to the basic Render Cache algorithm, including predictive sampling (predicting future camera motions by extrapolation and requesting samples ahead of time), a tiled z-buffer to improve cache efficiency, an implementation using SIMD opcodes, and a new prefilter with a large kernel in the reconstruction stage that is used to close bigger gaps. They also provide the source code of their render cache implementation under a GPL license.

This code was integrated with the raytracer used to render the "fairy" and "cs_italy" test sequences in the previous section. Using the render cache, the raytracer, which originally took about 1.7 seconds to render a single frame on the test machine, becomes suitable for interactive use, and the quality (given the resulting frame rate) is fully acceptable. However, especially after fast motions, there are gaps in the rendered image for a while as long as there are not enough samples available for the newly visible regions. Since the resulting image in these regions is unrelated to what is actually there, there is no point in spending many bits on such regions when transmitting a video stream; they are likely to be replaced soon anyway. This idea will be explored in the next section.

3.2.1. Per-block confidence information

During rendering, the render cache keeps track of where samples fall—it is required for the reconstruction filter. This also makes it possible to determine, for any given 2D region on the screen, how many points fell inside it, which is a measure of the "reliability" of the color values in that region. Areas with a high density of sample points are more likely to give an accurate image of the scene.

To make use of this information, for every 4×4 pixel block in the destination image the number of pixels having at least one source point projecting onto them is recorded. This information can then be passed to the video encoder to influence rate control: fewer bits should be allocated to areas with low sample density and hence lower confidence.

For the implementation, the `dvid` format was modified to support this additional per-block confidence information, and the `x264` encoder was further modified to make sure this information is available to the actual encoding functions—in particular, the `x264_image_t` and `x264_frame_t` structures and related functions were updated accordingly. The only other function changed was `x264_slicetype_mb_cost`, which estimates the magnitude of residuals in a given macroblock and is used by the rate control code to set quantization factors: the encoder always spends bits on areas with high error first, since that's where improved quality is most visible. By lowering the reported cost, areas can be "masked out" from consideration for a while. The modified code works

as follows: first, the number n of samples covering the (16×16 pixel) macroblock is computed from the counts for the 4×4 pixel blocks. The cost of the macroblock is then multiplied by $\frac{1}{2}(1 + n/256)$, so that the reported distortion for a block with no recorded samples in it is half of what it would be if the block was fully populated; the weighting formula was found by experimentation. This simple linear approach yielded the overall best results, although they were still not good. Quality was measured using a weighted PSNR, where the squared differences are multiplied by the importance rating for the block they fall into; this was done so that (potentially) bigger errors in the regions with low confidence rating did not influence the overall results too much.

Sequence	PBConf	wPSNR (dB)	Frames/s
Int. Fairy–300	no	35.719	151.35
Int. Fairy–300	yes	35.676	151.35
Int. Fairy–500	no	38.019	141.69
Int. Fairy–500	yes	38.029	141.69
CS_Italy–300	no	29.347	138.84
CS_Italy–300	yes	29.309	139.16
CS_Italy–500	no	31.004	128.77
CS_Italy–500	yes	30.996	128.90

Table 3.6.: Results when using per-block confidence information on the “interactive fairy” and “cs_italy” sequences.

Results using variants of the “interactive fairy” and “cs_italy” sequences that were written in realtime using the render cache are shown in table 3.6 (the terrain sequence was not rendered using a raytracer, and hence cannot use the render cache). “PBConf” indicates whether the per-block confidence rating was used to influence rate control, and “wPSNR” is the weighted PSNR. As can be seen, using the confidence information does not result in an appreciable change in runtime and results in *worse* quality, even as measured by the weighted PSNR metric.

Another attempt used the confidence information to force `x264` to skip encoding blocks where overall confidence was low; this, while increasing encoding speed slightly, resulted in a precipitous in objective quality, with a loss of on average 0.7 dB PSNR when very low-confidence blocks were skipped (with less than 16 out of 256 pixels being obtained from actual samples); hence, this strategy was abandoned.

Overall, it seems like there is not much to be gained by using the confidence rating during the encoding process; the reported results were the best I was able to achieve during about a week of experimentation. Very low-confidence blocks are relatively rare in any case; the render cache does a quite good job of filling holes within a few frames. Since the framebuffer for holes is not cleared, but just left as it was in the previous frame, the codec is able to efficiently encode such blocks without further assistance.

3.2.2. Relevance to server-side rendering

While the render cache works well as a "drop in" library to make relatively slow ray-tracers suitable for interactive use, it cannot easily make use of rendering work done for other clients—while it can easily fill holes using information from another point cloud corresponding to a different viewer, doing so increases the amount of work the render cache has to do considerably, analogous to multiple reference images for warping. As a result, such a system quickly reaches a point of diminishing returns: processing the additional points costs more time than would be spent on rendering them. Warping, the render cache or similar methods may be useful when an individual renderer is otherwise too slow to deliver frames at interactive rates, but there are no substantial gains to be expected from sharing the rendered pixels between several clients; the overhead is just too large.

This is, mainly, because such techniques work at the very low level of pixels and rays, which seems to be the wrong scale to look for coherencies and correspondencies between multiple viewers; a better strategy to exploit such coherencies is probably by working on a higher level while leaving the actual rendering infrastructure untouched.

Conclusions and future work

Chapter 2 considered warping for video coding, but section 2.3.1 showed quite conclusively that warping is not inherently superior from a prediction standpoint, not even with perfect reference data; quite the opposite, the motion compensation-based method had overall better prediction performance. Hence warping is probably not attractive for medium to high bitrate applications; however, the results in section 2.3.2 seem to indicate that warping might prove to be advantageous when low bitrates are targeted, since the relative cost of encoding disparity data is significantly lower than the cost of motion vectors. Because of the relatively high computational cost of warping (compared with conventional video coders) and the quite low expected gains, this line of inquiry was not continued in this thesis; however, for applications where the inherent assumptions of warping hold and the cost of data transmission is very high, development of a full warping-based video codec could pay off.

The warping-based motion estimation described in section 3.1 is quite simple to implement and showed significant quality gains on video sequences recorded from interactive usage, at a very small cost in performance—or, equivalently, reached a given target quality much faster than when using normal motion estimation. There is little reason not to try it when producing data for a video stream from a 3D renderer; the results suggest that submitting "hint" motion vectors to a video encoder could be a promising approach in general, if the application has such motion information available.

In contrast, the techniques described in 3.2 do not fare so well; using warping to accelerate rendering has too many problems and limitations to be a real option in practice, and while the render cache is overall more robust, it still works exclusively with raytracers and has no obvious way to benefit from coherence between multiple viewers without sacrificing additional performance per viewer. Overall, it just seems more practical to use existing optimized rendering hardware for a server-side rendering system.

Doing so efficiently is probably the most fundamental problem to be solved for server-side rendering. As in any client-server system, the server side is trivial to parallelize and can hide latencies when rendering the view for one client by processing other clients in the meantime; however, having to render a video stream for each client at a consistent framerate means that the renderers have to meet soft realtime requirements. Also, rendering even a single view can require a relatively large amount of geometry and texture data; clients should be assigned to servers in a way that maximizes reuse of such data between the several clients that are processed on a single server, or otherwise the amount of available (video) memory could limit the maximum number of clients more seriously than the amount of available CPU time does. All this makes the problems of work dispatch and load balancing a lot more difficult than they would be for e.g. normal

web servers; an efficient solution would be a large step towards practical server-side rendering.

Finally, a big problem that *wasn't* considered in this thesis is network latency: even with broadband connections, round-trip times between a client and the render server can be expected to be above 30 ms when communicating over the internet. Considering rendering and encoding time, it is likely that there will be a delay of over a tenth of a second between user input that causes the camera to move and the corresponding visual feedback. The reason for avoiding the topic is that little can be done about latency on the software level, except trying to avoid introducing unnecessary additional sources of delay. Still, the high latency has a notable impact on the user experience, and poses a significant problem in terms of user interface design.

Acknowledgements

I thank my advisors Prof. Dr. Reinhard Klein and Ruwen Schnabel for their support and several suggestions after reading various earlier drafts of this text; Dierk Ohlerich, the primary author of the "Altona" framework used to write several of the programs and tools created during the preparation of this thesis (cf. Appendix A); Nina Høegh-Larsen, who proofread this thesis in several stages of completion and uncovered several orthographic and stilistic mistakes; C. Gerald Knizia, who made many helpful typographic suggestions; Gero Müller, who wrote the raytracer (GRTLib) used in section 3.2; and Christopher Schwartz, who provided the converted "cs_italy" test scene.

A. Source code

The provided source code is not for one single program; rather, there are different tools and libraries, each of which implements at least one of the techniques mentioned. The different directories on the included DVD contain (in alphabetical order):

altona is an application framework by Dierk Ohlerich and others that was used to develop most of the supplied programs.

depthvid/compress is a library that implements a very basic wavelet- and warping-based video codec that is briefly described in section 2.3.

depthvid/depthvid is the basic warping library; it implements the image classes, the warping algorithm described in section 2.2 in `warping.cpp`, warping with bilinear patch reconstruction (for comparision) in `altwarping.cpp`.

depthvid/dvid2avi converts `.dvid` files to `.avi` files that can be played with nearly all media players (for previewing and as input to normal video codecs).

depthvid/encode_eval is a Python script used to obtain the video encoding results reported in section 3.1.2 and appendix B.

depthvid/ext contains the IJG JPEG library.

depthvid/model_eval is a small tool used to perform the quality evaluation of various prediction models; details are in section 2.3. It uses hardcoded filenames, since it was only ever used for one purpose.

depthvid/raytrace contains GRTLib (a raytracer), the RenderCache libraries, and the implementation of an interactive viewer using the raytracer to render and the warping/render cache techniques to display at interactive rates even though the original renderer cannot. It is described in section 3.2.

depthvid/warpapp provides a frontend to test the warping algorithms described in chapter 2. It has a graphical user interface and should be self-explanatory.

depthvid/x264_mod is the modified `x264` encoder that supports warping-based motion estimation as described in chapter 3 and per-block confidence rating information as described in section 3.2.1.

depthvid/yuvview is a viewer for raw `.yuv` files as used by the Dirac and `x264` encoders. It has a user interface similar to “warpapp”.

pdf contains this document in `.pdf` format.

srcdata contains the various test sequences as `.dvid` files.

A.1. Build process

All code (except the `encode_eval` script) was written in C++ using the Microsoft Visual C++ 2005 compiler. The directory `raytrace` includes project and solution files that can be used directly from the IDE; the other (Altona-based) projects use a special system that generates them from a textual description (the files with `.mp.txt` extension). The `altona` and `depthvid` directories should be copied to a folder on a writeable disk, e.g. "`c:\code`"; then, the `makeproject` utility (included in the `altona/bin` directory) should be run with the command line `makeproject -w -r c:\code`; this generates project and solution files for everything.

Various libraries are used, but nothing exotic: the Altona programs require the DirectX 9 SDK, `raytrace` uses Qt 4.2, and `x264_mod` requires pthreads_win32 for multi-threading support.

B. Complete results for warping-based motion estimation

The results given in section 3.1.2 for the "fairy", "interactive fairy" and "cs_italy" sequences are summarized to reduce clutter; in particular, the results obtained with multiple reference frames have been deleted, since they do not add any significant information to the discussion. Nevertheless, the full results of the experiments are reported on the following pages for the sake of completeness.

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.8544471	31.999	146.94
300	1	yes	1	0.8543651	31.967	147.32
300	1	no	2	0.8593798	32.159	126.32
300	1	yes	2	0.8589095	32.119	127.15
300	1	no	5	0.8695813	32.459	89.44
300	1	yes	5	0.8686421	32.387	90.57
300	3	no	1	0.8611278	32.396	138.46
300	3	yes	1	0.8615861	32.377	133.02
300	3	no	2	0.8670841	32.616	116.84
300	3	yes	2	0.8671025	32.596	114.97
300	3	no	5	0.8750907	32.871	77.94
300	3	yes	5	0.8743009	32.826	78.36
500	1	no	1	0.8924078	33.747	136.82
500	1	yes	1	0.8927671	33.723	135.22
500	1	no	2	0.8974561	33.973	114.52
500	1	yes	2	0.8974329	33.935	114.97
500	1	no	5	0.9040100	34.256	79.45
500	1	yes	5	0.9029872	34.184	80.11
500	3	no	1	0.8992227	34.236	127.15
500	3	yes	1	0.9003598	34.255	123.34
500	3	no	2	0.9054001	34.541	105.50
500	3	yes	2	0.9055953	34.532	102.86
500	3	no	5	0.9104389	34.798	69.56
500	3	yes	5	0.9101451	34.765	69.32
1000	1	no	1	0.9377240	36.744	120.76
1000	1	yes	1	0.9378214	36.714	120.50
1000	1	no	2	0.9419348	37.056	98.47
1000	1	yes	2	0.9414331	36.991	97.96
1000	1	no	5	0.9457670	37.369	66.44
1000	1	yes	5	0.9449789	37.291	66.36
1000	3	no	1	0.9438986	37.383	114.52
1000	3	yes	1	0.9448177	37.429	110.13
1000	3	no	2	0.9481957	37.752	92.75
1000	3	yes	2	0.9482873	37.751	89.71
1000	3	no	5	0.9515541	38.071	59.69
1000	3	yes	5	0.9511365	38.032	59.14

Table B.1.: Full encoding results for the “fairy” sequence.

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.8940561	34.118	144.35
300	1	yes	1	0.8976926	34.656	141.18
300	1	no	2	0.8983479	34.358	127.15
300	1	yes	2	0.9011825	34.860	123.34
300	1	no	5	0.9061646	34.734	90.42
300	1	yes	5	0.9133149	35.303	89.16
300	3	no	1	0.8959362	34.219	138.46
300	3	yes	1	0.8997786	34.763	130.32
300	3	no	2	0.9000404	34.476	118.76
300	3	yes	2	0.9036171	34.993	113.15
300	3	no	5	0.9072016	34.821	80.22
300	3	yes	5	0.9146864	35.443	78.04
500	1	no	1	0.9310123	36.642	133.95
500	1	yes	1	0.9349554	37.420	130.61
500	1	no	2	0.9352065	36.969	115.65
500	1	yes	2	0.9385026	37.715	112.50
500	1	no	5	0.9401657	37.339	80.00
500	1	yes	5	0.9448482	38.109	79.23
500	3	no	1	0.9324642	36.768	128.85
500	3	yes	1	0.9371525	37.607	121.26
500	3	no	2	0.9365830	37.095	108.26
500	3	yes	2	0.9407811	37.923	102.86
500	3	no	5	0.9413833	37.486	70.85
500	3	yes	5	0.9458558	38.242	70.07
1000	1	no	1	0.9685249	40.839	119.25
1000	1	yes	1	0.9703983	41.906	115.21
1000	1	no	2	0.9713022	41.251	101.78
1000	1	yes	2	0.9723293	42.198	96.32
1000	1	no	5	0.9731321	41.572	68.17
1000	1	yes	5	0.9744583	42.550	67.68
1000	3	no	1	0.9694822	41.006	114.29
1000	3	yes	1	0.9714564	42.091	106.08
1000	3	no	2	0.9719193	41.372	95.06
1000	3	yes	2	0.9735883	42.437	89.02
1000	3	no	5	0.9739072	41.732	61.54
1000	3	yes	5	0.9754957	42.766	60.50

Table B.2.: Full encoding results for the “interactive fairy” sequence.

Settings				Results		
Bitrate (kbit/s)	Ref	WarpME	SubME level	SSIM-Y	PSNR-Y (dB)	Frames/s
300	1	no	1	0.6353905	25.557	131.20
300	1	yes	1	0.6679018	26.709	129.44
300	1	no	2	0.6397960	25.668	113.39
300	1	yes	2	0.6722945	26.828	112.71
300	1	no	5	0.6493255	25.838	81.71
300	1	yes	5	0.6833704	27.024	81.24
300	3	no	1	0.6396398	25.635	122.03
300	3	yes	1	0.6724197	26.802	117.54
300	3	no	2	0.6435966	25.756	104.35
300	3	yes	2	0.6773597	26.952	102.67
300	3	no	5	0.6535233	25.931	71.29
300	3	yes	5	0.6868387	27.122	69.57
500	1	no	1	0.7036136	26.687	120.24
500	1	yes	1	0.7314405	27.981	119.74
500	1	no	2	0.7087490	26.836	102.31
500	1	yes	2	0.7372080	28.129	100.52
500	1	no	5	0.7161530	26.995	72.09
500	1	yes	5	0.7444381	28.296	72.18
500	3	no	1	0.7068231	26.767	113.39
500	3	yes	1	0.7359362	28.088	108.26
500	3	no	2	0.7124135	26.936	93.82
500	3	yes	2	0.7418941	28.257	91.72
500	3	no	5	0.7209729	27.109	63.78
500	3	yes	5	0.7487422	28.426	62.74
1000	1	no	1	0.7982175	28.812	106.67
1000	1	yes	1	0.8173899	30.186	105.68
1000	1	no	2	0.8038153	28.988	88.75
1000	1	yes	2	0.8229522	30.386	87.41
1000	1	no	5	0.8115458	29.204	60.63
1000	1	yes	5	0.8285797	30.566	60.00
1000	3	no	1	0.8016265	28.911	102.31
1000	3	yes	1	0.8217056	30.326	96.48
1000	3	no	2	0.8087717	29.133	83.36
1000	3	yes	2	0.8272907	30.541	79.89
1000	3	no	5	0.8145216	29.304	54.60
1000	3	yes	5	0.8327855	30.710	53.63

Table B.3.: Full encoding results for the “cs_italy” sequence.

Bibliography

[AB91] E. H. Adelson and J. R. Bergen. The Plenoptic Function and the Elements of Early Vision. In M. Landy and J. A. Movshon, editors, *Computational Models of Visual Processing*. MIT Press, 1991.

[AM+07] L. Aimar, L. Merrit, et al. x264 - a free h264/avc encoder. Available at http://www.videolan.org/developers/x264.html (accessed May 11, 2008), November 2007. SVN Version 680.

[ARPC06] D.G. Aliaga, P. Rosen, V. Popescu, and I. Carlbom. Image warping for compressing and spatially organizing a dense collection of images. *Signal Processing: Image Communication*, 21(9):755–769, October 2006.

[BBC] BBC. Dirac encoder. Available at http://sourceforge.net/projects/dirac/ (accessed May 2, 2008).

[BBC08] BBC. *Dirac Specification, Version 2.2.0*, April 2008. Available at http://dirac.sourceforge.net/DiracSpec2.2.0.pdf (accessed Apr. 23, 2008).

[CG02] C. Chang and S. Ger. Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering. In *PCM '02: Proceedings of the Third IEEE Pacific Rim Conference on Multimedia*, pages 1105–1111, London, UK, 2002. Springer-Verlag.

[DL03] J. Duan and J. Li. Compression of the layered depth image. *IEEE Transactions on Image Processing*, 12(3):365–372, March 2003.

[HM99] T.C. Hudson and W.R. Mark. Multiple Image Warping for Remote Display of Rendered Images. Technical Report TR99-024, University of North Carolina at Chapel Hill, 1999.

[ISO93] ISO/IEC. *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 2: Video*, 1993. Reference number ISO/IEC 11172-3:1993 (referred to as "MPEG-1").

[ISO01] ISO/IEC. *Information technology – Coding of audio-visual objects – Part 2: Visual*, 2001. Reference number ISO/IEC 14496-2:2001(E) (referred to as "MPEG-4 part 2").

[ITU05] ITU-T. *Recommendation H.264: Advanced video coding for generic audiovisual services*, March 2005. Also an ISO standard as ISO/IEC 14496-10:2005 (referred to as "H.264").

[LK81] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI81 Vol. 2*, pages 674–679, 1981.

[Mar98] R.W. Marcato, Jr. Optimizing an Inverse Warper. Master's thesis, Massachusetts Institute of Technology, May 1998.

[Mar99] W.R. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping.* PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1999.

[Mar00] I.M. Martin. Adaptive Rendering of 3D Models over Networks Using Multiple Modalities. Technical Report RC21722, IBM T.J. Watson Research Center, April 2000.

[McM97] L. McMillan, Jr. *An image-based approach to three-dimensional computer graphics.* PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1997.

[MS03] H. S. Malvar and G. J. Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. Technical Report JVT-I014r3, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, July 2003.

[PA06] V. Popescu and D. Aliaga. The depth discontinuity occlusion camera. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 139–143, New York, NY, USA, 2006. ACM.

[SGHS98] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, New York, NY, USA, 1998. ACM.

[TPB05] G. Thomas, G. Point, and K. Bouatouch. A Client-Server Approach to Image-Based Rendering on Mobile Terminals. Technical Report RR-5447, INRIA, INRIA Futurs, Bordeaux, January 2005.

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[WDG02] B. Walter, G. Drettakis, and D. P. Greenberg. Enhancing and optimizing the render cache. In *Rendering Techniques 2002: Proceedings of the 13th Eurographics workshop on Rendering*, pages 37–42. Springer-Verlag, 2002.

[WDP99] B. Walter, G. Drettakis, and S. Parker. Interactive Rendering using the Render Cache. In *Rendering Techniques '99: Proceedings of the 10th Eurographics workshop on Rendering*. Springer-Verlag, 1999.

[WMD+04] R. Wahl, M. Massing, P. Degener, M. Guthe, and R. Klein. Scalable compression and rendering of textured terrain data. *Journal of WSCG*, 12(3):521–528, February 2004.

[WSS00] M. Weinberger, G. Seroussi, and G. Sapiro. The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS. *IEEE Transactions on Image processing*, 9(8):1309–1324, August 2000.

[XLLZ01] G. Xing, J. Li, S. Li, and Y. Zhang. Arbitrarily shaped video object coding by wavelet. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(10):1135–1139, October 2001.

[YN00] I. Yoon and U. Neumann. Web-Based Remote Rendering with IBRAC (Image-Based Rendering Acceleration and Compression). *Computer Graphics Forum*, 19(3):321–330, 2000.

[ZC04] C. Zhang and T. Chen. A Survey on Image-Based Rendering - Representation, Sampling and Compression. *EURASIP Signal Processing: Image Communication*, 19(1):1–28, 2004.

Index

B-frame, 15, 31
bitrate, 14, 15, 17, 43, 44, 55

camera
  model, 20
center-of-projection, 20
coordinate system
  camera, 20
  image, 20
  world, 20

discrete cosine transform (DCT), 16
depth discontinuity, 23, 28
Dirac, 15–17, 31, 35–37
disparity
  cost of storage, 35–37
  definition, 21
  filtering of, 26, 28
`dvid` format, 41, 51, 57, 58

entropy coding, 17, 36
exposure error, 10, 12, 22

GPU, 50
group of pictures (GOP), 16

H.264, 15–17, 40–43
hole-filling, 26

I-frame, 15
IDCT, *see* discrete cosine transform
image plane, 20
image-based rendering, 9
inter coding, 15, 35
intra coding, 15, 35
inverse warping, 11, 24
invisible occluder error, 11–13, 23

JPEG-LS, 14

layered depth image, 10, 12, 14, 22
low bitrate applications, 35

macroblock, 16, 25, 40, 51
motion compensation, 16, 19, 35–37, 55
  overlapped block, 16, 31
motion estimation, 16, 40–41
  subpixel refinement, 42–44
  warping based, 55, 59
  warping-based, 40–49
    efficiency, 43–49
motion vector, 16, 17
MPEG, 11, 14–17, 25

network bandwidth, 9, 12, 13, 39, 49

occlusion error, 10, 12, 22
optical flow, 16

P-frame, 15
partition, 16, 40
plenoptic function, 9
PSNR, 11, 15, 31, 32, 43

quantization, 17, 36, 42, 51

rate control, 17, 42, 51
reconstruction, 23–29
  bilinear patch, 23, 29, 32, 34
  gaussian cloud, 23
reference frame, 19, 29, 40, 42, 43
render cache, 50, 55
residual image, 16

SCARPED, 32, 41

server-side rendering, 7, 39, 49, 55
structural similarity index (SSIM), 43

test sequence
  cs_italy, 47, 52, 62
  fairy, 32, 44, 60
  interactive fairy, 46, 52, 61
  terrain, 31, 43
two-pass encoding, 17, 43

video codec, 15, 19, 39
video object wavelet codec, 14

warping
  as predictor, 19
  equation, 21, 28, 40, 42, 44
    Jacobian of, 23, 28
  matrix, 22, 28, 42
  motion estimation, *see* motion estimation, warping-based
  of planar depth images, 9–10, 20–29
  prediction efficiency, 29–35
  shortcomings, 22
  with sample shifting, 28, 32
warping equation, 50
wavelet, 14, 16, 36

x264, 41–43
`x264_frame_t`, 41, 51
`x264_image_t`, 41, 51

YCbCr color space, 14, 31, 41
YCoCg color space, 35