GPU-ASSISTED GEOMETRY
PROCESSING FOR NOVEL VIEW
SYNTHESIS FROM DEPTH VIDEO

ALEXANDER BOGOMJAKOV
GPU-ASSISTED GEOMETRY PROCESSING
FOR NOVEL VIEW SYNTHESIS FROM
DEPTH VIDEO

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

ALEXANDER BOGOMJAKOV

SUBMITTED TO THE SENATE OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY
TEBETH, 5768
HAIFA
JANUARY, 2008
ACKNOWLEDGEMENT

I wish to express gratitude to my supervisor Prof. Craig Gotsman for his guidance, support and help.

THE GENEROUS FINANCIAL HELP OF TECHNION IS GRATEFULLY ACKNOWLEDGED
## Contents

Abstract 1

1 Introduction 3

1.1 This Work ......................................................... 10

2 Background 13

2.1 Graphics Pipeline ................................................. 13
2.2 Multi-Pass Rendering ............................................. 16
2.3 Projective Texture Mapping ................................. 17
2.4 Projective Shadow Mapping ................................. 18
2.5 Depth Peeling ................................................... 20

3 GPU-Assisted Z-Field Simplification 22

3.1 Introduction ...................................................... 23
3.2 Related Work .................................................... 24
3.3 Z-Field Simplification ........................................ 26
3.4 Modified Algorithm ............................................ 28
3.5 Simplification in Graphics Hardware .................. 30

3.5.1 Frame Arrangement ........................................ 30
CONTENTS

3.5.2 Rendering Passes .............................. 31
3.5.3 Meshing ...................................... 36
3.6 Results ......................................... 36
3.7 Discussion ...................................... 38

4 Free-Viewpoint Video from Depth Cameras 40
  4.1 Introduction .................................. 41
  4.2 Related Work .................................. 44
  4.3 Contribution ................................... 46
  4.4 The Visual Hull ................................. 47
    4.4.1 Definitions ................................ 47
    4.4.2 Rendering The Visual Hull ................. 48
    4.4.3 Improved Projective Rendering ............... 51
  4.5 The Depth Hull ................................ 53
  4.6 Rendering Depth Hulls ......................... 54
  4.7 Experimental Results ......................... 56
    4.7.1 Camera Setup ................................ 56
    4.7.2 General Performance ....................... 58
  4.8 Conclusion and Discussion ..................... 59

5 Reduced Depth and Visual Hulls of Complex Scenes 63
  5.1 Introduction .................................. 64
  5.2 Related Work .................................. 67
  5.3 Definitions ................................... 68
  5.4 RDH rendering algorithm ....................... 71
  5.5 The VH case ................................... 73
## CONTENTS

5.6 Depth and correspondence ........................................... 76  
5.7 Occlusions ............................................................... 76  
5.8 Experimental Results .................................................. 78  
5.9 Conclusions ............................................................. 80  

References .............................................................................. 86  

Hebrew Abstract ..................................................................... i
# List of Figures

2.1 Graphics pipeline ................................................. 14
2.2 Multi-pass rendering ........................................... 17
2.3 Projective texture mapping ................................. 18
2.4 Projective shadow mapping ............................... 19
2.5 Depth peeling .................................................. 20
3.1 Z-field examples ............................................... 23
3.2 Z-field triangulation ........................................ 26
3.3 Vertex removal ................................................ 26
3.4 Parent-child relationships ................................. 28
3.5 Vertex levels ................................................... 29
3.6 Frame arrangement ........................................... 32
3.7 Simplification rendering passes 1–4 .................... 34
3.8 Simplification rendering pass 5 .......................... 35
3.9 Height field simplification example ................... 39
3.10 Simplified height fields example ..................... 39
4.1 Novel view rendering example .......................... 42
4.2 Hull comparison in 2D ........................................ 48
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Rendering quality comparison</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Camera setup</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison of DH renderings</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>A hand sequence</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>A head sequence</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of DH rendering</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Depth map definition</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>Umbra of a view</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>A depth hull and its second layer</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>Visual hull ambiguity</td>
<td>74</td>
</tr>
<tr>
<td>5.6</td>
<td>RVH rendering using correspondence</td>
<td>75</td>
</tr>
<tr>
<td>5.7</td>
<td>Computing correspondences from a depth view</td>
<td>78</td>
</tr>
<tr>
<td>5.8</td>
<td>Occlusions</td>
<td>79</td>
</tr>
<tr>
<td>5.9</td>
<td>Hand RDH example</td>
<td>82</td>
</tr>
<tr>
<td>5.10</td>
<td>Two men example with different hulls</td>
<td>83</td>
</tr>
<tr>
<td>5.11</td>
<td>Paper prisms in a circle arrangement</td>
<td>84</td>
</tr>
<tr>
<td>5.12</td>
<td>Paper prisms in a row arrangement</td>
<td>85</td>
</tr>
</tbody>
</table>
Abstract

Depth cameras, which provide color and depth information per pixel at video rates, offer exciting new opportunities in computer graphics. We address the challenge of supporting free-viewpoint video of dynamic 3D scenes using live data captured and streamed from widely-spaced viewpoints by a handful of synchronized depth cameras. We introduce the concept of the depth hull, which is a generalization of the well-known visual hull. The depth hull reflects all the dense depth information as observed from several centers of projection around the scene. It is the best approximation of the scene geometry that can be obtained from a given set of depth camera recordings.

Contemporary graphics hardware based on a Graphics Processing Unit (GPU) has developed to successfully compete with the CPU for many specialized tasks. Highly parallelized streaming architecture and a large number of concurrent processing units allows the graphics hardware to carry out calculations far beyond those needed for mere polygonal rendering. We take advantage of this computation power to perform real-time processing for different stages of free-viewpoint video.

We first present a GPU-based adaptation of a simplification algorithm used for preprocessing of the depth data. This non-rendering computation maps well to the graphics hardware because the data is organized on a regular grid. We then present a general improvement to the best existing visual hull rendering algorithm, which is
of independent interest. We use this to contribute a hardware-accelerated method for rendering novel views from depth hulls in real-time. This method is based on a combination of techniques from projective shadow mapping and constructive solid geometry (CSG). Our rendering method achieves high-quality results even when only a modest number of depth cameras are deployed. They are applicable to any set of images with accompanying dense depth maps that correspond to arbitrary viewing positions around the scene. We provide experimental results using a system incorporating two depth cameras recording a dynamic scene. We also provide an adaptation of the depth hull and the visual hull for rendering the geometry of complex scenes, consisting of multiple objects.
Chapter 1

Introduction

Novel view synthesis is generation of new images that present a scene from angles different from the ones that were used to capture it. The input to the problem is given as one or more reference images, each taken from a different viewpoint. The output is the image corresponding to the camera position and direction that are arbitrarily chosen by the user. In an interactive application this effectively creates a user-controlled virtual camera. There are a number of existing as well as emerging applications of novel view synthesis. It can be used in conjunction with 3D display technologies such as stereo imaging.

One recently appeared application is 3D photography. Here, a real-world scene is captured and represented in such a way, that allows later examination from different viewpoints. Similarly to the 2D photography, the captured frame is static. However, it can be interactively viewed by changing the angle and the position of the viewer. Since a frame is captured only once and then viewed many times, it is possible to spend as much time as needed to construct the best possible representation of the scene.
CHAPTER 1. INTRODUCTION

Novel view generation is the key operation of free-viewpoint video. In usual video the user’s viewpoint is fixed by the viewpoint of the camera that is capturing the scene. Sometimes many cameras are used and the user’s picture is switched between the different cameras. Free-viewpoint video is an application where the constraints on the user viewpoint are removed. Just as each frame of the conventional 2D video is actually a photograph, each frame of the free-viewpoint video is a 3D photograph.

Since the free-viewpoint video is essentially a sequence of 3D photographs, the problem can be approached by spending as much time as needed to reconstruct each frame. In addition, frame to frame coherency can be exploited to speed-up the process and improve quality. There is an approach [42] that first reconstructs as much geometry as possible in each frame and then performs a global optimization step which finds a smooth 4D surface that represents the entire sequence.

Live free-viewpoint video will allow different cinematographic effects, that before were possible only through off-line processing, to be applied to live transmissions. This will greatly enrich the visual experience in such applications as television and video conferencing. The problem here, however, is more difficult, because all the stages from the acquisition through (possibly) reconstruction to display have to be performed on-the-fly. Even when these stages are pipelined, each has to be fast enough to allow video frame-rate. This is the version of the problem that we are trying to deal with here.

Image-based techniques render novel views directly from the images of the reference views. These methods often consider the problem as an image warping problem. Generally, the input images are warped and combined in order to synthesize a novel view. For each output pixel it is necessary to identify the correct sampling points of the input images. There is no dependency on the ability to recover the geometry at
each point. Using an image-based approach it is possible to reproduce the appearance of objects whose geometries are not well defined, like bushes or fog. These techniques are also more suitable for reproduction of view-dependent appearance because they do not usually rely on the assumption of Lambertian surfaces.

On the other hand, it is usually necessary to process relatively large numbers of input images. This increases the complexity of either the acquisition setup or the registration process and may require the scene to be static. The static scene requirement arises when a single, usually hand-held, video camera is used to capture the scene from different angles. Many input images can also mean large data bandwidth, extensive storage and computation power requirements. These factors make image-based methods not very suitable for live free-viewpoint video.

Geometry-based methods construct some kind of geometric approximation of the scene objects and use it for rendering. From the image warping point of view, these methods provide a geometric proxy which aids the image warping process. A unifying framework of image-based and geometry-based techniques for such image warping is given by the Unstructured Lumigraph [14].

Geometry-based methods include mesh-based, volumetric and point-based techniques. They have the advantage that the display part can be very efficient. Once some representation of the scene is constructed, it is always possible to exploit graphics hardware to do the rendering. While it is possible to implement at least some parts of image-based methods on the GPU, geometry-based approaches lend themselves more naturally on the graphics hardware. This is mostly because they use such primitives as triangles and points.

Unfortunately, with these methods it is difficult to model view-dependent appearance, so usually Lambertian surfaces are assumed. It is also generally impossible to
cover the entire geometry of a fairly complex scene. For better image quality of the result gaps and missing parts need to be filled, covered or at least masked. These methods are also dependent on the ability to measure or compute the geometry. However, if this problem is solved, geometry-based methods can provide the performance needed for free-viewpoint video, especially, live.

There is a separate class of techniques, which assume a-priori knowledge about the scene content. In this case a generic model of the scene can be constructed in advance. The captured data is used to modify the generic model – most commonly humans – to reflect the actual scene.

The advantage is that the model-based representation can have high quality appearance. In the places where the scene is known the model looks like the real objects, while the places where the real data is missing are intelligently completed with the template.

The most obvious disadvantage is that there is no generality of the scene content. Nothing else can be represented except for what is explicitly modeled by the template model. Also the fitting of the template to the real data may involve an optimization process, which could be too lengthy for live video processing.

The acquisition is usually performed by several video cameras. Depending on the approach, the number of cameras may vary from as few as 2-3 to as many as 80 or more. Management of a multi-camera system has some specific aspects like synchronization, calibration, registration, data transmission, cost, etc. Naturally, the complexity of all these factors grows with the number of cameras. Still, there are systems that successfully use large numbers of cameras [21,22,59]. Given images from the video cameras many geometry-based methods need to perform further processing in order to recover approximate geometry of the scene.
Some off-line methods use laser range scanners in addition of the regular video cameras. The range scanners provide partial geometry of the scene, while the images from the video cameras are used for texturing. An advantage of such a setup is that high quality partial geometry is directly available from the acquisition stage and there is no need to perform additional processing to get it. A disadvantage is mostly the time that it takes to perform a scan, so this approach is not yet suitable for a live setup.

In the case of range scanners as well as depth cameras, pieces of the scene geometry from different viewpoints and different cameras are available directly from the acquisition process. Depth cameras can be used to quickly obtain partial geometry of the scene, which usually even comes with corresponding texture images. This is an emerging technology and far from being mature, but many working prototypes as well as some commercial systems are available. A depth camera usually gives a depth image of the scene as seen from the camera and a color or grayscale image, often acquired by the same sensor, so the geometry and texture registration is ideal and readily available. Some of these cameras are able to provide the full speed of 30 3D frames per second [1]. Naturally, a speed-quality tradeoff exists. The fastest of these cameras are based on infrared time-of-flight [1,3], but suffer from large levels of noise and poor depth resolution. In contrast, structured light cameras [6,13] provide superior geometry quality, but work at limited frame-rates. This work is based on Vialux structured light Z-Snapper cameras.

Many geometry-based methods that use regular video cameras for acquisition, compute geometry from the images. Some use stereo [16] to compute depth images from pairs of views. Kanade et al. [29] perform multi-baseline stereo. Magnor et
al. [42] use photo-consistency methods, which are based on the early work on voxel-coloring [55]. Several other works that are based on photo-consistency [39, 60, 61] actually defer the depth computation to the display stage, but are able to compute it any time, if needed. Similarly, Li et al. [36–38] implicitly reconstruct Visual Hull geometry in real-time during rendering. Visual Hull (VH) of an object is the maximal object that has the same silhouettes. They use triangles to represent extruded silhouettes. Polyhedral Visual Hull [43] uses these triangles to construct polyhedral VH approximation of the scene, which is basically the same technique as used by some 3D scanners based on volume carving [32]. Other methods use volumetric representations and use volume carving to throw away irrelevant voxels [18].

The acquired and reconstructed data often needs to be organized in a specific manner in order to allow efficient execution of operations needed for a specific algorithm. Some image-based methods like the Lumigraph [22] and Light Field [35] store the input images in organized collections, often associated with a grid or some other parametrization of space. Some geometric methods construct mesh representations, which reflect the scene geometry as completely as possible. Matusik et al. [43] use polyhedral – i.e. 3D mesh – representation of the captured objects. Chen et al. [17] prefer integrating the extracted geometry into one large mesh representing the entire scene. Volumetric methods [18] may store the geometry data in the form of voxels.

An important observation here is that our final goal is not geometric processing but display. This means that it should be possible to use some kind of partial representation, which does not require heavy geometric processing. For example, it is usually not necessary to ensure continuity properties higher than $G^0$. Moreover, if the rendering of an object appears correct, even $G^0$ may not be required. Indeed, the object could as well be rendered as a polygon soup. However, if an algorithm needs to
take into account the normals of the object, at least a piece-wise mesh representation is useful, which, in addition, is more efficient for rendering on graphics hardware.

Some VH algorithms [36–38] are based on polygonal representation of the object silhouettes. Depth map based techniques [11, 46–48] store the data either as optimized depth meshes or as depth images, which still have to be converted into meshes for rendering.

Image-based rendering of novel views usually proceeds by casting rays into the scene. It is necessary to determine which reference images to sample and at which locations. Light-field methods [22, 35] identify rays of the reference images which are the closest to the desired rays. The colors carried by the identified reference rays are combined using weights according to some sampling scheme in order to produce the colors for the desired rays. Image-based visual hull algorithm [44] projects the desired rays onto the reference views and intersects these projections with the silhouettes of the objects. By comparing the points of the silhouette intersections the algorithm determines the point where the rays hit the visual hull. The corresponding points in the reference images are sampled to obtain a color.

Polyhedral visual hulls [43] as well as other mesh-based [17] techniques use the standard polygonal rendering in graphics hardware because they have a mesh model of the scene.

There is an intermediate class of techniques, which use the graphics hardware both for reconstruction and rendering simultaneously. These are the hardware accelerated visual hull rendering techniques [36–38] as well as most photo-consistency techniques [39, 60, 61].

The hardware-accelerated photo-consistency approach performs a sweep of the scene volume by a plane or a surface. The surface may be adaptively tesselated
CHAPTER 1. INTRODUCTION

during the process. Eventually, each relevant point of the scene volume is projectively textured with the corresponding colors from the reference images, and if the colors from the different images agree within certain tolerance, the point is rendered.

The hardware-accelerated visual hull methods use triangular representation of the extruded silhouettes. The surfaces of the extruded silhouettes need to be intersected with each other in order to produce the visual hull surface. This intersection can be performed by different hardware accelerated techniques, such as constructive solid geometry (CSG) rendering or by using projective texture mapping.

1.1 This Work

This work extends the concept of Visual Hull by introduction of the Depth Hull. While the visual hull is an object constructed from the silhouettes of an object in 2D reference images, the depth hull is a similar body, but constructed from 2.5D, i.e. depth references images.

We show how the GPU can be used for some of the different stages of real-time novel view synthesis from depth images. The general idea is to move as much computation as possible to the GPU. Ideally, we would like to be able to upload frame data to the GPU once and then perform there all the necessary computations, including rendering. This should greatly reduce the CPU – GPU bus traffic, which often constitutes a bottleneck, and free the CPU for other tasks.

We present an approach to accelerate the surface simplification process of the depth maps using graphics hardware. Depth maps carry the same kind of information as height fields, which is a scalar distance value. This is the reason that the same algorithms that are used for simplification of height fields can be applied to depth
1.1. THIS WORK

maps. This is even more true considering that the most common application of height fields is for real-time terrain rendering, as done for example in fly-through applications [52]. In these applications the simplification process is often performed on-the-fly to allow view-dependent optimization of the terrain geometry. Height fields are often represented as grayscale images, which makes it convenient to store them as textures. Our idea is to use the graphics hardware to perform necessary simplification calculations. GPU can perform such calculation on many texels of the height field texture simultaneously. We adapt the real-time height-field simplification algorithm of Lindstrom et al. [41] for execution on the graphics hardware. The benefit of this approach is reduced computation load on the CPU. Moreover, contemporary GPUs often exhibit much higher computation power than the CPU for highly parallelizable problems.

The central part of this work is the algorithm to render the Depth Hull geometry while simultaneously computing it on the GPU. An improved hardware-accelerated algorithm for VH rendering is also presented. Depth Hull, although not under this name, has been used before in the context of 3D reconstruction [18, 48, 50]. These methods used volumetric approaches. Our Depth Hull is reconstructed directly as a polygonal object. This is important because in this way it is highly adapted for efficient real-time rendering. Our basic reconstruction and rendering algorithm is an adaptation of the hardware-accelerated algorithms for VH. We describe improvements to these algorithms which are also applicable to the DH rendering. The result is an algorithm that provides a much better trade-off between speed and quality.

Another contribution of this work is an algorithm to identify and eliminate phantom geometry during rendering. Phantom geometry is an artifact that can be present, for example, in the reconstruction of a multi-object scene. It is related to the space
occluded by the objects in the reference views. Some regions of space may be occluded by different objects as viewed in different reference views. If such a region is occluded from all cameras it may be difficult to decide whether it belongs to some object or not. Based on the reasoning that phantom geometry is usually an unwanted artifact, we developed an algorithm to identify and remove these regions during rendering of DH. Because in the VH case less information is available, it is more difficult to even identify these regions. We present some analysis of the phantom geometry in the VH case and describe possible approaches for a solution.

Portions of this work were published in [9–11] and a paper describing some of the results in Chapter 5 has been submitted to Eurographics 2008.
Chapter 2

Background

In this chapter we describe the important real-time graphics techniques that will be used in this work.

2.1 Graphics Pipeline

One of the main issues of the real-time graphics is performance. The highest performance is achieved by performing as many operations as possible in parallel. This is the reason that the graphics system breaks up the rendering process into a pipeline, see Figure 2.1. Most existing standard graphics systems share a similar architecture to the one described here. We will base our description on OpenGL because it is what is used in this work.

OpenGL is organized as a state machine. The application can control the state by issuing commands that modify OpenGL internal state variables. The current state defines the specific calculations that are carried out at each stage of the pipeline. In addition to the finite state machine, also known as fixed function architecture, the
graphics system has two programmable blocks – a vertex processor and a fragment processor. These blocks can be programmed to carry out arbitrary calculations under certain restrictions on the data flow and control flow.

In order to render a geometric primitive, the application has to send the geometry, i.e. the coordinates of its vertices along with associated per-vertex data.

In the fixed function vertex processing mode the first stage of the pipeline performs such operations as geometric transformations and lighting calculations. If programmable vertex processing is enabled, the calculations are specified by a user-defined program, the *vertex shader*. For example, the vertex shader can be programmed to perform exactly the same calculations as the fixed function logic. In order to increase performance, hardware may have several vertex processing units working in parallel on different vertices.

After all vertices of a primitive have been specified, they are transferred to the
2.1. GRAPHICS PIPELINE

primitive assembly stage. The processed vertices with associated data are grouped to form a primitive, which then is clipped against the view frustum and triangulated. The resulting triangles are rasterized by performing scan conversion. During this process triangles are discretized into fragments which are associated with per-fragment data. The per-fragment data is obtained from per-vertex data through bilinear interpolation. Fragments eventually correspond to pixels of the framebuffer and thus correspond to specific window coordinates. During rendering for some pixels there can be produced many fragments, depending on the overdraw characteristics of the scene and the specific rendering technique.

Depending on the current state, the fragments are further passed either to the fixed-function or programmable processing. The fixed-function fragment module may perform texture mapping, color sum and fog calculations. The programmable fragment processor performs calculations specified by a user-defined program, the fragment shader. Fragment shaders can perform texture accesses and arithmetic operations. As with programmable vertex processing, there can be several execution units running fragment shaders on several fragments in parallel. This highly parallelizable computation model restricts data dependencies between vertices as well as between fragments. This is also the reason that (currently) memory access for writing cannot be performed from within the shader programs. Memory reads are allowed and are done in the form of texture accesses. Instead of writing computation results to memory, shaders pass them via established channels down the pipeline. After processing, each fragment is passed further along the pipeline in the form of its color and depth.

Before the fragment is finally written to the framebuffer it passes several per-fragment operations. The most important operations used in this work are, in the order of execution, the alpha test, the stencil test and the depth test. The alpha
test allows rejecting fragments based on the comparison of their alpha color channel with a reference alpha value. If the alpha test fails, the fragment is rejected – it is not passed to further processing and does not affect the contents of the framebuffer.

The stencil mechanism maintains a buffer of unsigned integer values corresponding to each pixel of the frame buffer. For each incoming fragment the stencil test compares the corresponding value in the stencil buffer with a reference value. If the test fails, the fragment is rejected. Depending on the outcome of the stencil and the depth tests, the stencil mechanism can be configured to modify the corresponding values in the stencil buffer. The depth test mechanism maintains a floating point buffer of depth values and is mostly used for hidden surface removal. The depth test can be configured to perform different types of comparison, e.g. \textit{less}, \textit{greater}, \textit{equal}, etc.

The term \textit{framebuffer} refers to the collection of all internal buffers used by the graphics system and associated with a rendering context. Usually, the final goal of rendering is modification of a color buffer. There may be several color buffers. Other types of buffers comprising the framebuffer include multiple stencil buffers, auxiliary buffers, a single depth buffer and accumulation buffer. OpenGL supports read and write operations on pixels between the framebuffer and a user-supplied buffer. User-supplied buffers may reside in the main system memory as well as in the graphics memory. Pixel transfer between the graphics and the system memories can be a costly operation, especially in the GPU-to-CPU direction.

\section{Multi-Pass Rendering}

Multi-pass is a rendering technique, in which the geometry of the scene is sent for rendering several times. For each such pass the graphics system is configured to
2.3. PROJECTIVE TEXTURE MAPPING

Figure 2.2: Multi-pass rendering is used to produce the wireframe on a solid object effect.

perform a different kind computation. At the end the effects of the different passes are combined to produce the final result. A simple example is the wireframe over solid rendering, see Figure 2.2. The first pass draws shaded solid faces. The second pass superimposes the wireframe while taking advantage of the already computed contents of the depth buffer in order to remove the outlines of hidden triangles. More sophisticated techniques may perform a greater number of passes and more extensively use the intermediate contents of the framebuffer, e.g. as textures.

There are several mechanisms that can be used for combination of the effects of different passes into a single result. These include accumulation buffer, render-to-texture capabilities, buffer objects and programable fragment shaders.

2.3 Projective Texture Mapping

Projective texture mapping [8,19] is a technique for automatic generation of texture coordinates such that the texture image appears to be projected onto the scene geometry as if with a slide projector. The projection source is analogous to a camera and has associated position and orientation in space. For a given 3D point in object
space in order to obtain its corresponding projective texture coordinates it is necessary to apply the transformations that map this point on the projector image. These are similar to the usual transformation performed during rendering. The difference is that instead of the camera view and projection matrices it is necessary to use the corresponding matrices of the projector. Programmable shaders make it easy to perform all the necessary calculations prior to accessing textures.

### 2.4 Projective Shadow Mapping

Projective shadow mapping [20] uses the same mechanism as the projective texturing in order to cast shadows instead of images. Shadows can be cast by arbitrary geometry and on arbitrary geometry. The role of the projector is now played by a light source. The projection image now contains a depth texture, which is a grayscale representation of the scene depth as it is seen from this light source. This is a multi-pass
2.4. PROJECTIVE SHADOW MAPPING

![Projective Shadow Mapping Diagram](image)

**Figure 2.4:** Projective shadow mapping. First the depth map is computed by rendering the scene from the light viewpoint. Then the shadows are cast by projective texture mapping with depth comparison. The position and direction of the light source are shown by the arrow.

Technique. In the first pass the scene is rendered from the point of view of the light source and the resulting contents of the depth buffer are copied to the depth texture. In the second pass the shadowed scene is rendered. For each fragment projective texture mapping calculations are performed in order to sample the depth texture. The fragments $Z$ coordinate in the light reference frame is compared with the depth sampled from the texture. If the fragment is farther from the light than the texture sample, then it is shadowed. Otherwise, the fragment and the texture depths are approximately equal and the fragment is lit. Graphics hardware provides specific support for this technique by allowing depth texture formats and adding the possibility for automatic depth comparison upon depth texture access. This capability is very similar to a read-only depth buffer test and allows for different comparison functions, e.g. *less*, *greater*, *equal*, etc.
2.5 Depth Peeling

Depth peeling is a multi-pass technique that allows depth-sorted processing of the scene geometry. It is used in situations such as transparent object rendering, where the depth ordering of the primitives is important. The scene is treated as a series of depth layers, starting, for instance, with the nearest. The nearest depth layer is defined by the contents of the depth buffer after a single rendering pass with the usual hidden surface removal. Although each point may lie at different depth, it is still considered the first layer because this is the nearest geometry to the camera. The second layer is defined as the geometry that lies immediately behind the first layer, the third layer behind the second, and so on. Given the $i$-th layer, the $i+1$-st layer can be computed by performing two depth tests in a row. The first depth test is read-only and compares the incoming fragment against the depth of the $i$-th layer. The fragment is passed further only if it is strictly deeper. The other depth test is the usual hidden surface removal test, which is needed to produce the nearest of all the geometry that is deeper than the $i$-th layer.
In the graphics hardware the extra depth test is performed with the help of the projective shadow mapping mechanism. The previous layer is loaded as a depth texture. The light source of the shadow mapping is positioned to coincide exactly with the camera by using the same matrices. In this configuration each pixel of the texture exactly corresponds a pixel in the framebuffer. The depth texture effectively becomes an additional depth buffer. The hardware is configured to perform the appropriate depth comparison upon texture access. Note that this is read-only operations since the texture content cannot be modified. This test is performed using the comparison function \textit{greater} and is done before the regular depth test, which is configured to \textit{less}. After the rendering pass is complete, the depth buffer hold the contents of the next depth layer. If further depth peeling is needed, this data has to be copied to the depth texture and the process repeated.

The procedure described above produces a front-to-back traversal of the scene depth layers. In order to perform a back-to-front traversal it is sufficient to swap the comparison functions, \textit{greater} to \textit{less} and vice versa. The order of the tests, however, must remain the same.
Chapter 3

GPU-Assisted Z-Field

Simplification

Height fields and depth maps which we collectively refer to as z-fields, usually carry a lot of redundant information and are often used in real-time applications. This is the reason why efficient methods for their simplification are necessary. On the other hand, the computation power and programmability of commodity graphics hardware has significantly grown. We present an adaptation of an existing real-time z-field simplification method for execution in graphics hardware. The main parts of the algorithm are implemented as fragment programs which run on the GPU. The resulting polygonal models are identical to the ones obtained by the original method. The main benefit is that the computation load is imposed on the GPU, freeing-up the CPU for other tasks. Additionally, the new method exhibits a performance improvement when compared to a pure CPU implementation.
3.1 Introduction

The performance of the graphics hardware increases at a rapid pace. The nature of the processed data allows for the computation to be organized according to the Single Instruction Multiple Data (SIMD) principle, which, among others, has the advantage of increased throughput via parallelized execution. Each of the input vertices undergoes exactly the same computation as all others. The independence of the computations allows further improvement of the performance by physical replication of the parallel execution units. Analogously to the vertices, this is also done for the frame buffer fragments. Today, high-performance GPUs provide developers with the possibility to utilize this ever-growing power not only quantitatively but also qualitatively. GPUs such as NVIDIA GeForce FX provide hardware support for the user to intervene in the rendering pipeline, replacing the fixed-function vertex and fragment processing with flexible shader programs. Instead of just using more and more polygons, the image quality and realism is significantly increased by using advanced rendering effects that now can be programmed for execution, in real time, on the geometry and
the raster processing levels. As the programmability features extend the GPU functionality towards a general purpose CPU, it becomes more obvious that the graphics hardware can now be utilized to solve a wider class of problems, not necessarily related to rendering effects. In this work we describe how to use the programmable fragment engine to run a specific mesh simplification algorithm. The algorithm we explore was initially developed for real-time simplification of height fields. Later it was adopted for simplification of depth maps. Although coming from different sources, there is no essential difference between these two kinds of data. Both are represented by a regular rectangular grid with a height or a depth, i.e. z-value, at each grid point, relative to the x-y plane. Consequently, we use the generalized term z-field. Figure 3.1 shows examples of a height field and a depth field after simplification.

3.2 Related Work

The complexity of the polygonal representation of the z-field data can be reduced using a variety of geometric simplification methods. Most mesh simplification techniques [27,31,54] are not applicable to the problem of real-time resolution reduction of z-fields due to the relatively high computational cost of the optimization process that they use.

Several techniques have been developed specifically to address the issue of fast simplification of height field data. Gross et al. [23] produce adaptive surface meshing using wavelet transforms. Lindstrom et al. [41] developed a real-time height field simplification algorithm. Their method uses quad-tree partitioning of the height field domain and constructs a level-of-detail hierarchy, defined through a specific type of vertex removal operation. With some modifications, Chai et al. [16], proposed
using the Lindstrom simplification for depth map data, e.g. the output of a range-scanner. These come from real-time stereo reconstruction and must be processed as quickly as possible, preferably with as little computational effort as possible, at least on the CPU part. Employing the graphics hardware, whenever possible, to assist in the computation is gaining popularity, even with the fixed-function graphics pipeline [30]. With the introduction of the programmable GPU [40] the applicability of this technique has been significantly extended. Purcell et al. [51] as well as Carr et al. [15] demonstrated the feasibility of ray-tracing with graphics hardware. The GPU has also been used for physical simulation [28], matrix multiplication [33] and even solving sparse linear systems [12]. The interested reader is referred to the web-site www.gpgpu.org, which is specifically dedicated to the general purpose computations using graphics hardware.

The technique presented here is based on the simplification method of Lindstrom et al. [41]. It generates the same output, but is adapted to run its main parts in the programmable graphics hardware. By doing that we are able to achieve some speed-up and relieve the CPU of most of the computation load. The latter may be important in a client-server configuration where maximal utilization of the available resources is important to provide the best performance. Another case is a view-dependent scenario, where the simplification must be done on the client. If the client’s graphics subsystem is more powerful, it makes sense to take advantage of this in order to free-up additional CPU resources.
3.3 Z-Field Simplification

The surface triangulation of the z-field before the simplification is defined as follows. The smallest sub-mesh is defined to be a mesh over $3 \times 3$ vertices, and successively larger sub-meshes are formed from the smaller ones by arranging them into $2 \times 2$ arrays, as shown in Figure 3.2. Defined in this way, a sub-mesh of level $l$ has the dimensions $(2^l + 1) \times (2^l + 1)$ and is the highest resolution block that can be defined.
over these vertices. A block is a discrete level of detail sub-mesh, defined over a square region of vertices. Discarding every other row and column of four higher resolution blocks creates one block at the next resolution. The triangulation before the simplification consists of the highest resolution blocks. The discrete level of detail block hierarchy corresponds to a quad-tree structure. The simplification is performed in two phases. At the coarse-grained phase, entire blocks are considered, and the blocks of the lowest possible resolution, whose maximum error does not yet exceed a certain threshold, are chosen to represent the mesh. At the fine-grained phase, individual vertices are considered for removal. When a vertex is removed, two adjacent triangles, sharing this vertex at the same resolution level are merged into one triangle. The resulting triangle and its co-triangle are then considered for further simplification (Figure 3.3). The view-independent simplification error at the removed vertex is defined as:

\[ \delta_B = \left| B_z - \frac{A_z + C_z}{2} \right| \]

The view-dependent version is the error projected to the viewer image plane. To ensure validity of the resulting triangulation, the removal process is ruled by the vertex dependencies, induced by the parent-child relations (Figure 3.4). The removed vertex, unless it is on the boundary, always has exactly four neighbors, of which two are its parents and two its non-parent neighbors. If the vertex has a degree greater than four, some of its children are present and hence the vertex cannot be removed. After removal of the vertex, the resulting quadrilateral is triangulated by connecting the two non-parent vertex neighbors, which can be viewed as two pair-wise merges of the four triangles containing the vertex.

Two attributes are associated with every vertex: enabled and activated. A vertex
CHAPTER 3. GPU-ASSISTED Z-FIELD SIMPLIFICATION

Figure 3.4: The two kinds of the parent-child relationship. The arrows point from children to parents.

is enabled if it is activated or any of its children is enabled. A vertex is activated if its simplification error exceeds the threshold. During the fine-grained simplification, every vertex is evaluated to determine whether it is enabled or not. It is for acceleration of this process that we propose using the programmable graphics hardware. After the correct enabled states of all the vertices are determined, the final construction of the mesh triangles can be easily performed by a DFS traversal of the vertex dependency tree [16, 41].

3.4 Modified Algorithm

According to the algorithm of Lindstrom et al. [41], when a vertex is evaluated, if its enabled state is changed, a notification is sent to both of its parents. A notified vertex checks whether its enabled state should be altered, and if it does, sends, in turn, a notification to both of its parents, creating a recursion. If none of the children of the notified vertex is enabled, its state is dictated by its activated attribute. Computation of the correct value of the activated attribute is relatively expensive. For this reason, in the original formulation it is not performed during the notification, rather only during the vertex evaluation if none of its children is enabled. This allows avoiding
unnecessary computations but can result in a one-frame delay before the activated
attribute is corrected. The delay is possible if at least one of the vertex children is
evaluated after the vertex itself. Consider the case where vertex u is the only enabled
child of vertex v. Suppose that v is evaluated before u. For the current frame,
activated(v) will not be computed because one of its children is enabled. Suppose
that after the evaluation u became disabled. Now activated(v) is important, but it
will not be computed until next frame. For similar reasons, during the evaluation for
the same frame, notifications may travel the same route more than once.

A key observation is that these events may be avoided by arranging the vertices
considered for the simplification in such a way that children are always evaluated
before their parents. Thus, when a vertex is considered, all necessary information
about its children is available, up-to-date and no notification is necessary. This can
be achieved by grouping the vertices according to their level in the simplification
hierarchy, starting with the leaves at level 0, which are the vertices whose parents are
their immediate neighbors above and below or on the left and on the right, Figure 3.5.
For a grid of \( n = (2^k + 1) \times (2^k + 1) \) vertices there are \([n/2]\) leaves and \(2k+3\) levels. The
highest level consists of a single vertex. All vertices in the same level are independent
and can be processed in parallel. The simplification process proceeds level by level,
starting at level 0. With the given viewing parameters and the error threshold, the correct simplification of the z-field for the current frame is achieved after a single sweep of all levels.

### 3.5 Simplification in Graphics Hardware

The fact that the processing of the vertices of the same level can be done independently and in parallel and the fact that the problem domain naturally corresponds to the fragments of the frame buffer allows natural mapping of our solution onto programmable graphics hardware. Each vertex of the z-field is associated with a fragment of the frame buffer. This way the z-field can be viewed as a grayscale image where each pixel represents a z-value. The unsimplified input data is thus represented by a 2D texture. The output of the algorithm is written into the color buffer as a binary mask, specifying for each z-field vertex, associated with the corresponding pixel, its enabled state. Essentially, the output is a 1-bit (monochrome) image. The algorithm works in a multi-pass fashion. Every five passes compute the enabled state for all vertices of one particular level. The first level is an exception and is computed in a single pass. The level to be processed is chosen with the help of a special frame arrangement. The effect of individual vertices on the final result is controlled with the depth test mechanism.

#### 3.5.1 Frame Arrangement

In order to avoid the unnecessary rasterization of vertices belonging to levels other than the current, we rearrange the vertices inside the frame. Instead of the initial interleaved arrangement, the vertices of each level are grouped as illustrated on the
3.5. SIMPLIFICATION IN GRAPHICS HARDWARE

Figure 3.6. As a result, each level is covered by a single continuous rectangle, which does not cover any vertices of other levels. Thus, in order to process a given level, it suffices to render a quad covering the corresponding vertices. Only vertices of the current level will be rasterized. The arrangement mapping is precomputed and stored in a 2D texture which is used for computing the correct texture coordinates when accessing the z-field texture, see Figure 3.8. During the computation of the arrangement, it is necessary to know, for each vertex, which level it belongs to. This can be precomputed by a simple algorithm, similar to the notification mechanism described above. Starting with the leaves at level 0, the vertex level is marked, and then the procedure is called recursively for the parents of the marked vertices, with the level number incremented by one.

3.5.2 Rendering Passes

The computation is independent and identical for every vertex and thus is particularly suitable for the SIMD approach, which is implemented very efficiently in the latest graphics hardware. However, this computational model imposes a number of constraints, one of which is linear control flow. Currently, conditional execution, even if supported, imposes a significant overhead. In order to eliminate conditional code we had to split the computation into several passes. A depth test is used for selection of the vertices that will participate in a pass. The total number of passes is less than five times the number of levels and is logarithmic in the z-field resolution. In the first four passes, for each vertex the enabled flags of its children are tested; these flags were computed at the previous level. During any of these passes, if an enabled child of a vertex is detected, the vertex is marked as enabled and its fragment depth is modified, so that it will not participate in the subsequent passes. Only the vertices
with all children disabled make it to the fifth pass of the current level. In this pass
the delta for each vertex is computed and compared against the threshold. If this test
fails, the vertex becomes disabled, otherwise it is enabled. This core computation on
the z-field vertices is performed in two fragment programs. Processing of one level
consists of the following steps.

1. Prepare the enabled flags, computed at the previous level, for read access at the
current level.
2. Perform the four children checking passes using the first fragment program. At
each pass, another child of every vertex is tested.
3. Perform the delta test pass, executing the second fragment program.

Step 1 is not performed for the first level because it does not use any previously
computed flags. For any other level it amounts to copying the color buffer rectangle, corresponding to the previous level, into a dedicated 2D texture which serves as a read-only memory for the fragment program. This relatively expensive pixel copying operation is needed because, currently, in the graphics hardware there is no such memory unit which is accessible both for reading and writing at the same time by the fragment program. Precisely for this reason the existing "render texture" OpenGL extensions do not seem to be directly applicable here (see the Discussion section).

However, due to the arrangement of the levels in the frame, no redundant copying is performed and each pixel is copied only once. A pass is performed by rendering a single quad which covers vertices of the corresponding level. After the quad is rasterized, the selected fragment program is executed for each fragment of the rasterization. For efficiency, the coordinates of each of the four children are precomputed and stored in four different 2D textures. The two 16-bit coordinate values are packed into a 4-byte RGBA texture word. The first program reads child coordinates from the corresponding texture and uses them to access the texture of the enabled flags. The second program uses the same scheme for accessing the precomputed neighbor coordinates. In order to mark a vertex as enabled, the fragment programs issue the color value of 1.

To prevent participation, in the subsequent passes, of a vertex for whom an enabled child is detected, the first fragment program issues the depth value of 1. For both programs the default fragment depth is zero, which is also the clearing value for the depth buffer. The depth test for the first program will succeed only if the incoming fragment depth is greater than that stored. For the second program the depth test succeeds if it is equal. Figures 3.7 and 3.8 summarize the execution of the two types of rendering passes. Each vertex of the output z-field corresponds to exactly one
output pixel. Therefore, the resolution of the frame buffer and all the textures must be sufficient to represent the finest underlying resolution of the simplified z-field. This resolution, however, does not necessarily have to be the resolution of the input z-field. In fact, just like any discrete image, the input z-field can be filtered and resampled, and the graphics hardware provides efficient means, e.g. mipmaps, for doing that.
Figure 3.8: Rendering pass 5 execution diagram. Solid lines denote control flow, fat dark-gray lines denote data paths, dashed lines – texture accesses. This is the second fragment program – the fifth pass.
3.5.3 Meshing

The final meshing is done on the CPU. After the enabled flags are computed on the GPU they need to be read from the frame buffer into the main system memory. To minimize the time that is spent on the expensive pixel-read operation, we pack the enabled bits, in sets of eight, into bytes. These bytes are stored in one of the four color channels of the color buffer. This way we only require one eighth of the total number of pixels and for each pixel we only need to read a single one-byte channel. The packing is performed in one additional pass, after all the computation passes. In this pass the third fragment program is executed. Its function for each fragment amounts to reading the corresponding consecutive eight enabled flags, starting at an x coordinate which is a multiple of 8, and packing them together into one channel. The rendered rectangle spans the entire height and one eighth of the width of the frame, although the entire enabled flags texture is sampled. The meshing procedure is exactly the same as in the CPU counterpart, but with a slight overhead from remapping and unpacking when reading the values of the enabled flags.

3.6 Results

We implemented the technique described above using OpenGL and Cg - a high level programming language for graphics. The code runs in hardware on graphics cards based on the GeForce FX or any other GPU whose driver supports the ”NV_fragment_program2” OpenGL extension. Table 3.1 summarizes the runtime results. The CPU used in our tests is a 2.8GHz Xeon with 1GB RAM. The GPU used is a 400MHz GeForce FX 5900 with 128MB RAM. The runtimes refer to the
3.6. RESULTS

<table>
<thead>
<tr>
<th>Z-field dataset</th>
<th>Initial resolution</th>
<th>Retained vertices</th>
<th>Simplification time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seliun</td>
<td>257x257</td>
<td>511</td>
<td>0.0088 0.0066</td>
</tr>
<tr>
<td>seliun</td>
<td>257x257</td>
<td>1223</td>
<td>0.0093 0.0076</td>
</tr>
<tr>
<td>elevh</td>
<td>257x257</td>
<td>821</td>
<td>0.0087 0.0064</td>
</tr>
<tr>
<td>elevh</td>
<td>257x257</td>
<td>2035</td>
<td>0.0093 0.007</td>
</tr>
<tr>
<td>k.baram</td>
<td>513x513</td>
<td>641</td>
<td>0.0333 0.0215</td>
</tr>
<tr>
<td>k.baram</td>
<td>513x513</td>
<td>1113</td>
<td>0.0337 0.0218</td>
</tr>
<tr>
<td>k.baram</td>
<td>513x513</td>
<td>5172</td>
<td>0.0356 0.025</td>
</tr>
<tr>
<td>shomeron</td>
<td>513x513</td>
<td>1024</td>
<td>0.0333 0.022</td>
</tr>
<tr>
<td>shomeron</td>
<td>513x513</td>
<td>4177</td>
<td>0.0355 0.0243</td>
</tr>
<tr>
<td>c1AvgHand</td>
<td>513x513</td>
<td>6153</td>
<td>0.0364 0.0249</td>
</tr>
</tbody>
</table>

**Table 3.1:** Simplification runtimes. The CPU column refers to the pure CPU implementation of the algorithm.

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragment programs execution</td>
<td>62%</td>
</tr>
<tr>
<td>Z-texture upload</td>
<td>16%</td>
</tr>
<tr>
<td>Meshing</td>
<td>14%</td>
</tr>
<tr>
<td>Texture bind &amp; copy</td>
<td>3%</td>
</tr>
<tr>
<td>Frame buffer read</td>
<td>3%</td>
</tr>
<tr>
<td>Cg parameter manipulations</td>
<td>2%</td>
</tr>
<tr>
<td>Total</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table 3.2:** Time distribution between the components of the GPU-assisted simplification for a typical run.

The complete simplification process, which for the GPU-assisted case includes texture upload, execution of the fragment programs, frame buffer reading and meshing. The runtime seems to be dependent almost entirely on the resolution of the input z-field, although we have some times experienced a very slight decrease in speed with an increase in the output mesh resolution. The z-field texture upload consumes about 16% of the total simplification time, which is dominated by the execution of the fragment programs, see Table 3.2. This texture upload time could be reduced by using different extensions, e.g. "compressed textures". Many redundant fragment program
executions could be avoided by early fragment rejection from depth or stencil tests, if supported and correctly employed. Although our implementation still leaves a lot of room for further optimization, the GPU already outperforms the CPU in the z-field simplification.

3.7 Discussion

Our initial Cg implementation contained a single fragment program and executed only one rendering pass per simplification level. However, this program included a great deal of conditional code, which affected the performance significantly. The reason for this is that the GPUs do not, yet, support conditional execution at the instruction level. Although Cg allows conditional statements, it is implemented by executing all the conditional code and then conditionally assigning the result. Elimination of all of the conditional code led to splitting the main program into two and multiplying the number of passes by a factor of 5. We believe that this significant increase in the number of passes, will be avoided in the future when GPUs with a more advanced instruction set are available. A possible alternative to the five passes per level is a two-pass approach, where the four children testing passes are unified into one. We could determine whether a vertex has enabled children by testing the sum of their enabled bit values. While not requiring any conditional code, this approach would perform many redundant texture accesses, which could not be avoided even by employing early fragment rejection. In our experiments, without early fragment discards, the four-pass and the one-pass children testing performed similarly. The time cost of the pixel copying is quite low, less than one millisecond. Still, it could be eliminated almost completely by using a double p-buffer. A P-buffer is an OpenGL extension
3.7. DISCUSSION

Figure 3.9: (a) A height field. (b) The enabled mask, computed on the GPU. (c) and (d) The resulting triangulation.

Figure 3.10: Simplified height fields.

For accelerated off-screen rendering. When processing one level, one p-buffer could be written to, while the other could be read from, by selecting it as a texture. For the next level the two p-buffers would be swapped. In the context of our work, we were particularly interested in simplification of depth maps that can change at every frame and have limited resolution. However, our method can be used for simplification of large height fields as the fine-grained step, complementary to the coarse-grained block-based simplification.
Chapter 4

Free-Viewpoint Video from Depth Cameras

Depth cameras, which provide color and depth information per pixel at video rates, offer exciting new opportunities in computer graphics. We address the challenge of supporting free-viewpoint video of dynamic 3D scenes using live data captured and streamed from widely-spaced viewpoints by a handful of synchronized depth cameras. We introduce the concept of the depth hull, which is a generalization of the well-known visual hull. The depth hull reflects all the dense depth information as observed from several centers of projection around the scene. It is the best approximation of the scene geometry that can be obtained from a given set of depth camera recordings.

We first present a general improvement to the best existing visual hull rendering algorithm, which is of independent interest. We then use this to contribute a hardware-accelerated method for rendering novel views from depth hulls in real-time. This method is based on a combination of techniques from projective shadow mapping and constructive solid geometry (CSG). Our rendering method achieves high-quality
results even when only a modest number of depth cameras are deployed. They are applicable to any set of images with accompanying dense depth maps that correspond to arbitrary viewing positions around the scene.

We provide experimental results using a system incorporating two depth cameras recording a dynamic scene. To the best of our knowledge, these are the first results on free-viewpoint image synthesis using commercially-available depth cameras.

4.1 Introduction

Free-viewpoint video is an emerging area of active research in computer graphics. The goal is to allow the viewer of a video dataset, whether recorded or live, to freely and interactively change his viewpoint, despite the footage being recorded by just a small number of static cameras.

Free-viewpoint video relies on novel view synthesis from the acquired set of video images. Novel view synthesis means rendering the scene from a viewpoint different from those acquired by the physical cameras. A free-viewpoint video system then enables the user to freely navigate his viewpoint through the scene with a virtual camera. Potential applications can be easily imagined in the sports, gaming, entertainment, and defense industries.

While novel view synthesis has been studied for the last decade, beginning with the pioneering work of Kanade [29], progress has been limited due to the lack of accurate dynamic scene geometry information. Although some encouraging results have been obtained, it seems that unless a very large number of cameras are used, it is difficult to obtain high-quality results for omni-directional viewpoints. Indeed,
Figure 4.1: Novel view rendering (bottom) using four reference cameras (top). Two depth cameras (top row) were used for geometry only and two video cameras (middle row) were used for texture.
the more successful systems employ 32-100 cameras. Kanade describes a system employing 51 cameras [29]. Unfortunately, the more cameras used, the more processing is required to take advantage of all the cameras and the more the system becomes sensitive to camera calibration inaccuracies, so application speed will deteriorate as the image quality improves. On the other hand, a small number of cameras do not capture sufficient information about the scene in order to be able to render novel views reliably. Accurate and dense depth information is needed. While scene depth can be reconstructed from close pairs of video cameras using software stereo techniques, it is notoriously difficult to do so reliably or fast, so high-quality real-time stereo is still elusive. Instead, a camera that can provide reliable and dense depth information at video rates would constitute a breakthrough in novel view synthesis of dynamic scenes.

Over the last few years, new camera technologies capable of real-time depth image acquisition have been developed. These cameras, called depth cameras – or Z-Cams for short – provide a depth value for each pixel, along with the regular RGB color values. Various sensing technologies are used in Z-Cams, such as time-of-flight infrared [1,3], stereo [2,4] or structured light [6], and advanced prototypes are available.

Unfortunately, for novel view synthesis just rendering the geometry captured by the Z-Cams without any additional processing is not satisfactory. Because of occlusions, there will typically exist parts of the scene that are not visible from any of the Z-Cams, but are visible from the novel viewpoint. This will result in ”holes” in the rendered objects, which occurs more frequently as a smaller number of Z-Cams are used.

In this paper we present an efficient reconstruction and rendering solution which greatly reduces the visual artifacts in the occluded areas by providing a plausible
estimate of the missing geometry. We call this estimate the depth hull. We describe techniques to construct and render depth hulls in real-time. We also provide experimental results using our methods with real-world data acquired by a synchronized setup of two depth cameras and a few video cameras. To the best of our knowledge, these are the first free-viewpoint renderings made using depth cameras.

4.2 Related Work

The field of image-based rendering (IBR) deals with rendering images based on other images of the scene and little explicit geometric information. This contrasts with the traditional approach of using polygonal 3D models, since, in many cases, they are simply not available, or difficult to obtain in a complete manner. Pure image based techniques (which use no explicit geometry) like light field [35] and lumigraph [22] rendering are known to have the power of reconstructing a complex scene appearance. However, they are computationally expensive and require storage and processing of large amounts of data, which makes them, as yet, unsuitable for real-time applications such as live video streaming.

More practical image-based techniques attempt to extract some geometric information from the image and render novel views based on that. The approach closest to that we advocate uses the concept of the visual hull, introduced by Laurentini [34]. It is an important concept in the field of 3D acquisition from silhouettes using space carving [32]. Silhouettes are typically extracted from multiple images of the scene, and the visual hull is the best approximation of the scene geometry consistent with those silhouettes. Hence it makes sense to use it in lieu of the scene geometry when rendering a novel view based on multiple video cameras. Matusik et al. [44] showed
how to use image-based visual hulls for novel view synthesis without actually constructing them. Later Matusik et al. [43] described how to construct visual hulls as polyhedral objects. The advantage of using polyhedral visual hulls is that it allows hardware accelerated polygon rendering as well as some geometric processing. Recently, Li et al. [36-38] demonstrated how to perform hardware-accelerated rendering of the visual hull even without explicit construction of its geometry. Another hardware accelerated technique which does not require explicit geometry reconstruction is based on photo-consistency [39,56,61]. The rendered object is bound by the visual hull and it is a tighter fit of the true scene geometry.

Although the polygonal approach to novel view generation seems to be natural, a number of other approaches are possible. A volumetric approach has been used by Moezzi et al. [45] and Hasenfratz et al. [25]. The voxels occupied by the model are identified by a technique similar to the visual hull volume carving technique. However, for rendering, the model is still converted to polygons, e.g. using Marching Cubes.

The first attempt to use explicit and dense (pixel resolution) depth information for novel view synthesis was probably made by Kanade et al. [29]. In their ”virtualized reality” system, they acquired depth images using stereo pairs, and rendered novel views by simply rendering the depth maps from the new viewpoints. Pulli et al. [50] used several color images registered with range images, also acquired using stereo techniques, to create novel views of a static object. The focus of these attempts, and those of their contemporaries, is on the reliable acquisition of depth images using stereo which are correlated well with the color images. They do not make much effort to obtain the best quality novel views possible. Pajarola et al. [47] describes a system to render novel views based on registered color and depth buffers acquired from six axis-aligned directions. The focus there is on the blending of the multiple reference
views and rendering using the GPU. A recent work of Waschb"usch et al. [57] describes a system for recording and playback of dynamic 3D scenes. They use structured light-assisted space-time stereo for offline depth map computation and point-based representation for novel view rendering. None of these works attempt to approximate the missing geometry, rather rely on a sufficient coverage of the scene. In contrast, we assume that the geometry acquisition problem is more or less solved. We obtain our dense registered color and depth data from depth cameras (Z-Cams). Our focus is to efficiently approximate the missing geometry when a small number of Z-Cams are used, and efficiently render the result using programmable graphics hardware. The filled geometry is projectively textured by images from additional video cameras that increase the overall object coverage.

Geometry reconstruction from several range images has been attempted before by Pulli et al. [18, 48, 50]. The resulting object is, in fact, a volumetric version of our depth hull. However, their goal was to generate a complete object representation, whereas our only desire is to render the object from a novel viewpoint. Moreover, since these authors used a volumetric approach, their solution was neither meant for nor applicable to real-time viewing of a dynamic scene, where the entire process, from the acquisition to the rendering, must be performed in a fraction of a second.

4.3 Contribution

To use the depth information obtained from the Z-Cams, we define the depth hull concept. It is a generalization of the well-known visual hull and we show its relationship to the other types of known hulls. The use of the depth hull provides an important advantage over the existing methods for rendering novel views using depth
4.4. THE VISUAL HULL

information. It allows filling in the missing parts of the geometry, thus providing a complete and conservative approximation of the rendered scene. It is the best approximation of the scene geometry consistent with the given depth maps. We first describe an improvement to existing algorithms for rendering the visual hull. This is of independent interest. We then extend this algorithm for depth hull rendering. The algorithms use programmable graphics hardware and are based on projective shadow mapping and CSG rendering. The resulting method is shown to be capable of generating quality novel views of 3D scenes at real-time rates. We demonstrate this on dynamic scenes of real-world objects captured using commercially-available Z-Cams.

4.4 The Visual Hull

4.4.1 Definitions

The external visual hull (EVH), as defined by Laurentini [34], is the maximal geometric object consistent with all possible silhouettes of the original object as it is observed from outside its convex hull (CH). By definition, the external visual hull is contained in the convex hull of the object. When only a finite and small number of static cameras are considered we define the visual hull of an object relative to a viewing region. In our case, the viewing region consists of a finite number of camera viewpoints. From now on, the visual hull relative to a set of camera viewpoints will be called simply the visual hull (VH), see Figure 4.2. Notice that, unlike the EVH, a VH of an object may not be contained in the CH. Sometimes we will call the set of camera viewpoints the reference views.

The visual hull from a set of cameras is the intersection of their visual cones. The
visual cone of a view is defined as the extruded surface originating at the camera location and passing through the object’s silhouette. Since each such cone contains the object, so will their intersection.

The visual hull is only a rough approximation of an object’s geometry because it does not capture many of its concavities. To a certain degree this may be overcome by the consensus approach, otherwise known as photo-consistency. The geometric object reconstructed using this method is called a photo hull (PH). The photo hull is contained in the visual hull and provides a better approximation of the object’s geometry at its concavities. However, it still leaves much to be desired.

4.4.2 Rendering The Visual Hull

Visual hulls may be rendered using pure image-based algorithms [44, 58] or from a reconstructed 3D polygonal model [43], in which case the graphics hardware can be utilized. Alternatively, the graphics hardware can be used for simultaneous reconstruction and rendering of the visual hull from the new viewpoint. We briefly review
4.4. THE VISUAL HULL

![Figure 4.3](image_url)

**Figure 4.3:** Rendering quality and frame-rates of different visual hull (VH) rendering algorithms.

here two techniques [36, 38] of the third kind, as our rendering algorithms will be based on them.

A common feature of these two techniques, as well as of other polygonal techniques is that the silhouettes are represented as piecewise linear curves, i.e. chains of segments, and hence their corresponding visual cones are polygonal surfaces.

As demonstrated by Li et al. [38], it is possible and even advantageous to separate the geometry rendering and the texturing stages of the VH rendering. The VH geometry rendering stage generates the contents of the depth buffer, and then the faces are rendered again, but with the depth test function set to EQUAL. Polygon rasterization fragments that pass this test are guaranteed to belong to the VH, hence should be textured. This is performed by a process called projective texture mapping [26], which, unlike the usual (parametric) texture mapping, uses the 3D vertex positions to compute the texture coordinates. The reference images (those captured from the reference views) are combined to one texture using a weighting scheme. Several weighting schemes are described in the literature [14, 38, 49]. The particular choices of the geometry rendering and texturing schemes are independent of each
other. We will concentrate on the geometry part.

**Rendering With Projective Textures**

First, binary silhouette images are generated from the reference images by extracting the object contours and flagging all pixels within the contours. The VH is rendered by sending the polygons of the visual cones – the extrusions of the silhouettes – to the renderer. Projective texture mapping with the binary silhouette images as textures is used to cull those fragments of the rasterized polygons that lie outside the silhouette in at least one reference view. Fragment programs [40] are used to determine whether a fragment is within a silhouette. In the graphics pipeline, the fragment’s texture coordinates are obtained by interpolation of its polygon’s texture coordinates, which are the projective texture coordinates computed from the 3D vertex positions by the vertex program. For each fragment, all reference silhouette images are sampled, except for that defining the visual cone currently being rendered. If all samples return 1, then the fragment belongs to the foregrounds of all reference views, hence to the visual hull. Otherwise, i.e. when at least one sample is 0, the fragment is culled by the subsequent alpha test.

**CSG Rendering**

This technique uses the constructive solid geometry (CSG) approach. CSG rendering allows rendering of complex objects described by boolean operations on simpler objects: union, intersection and subtraction. Since the visual hull is itself a CSG object, the techniques are applicable. Recently, Guha et al. [24] have shown how to render CSG objects efficiently using programmable graphics hardware. In the case of the visual hull it is only necessary to render the intersection of the visual cones; union
and subtraction operations are not required.

The technique of Guha et al. renders the intersection of the given objects in a multi-pass manner, processing depth layers in a front-to-back order; this operation is called depth peeling. Since pixels in the depth buffer are overwritten many times, for a given pixel and depth value, the index of the depth layer containing it is the number of smaller depth values written to that pixel. For each depth layer, pixels belonging to the intersection are identified and protected from being overwritten later by the stencil buffer mechanism. The depth peeling is performed with the help of a viewport aligned texture, which stores the contents of the Z-buffer from the previous depth layer. When using this texture, the hardware checks whether the fragment is deeper than the previous depth layer. If not, the alpha channel is set to 0 and the fragment is rejected by the following alpha test. Otherwise, the fragment is passed on for further processing, which includes the usual depth test. This combination of two depth-tests, one by the texture (configured to GREATER) and one by the Z-buffer (configured to LESS), is called the two-sided depth test.

Identifying whether the pixel belongs to the intersection is done by examining the difference between the number of the back-facing and the front-facing fragments covering this pixel and belonging to the deeper layers. The pixel belongs to the intersection if and only if this difference is equal to the number of the reference views.

4.4.3 Improved Projective Rendering

The CSG technique provides accurate rendering of the visual hull. However, it makes heavy use of the raster-filling component of the graphics pipeline, performing many rasterizations and internal buffer copying operations for each frame. In contrast, the projective texturing technique is much lighter hence up to an order of magnitude
faster, but gives less accurate results. See Figure 4.3. The visual artifacts in the projective VH rendering are due to aliasing, which is always present in projective textures, and due to the fragment to silhouette comparison not being performed in the view reference frame.

As a precursor to our DH rendering technique, we describe now a rendering scheme which is based on the projective VH technique, but achieves image quality comparable to the CSG renderer, with the help of two additional passes and depth peeling operations. The new method is still considerably faster than the CSG renderer and thus achieves a better time/quality tradeoff than both the CSG and pure projective algorithms.

Our key observation is that the main artifacts of the projective VH occur at the intersections of the faces from different reference views. Since the polygon fragments are compared against the silhouettes in the projective textures rather than against other fragments, the resulting polygon trimming by fragment rejection is not aligned well with the actual geometry intersection as it is seen from the novel viewpoint. Fortunately, the combination of the texture access and the alpha test of the projective texture rendering algorithm can be configured in such a way that the trimming will be performed in a conservative manner, meaning that the polygon margins will always extend a little beyond the intersection, guaranteeing that no cracks will occur. The remaining problem is to get rid of these margins. This can be done using an inverted depth test. Additionally, the algorithm uses a depth texture with hardware comparison to perform the two-sided depth test used during the depth peeling. Algorithm 4.1 describes the procedure in pseudo-code.
4.5. THE DEPTH HULL

We now show how to adapt our improved VH rendering algorithm to datasets which contain also depth maps, such as those acquired using Z-Cams. Although depth maps provide much more 3D information than just silhouettes, our knowledge about the object geometry is still incomplete if they do not cover the entire scene. Luckily, for visualization purposes most of the time we are interested only in the visible geometry rather than a complete description of the object.

Together with the visual hull, Laurentini defined the internal visual hull (IVH) [34]. Intuitively, it is the minimal geometric object, enclosing the original object, for which every surface point is visible from at least one viewpoint outside of its convex hull. Note that the IVH of an object is not necessarily the object itself, because the object may have deep concavities with surfaces unexposed to any viewpoint outside of the convex hull, while its IVH would not have these.

Consider a visual cone cast by an object as it is seen from a Z-Cam. If this Z-Cam were a point light source, the part of the visual cone behind the depth map would

### 3PassVH

#### Pass 1.
Configure projective VH for conservative trimming, back-face culling and depth test - LESS.
Render projective VH
Copy Z-buffer to the depth texture

#### Pass 2.
Configure projective VH for conservative trimming, front-face culling, depth test - LESS, depth texture comparison - GREATER, stencil mark - 1.
Render projective VH
Copy Z-buffer to the depth texture

#### Pass 3.
Configure projective VH for conservative trimming, back-face culling, depth test - GREATER, depth texture comparison - LESS, stencil test - EQUAL 1.
Render projective VH

**Algorithm 4.1:** Improved VH Rendering Algorithm.
be exactly the shadow volume, or the umbra, cast by the object and the point light source.

**Definition:** The depth hull (DH) of an object relative to a set of Z-Cams is the intersection of its umbrae from all the Z-Cams in the set.

The relationship between the depth hull and the internal visual hull is the same as the relationship between the visual hull and the external visual hull. Relative to an infinite set of Z-Cams located outside of the object’s convex hull and looking at the object from all possible directions, the DH is exactly the IVH.

Since the umbra is contained in the corresponding visual cone, the intersection of the umbrae is contained in the intersection of the visual cones, i.e. \( \text{DH} \subseteq \text{VH} \). In general, for an object \( S \) and a set \( C \) of Z-Cams located outside \( \text{CH}(S) \), the following relationships hold:

\[
S \subseteq \text{IVH}(S) \subseteq \text{EVH}(S) \subseteq \text{CH}(S) \cap \text{VH}(S, C)
\]

\[
S \subseteq \text{IVH}(S) \subseteq \text{DH}(S, C) \subseteq \text{PH}(S, C) \subseteq \text{VH}(S, C)
\]

\( \text{DH}(S, C) \) is the best conservative approximation of the geometry of \( S \) obtainable by viewing it from \( C \).

### 4.6 Rendering Depth Hulls

Prior to rendering, the depth maps may be simplified and triangulated. This process is very fast and is performed on the fly [41, 41], accelerated by using the GPU, as described by Bogomjakov and Gotsman [9]. The result is a set of triangle meshes - one for each depth map - which are called the depth meshes [47]. Triangles at the depth discontinuities correspond to the internal and external silhouettes of the object. The external silhouette triangles coincide with the triangles that would be used to
represent visual cones for the VH rendering. Identification of the silhouette triangles may be done by any of the fast segmentation techniques discussed in [46]. Probably the simplest way is by comparing the depth span of a triangle to some threshold.

When rendering the DH, the silhouette and non-silhouette triangles are rendered separately. The silhouette triangles are rendered using projective shadowing [20], which is very similar to the projective texture mapping used for the VH. The technique proceeds as follows: The depth maps are loaded into the graphics hardware as depth textures, i.e. one-channel 2D textures configured for depth comparison and other specific operations by enabling the DEPTH_TEXTURE mode. The rasterized fragments of the silhouette triangles are processed by a fragment program. For every fragment, each depth texture is accessed in a projective manner. For this purpose the texture coordinates are computed from the fragment’s 3D coordinates by transforming them into the coordinate system of the corresponding reference view. The transformed coordinates represent the 2D texture coordinates \((u,v)\) along with the third coordinate, which represents the fragment’s depth as it is seen in this reference view. The depth texture is accessed using the \((u,v)\) coordinates, and the resulting value is compared with the fragment depth. If the fragment is closer than the depth texture sample, then it does not belong to the umbra of this view. If a fragment does not belong to at least one umbra, it does not belong to their intersection and hence should not be rendered. Such a fragment is culled using the alpha test.

The non-silhouette triangles are rendered in the standard manner, with the reference color images applied to them as regular non-projective textures.

Our adaptation of the projective technique for rendering a VH to rendering a DH, as described above, also enables us to use the 3-pass algorithm for DH rendering. The
3-pass algorithm (Algorithm 4.1) directly uses the projective VH with the only condition that it is configured for conservative polygon trimming. This configuration is equally applicable in the projective shadowing mode. So the 3-pass DH rendering algorithm works exactly like the 3-pass VH algorithm, but with the projective VH subroutine replaced by the projective shadow DH.

4.7 Experimental Results

We have implemented the three VH rendering algorithms described here and their adaptations for DH rendering. The hardware used in our tests is a 1.6 GHz Pentium M PC with 1GB RAM; the graphics card is equipped with an nVidia GeForce 6800 GPU (128MB RAM). The rendering is performed using the OpenGL API, with vertex and pixel shaders written in Cg. A demonstration video may be found at http://www.cs.technion.ac.il/~alexb/freeviewpoint.avi.

4.7.1 Camera Setup

Our camera setup (see Figure 4.4) consists of two Z-Cams and three ordinary video cameras, which we call V-Cams. Each of the Z-Cams is a Vialux Z-snapper [6] structured-light camera. When operating on its own, at the 1/2 VGA resolution in the off-line processing mode, each such camera is capable of producing up to 10 frames per second of geometric data. However, at its highest sub-millimeter precision in the on-line processing mode that we have to use (for technical reasons) in the multi-camera setup, the frame rate drops to 1.33 fps. The Z-snapper geometric data consists of four images – three floating point images for the X, Y and Z coordinates and a Boolean image of the pixel validity mask. In addition to the geometric data
4.7. EXPERIMENTAL RESULTS

Figure 4.4: Camera setup.

The Z-snapper provides a grayscale image of the scene. The V-Cams are Point Grey Firefly2 cameras. All five cameras are synchronized over a LAN, and this reduces the effective frame rate by a factor of two because the two Z-snappers have to operate sequentially.

The video cameras are calibrated and registered using standard calibration techniques implemented in the Caltech package [53]. The geometric data from different cameras is registered using the Iterative Closest Point (ICP) procedure; we use the ICP implementation found in Stanford’s Scanalyze software [7].

Incorporating V-Cams into the rendering is straightforward – they are used at the texturing stage as sources of projective textures. When texturing the silhouette triangles only they can be used because the regions behind the silhouette triangles are occluded from all the Z-Cams. However, each depth map is textured using its own color image, exclusively.
4.7.2 General Performance

As expected, the rendering speed of the 3-pass projective algorithms decreased by a constant factor of approximately 2.5-3 relative to the original projective algorithm. However, as can be seen in Figure 4.5, most of the artifacts at the mesh intersections have been eliminated.

Color plate Figure 4.6 shows a sequence of novel views generated using our depth hull rendering algorithms from the output of just two Z-Cams and two V-Cams, positioned around a moving and deforming hand. We emphasize that the Z-Cams are static. When the number of cameras is small, bad positioning may result in poor quality and phantom objects in some views. We found that near-opposite positioning of two Z-Cams is least sensitive to inaccuracies in registration, synchronization, coverage, low acquisition frame rate and other problems. As can be seen, these two Z-Cams are sufficient to generate quality renderings of the scene at a variety of novel viewpoints.

Color plate Figure 4.7 shows novel views at different moments of time of a moving head sequence. The sequence was captured by two opposing Z-Cams and two V-Cams, which provided the texture information on the sides (see Figure 4.4).

After these sequences were captured they underwent an optional off-line cleaning process. This involves removing geometric noise and filling small holes. Although not essential, this greatly improves the quality of the results. The cleaning process can be optimized to run in real-time so it can be incorporated into the online system that we intend to build in the future.
4.8 Conclusion and Discussion

We have described a real-time algorithm for quality novel view synthesis using the new generation of video cameras – those which supply also a depth value per pixel. This is achieved using the depth hull (DH) concept, an extension of the visual hull (VH) concept used in the past on video footage.

We demonstrated our results on real-world data from two commercially-available Z-Cams and several V-Cams. We captured sequences of dynamic objects and replayed these sequences from arbitrary viewpoints, which could be changed interactively during the playback.

Our approach supports the mixing of Z-Cams with V-Cams. V-Cams provide less information, but are much cheaper. Using large numbers of cameras increases the amount of processing, and also complicates the camera calibration procedure. An interesting open question is to determine the optimal number and positions of V-Cams and Z-Cams for a given scene. It seems that a single powerful computer can handle up to six cameras. This already gives quite a good coverage, but, if needed,
Figure 4.6: Rendering of a hand sequence from six different novel viewpoints at different points of time. The two Z-Cams were used only for geometry acquisition, while two V-Cams were used for texture acquisition.
Figure 4.7: Dynamic head sequence at different points of time and from different novel viewpoints. (Top) simple textured rendering of the captured depth information. (Bottom) the result of our 3-pass projective DH rendering which compensates for missing geometry.

the system may be scaled by employing a distributed rendering system, as in [43].

So far, the only Z-Cams accurate enough to support our application that we have encountered are based on structured light technology. Unfortunately, this method of geometry acquisition has several inherent shortcomings that affect the quality of the 3D reconstruction. Structured light requires a pattern projector and a video camera,
CHAPTER 4. FREE-VIEWPOINT VIDEO FROM DEPTH CAMERAS

which cannot reside on a common optical axis. Due to occlusions and self-occlusions there are typically areas that are visible to the camera but not illuminated by the projector. This leads to incomplete coverage of the object even from the camera viewpoint.

Another problem is that multiple such Z-Cams cannot operate simultaneously without their structured patterns interfering with each other. Beyond the serial mode of operation that this implies, multiple frames required for the acquisition of a single geometry frame prolong exposure times, thus significantly reducing the effective acquisition frame rate. We currently use two Z-cams commercially available from Vialux GmbH, which provide only the equivalent of 1 fps, but expect delivery of a system capable of delivering 10 fps within a few months.
Chapter 5

Reduced Depth and Visual Hulls of Complex Scenes

Depth and visual hulls are useful for quick reconstruction and rendering of 3D object appearance from a number of its reference views. However, for many scenes, especially multi-object, these techniques may produce significant artifacts known as phantom geometry. In depth hulls the phantom geometry appears behind the scene objects in the regions that are occluded from all the reference views. In visual hulls the phantom geometry may also appear in front of the objects because there is not enough information to unambiguously imply the object positions.

In this work we identify which parts of the depth and visual hull might constitute phantom geometry. We define the notion of reduced depth hull and reduced visual hull as the parts of the corresponding hull that are phantom-free. We analyze the role of the depth information in identification of the phantom geometry. Based on this, we provide an algorithm for rendering the reduced depth hull at interactive frame-rates and suggest an approach for rendering the reduced visual hull.
5.1 Introduction

Depth hull approximation is a technique for reconstruction of the appearance of a 3D object from its reference views, where each view provides a color image and a depth (range) image of the object. Although it has been used in the context of 3D reconstruction, it seems that its more attractive feature is the ability to quickly approximate the geometry of an object from a small number of reference depth images. This makes it suitable for applications such as free-viewpoint video, where a small number of depth cameras are used to capture a dynamic scene and then reproduce its appearance, typically from a novel viewpoint, for remote viewers.

The depth hull can be viewed as a successor technique to the traditional *visual hull*, which uses reference images without depth data. Geometric information about the object is obtained from its silhouettes. It is a much less powerful reconstruction
method than the depth hull, but it is widely used in many applications because it does not require any depth sensing equipment – its input is just images of the object.

There are many other methods for reconstruction of the geometry or directly the appearance of an object from a novel viewpoint. Many of them, e.g. stereo and photo-consistency, concentrate on recovering the visible geometry of the object. If the final goal is rendering, it is usually unnecessary to integrate all the available pieces of geometry into a single mesh. However, it is generally impossible to completely cover the surface of an object with a small number of cameras. Simply rendering the available geometry either using some kind of triangulated mesh, point splats or by any other technique which does not attempt to fill in missing parts will produce an incomplete image, like in Figure 5.1a. In this example the object was captured by two depth cameras, one in front and one from behind. The depth images were triangulated and rendered as textured triangle meshes. In order to get a complete image of the object, we would need to use some kind of off-line reconstruction algorithm, which is able to fill missing parts. This, however, would not suit applications where the reconstruction and rendering needs to be done on-the-fly at rates of many frames per second. Hull-based techniques are able to do just that. Figure 5.1b shows the depth hull rendering of the same object as in (a). Actually, some of the off-line reconstruction algorithms are, in fact, based on the depth hull [18, 50] or the visual hull [32] principles, but they are too slow for real-time rendering applications because they try to achieve more than mere object appearance.

Despite its promise, the application of the depth hull (DH) and the visual hull (VH) to a complex scene is problematic, especially when the number of reference views is small. DH and VH may produce geometric artifacts, otherwise known as phantom geometry, as in Figure 5.1c.
A complex scene is one which consists of an object with complex geometry or of many, even simple, objects. More precisely, we consider a scene to be complex if there exists a plane whose intersection with the scene consists of more than one connected region.

The depth hull of a complex scene may consist of several connected components, some of which are connected to the depth maps of the reference views. These correspond to the objects that are seen by at least one of the cameras. The parts not connected to any depth map represent the regions of space occluded from all the cameras. Potentially, these regions can contain scene objects. However, it is very common that these regions are, in fact, empty. In these cases, severe artifacts are often produced where the resulting approximating geometry bears no resemblance to the true geometry of the occluded objects and the only relation between them is that the approximation encloses the objects. Following this, from now on we assume that we are interested only in the portion of the DH connected to any of the depth maps and would like to eliminate the remainder. In this paper we present an algorithm for rendering this.

In the DH case the phantom geometry can only appear behind the true objects because the depth maps provide us with the knowledge that the space in front of the objects is empty. This information is unavailable in the VH, so the phantom geometry may also appear in front of the objects. This poses an ambiguity problem which requires additional information to resolve. In this work we consider the use of object correspondence information for resolving the VH ambiguity during rendering. We also show that this is equivalent to using depth information.
5.2 Related Work

The majority of the state of the art is devoted to use of the visual hull. Starting with the introduction of this concept by Laurentini [34], the visual hull became popular in the context of 3D reconstruction by space carving [32]. It was subsequently used for novel view generation from reference images [44, 58]. Later Matusik et al. [43] described how to construct polyhedral visual hulls from reference images, which allowed for some geometric processing and hardware-accelerated rendering. If geometric processing is not required and the main use of the visual hull is for novel view synthesis, explicit reconstruction can be avoided and hardware-accelerated methods for VH rendering directly from the reference images can be used [36–38].

The first use of the depth hull, although not under this name, was by Curless and Levoy [18]. They used a volumetric approach to reconstruct a depth hull from reference range images. More recently, the depth hull was used for novel view synthesis in a free-viewpoint video setup [11].

There are many other works that propose different approaches to novel view rendering. Some of them are purely image-based, like Light-Field rendering [35]. Others use a mixture of image-based and geometry-based techniques, like the Lumigraph [22] and ULR [14]. There are geometric approaches that, similar to ours, use some kind of geometric approximation of the scene. They differ from us in that they spend a lot of effort on recovering partial scene geometry. Different techniques are used for that purpose, e.g., stereo [16, 29], volume carving [32], photo-consistency, like the voxel coloring [55] and its successor photo-hull [39, 61]. In contrast, we assume that partial geometric data is readily available from the depth cameras [3, 5, 6] that we use to capture the scene. These cameras are increasingly fast and allow for much more flexible
setups, like live video. They allow us to create a more complete reconstruction of the scene by using the depth hull. Although this is conceptually similar to the visual hull, DH provides a much more accurate approximation of the scene from much fewer reference views. For example, voxel coloring and photo-consistency depend on large overlaps in multiple views. This leads to a large number or closely spaced views if good coverage of the scene is needed.

To the best of our knowledge, so far there are no published works that attempt to solve the problem of phantom geometry neither in the VH or in the DH. Wexler and Chellappa [58] mentioned the existence of the problem, but suggested no solution. For the most part the issue was evaded by working with objects and scenes that are simple enough for the available number of reference views to cover sufficiently. This, however, is not an entirely realistic scenario, because it is very difficult and sometimes impossible to increase the number of reference views as the scene becomes more complex. This is especially true in a live free-viewpoint video setup, where the scene is dynamic and it is desirable to make as few assumptions about it as possible.

5.3 Definitions

Let scene $S$ be the set of all points belonging to the objects in the scene. $O_i$ denotes the center of projection of the $i$th camera, $C = \{O_i\}$ is the camera set. We define the depth map $D_i$ of the $i$th camera as the set of all points of $S$ visible to camera $i$. The projection $P_i(P)$ of a point $P$ on the depth map of camera $i$ is the intersection of $D_i$ with the ray $L(O_i, P)$ starting at $O_i$ and passing through $P$. Note that not all points in space have a projection on $D_i$, but only those whose corresponding rays intersect with $D_i$. Let $L_i(P)$ denote the line segment $(P_i(P), P)$. See Figure 5.2 for
5.3. DEFINITIONS

Figure 5.2: The depth map of a view and projection of a point on it.

an illustration of $D_i$, $\mathcal{P}_i(P)$ and $\mathcal{L}_i(P)$.

The umbra $U_i$ of $D_i$ is the set of all points in space that lie at least as far as their projections on $D_i$, Figure 5.3. The umbra of a view is the space that is occluded in this view by the objects of the scene. Since the depth map is exactly the set of all visible points it can be considered as the occluder in this view. All the points that lie behind the depth map belong to the umbra.

The depth hull $DH(S, C)$ of the scene $S$ and the camera set $C$ is the intersection of all the umbrae $U_i$. The depth hull (Figure 5.4) is a conservative approximation which bounds the scene geometry. Since the umbra of each view encloses all the objects of the scene, the intersection of the umbrae also encloses all the objects. The more umbrae participate in the intersection the smaller the intersection is and hence the tighter the bounding approximation. $DH(S, C)$ is the smallest volume that can be obtained from the depth maps of $C$, yet guaranteed to contain all of the geometry of $S$. 
A point $P$ is said to be straight line connected to $D_i$ if $\mathcal{L}_i(P) \subseteq DH(S, C)$. The second depth layer $D_i^2$ of the DH relative to the $i$th camera is the set of all farthest points that are straight line connected to $D_i$.

Consider a ray from $O_i$ into the scene. First it hits the closest object at a point
which belongs to the depth map $D_i$, which is the first depth layer. After traveling inside the object, the ray exits through some exit point on the second depth layer. After that, the ray may enter and exit objects in the scene creating more entry and exit points, belonging to deeper depth layers. The set of the first exit points for all rays from $O_i$ defines the second depth layer $D^2_i$. More detailed explanation of depth layers and algorithms for their construction was given by Guha et al. [24].

We define the reduced depth hull of the scene $S$ and the camera set $C$ as the set of all points of the $DH(S, C)$ that are straight line connected to $\{D_i\}$.

Note that there are no points in the DH that are connected to $\{D_i\}$ but are not straight line connected to it. The DH consists of connected components of two types: those that are connected to $\{D_i\}$ and those that are not. The depth map connected components are guaranteed to contain valid scene geometry. The other components, however, are not guaranteed; in fact, they are often empty of any geometry of $S$ and constitute large artifacts.

\section{RDH rendering algorithm}

The RDH is a subset of the DH. The same relationship holds for their boundaries, which is what rendering algorithms actually compute. Rendering of the boundary of an RDH can be done by considering all points of the corresponding DH boundary and using only those that belong to the RDH. By definition, for a point $P$ to belong to the RDH it has to be straight line connected to $\{D_i\}$. This means that there must exist a view $i$ for which $P$ lies between the first and the seconds depth layers. This is what is checked by our RDH rendering algorithm. The algorithm is specifically designed to utilize many of the capabilities of modern graphics hardware. It allows
accelerated RDH rendering without explicit reconstruction of its geometry.

**Algorithm 5.1**: Reduced Depth Hull rendering algorithm.

In the first stage of the algorithm the second depth layers of the reference views are constructed. This is performed by depth peeling [24]. The first depth layers are readily available as the depth maps of the reference views. Since only the second depth layers are needed, it is necessary to perform only one iteration of the depth peeling procedure for each reference view.

In the second stage the DH is rendered while discarding the pieces that do not belong to the RDH. Any of the existing DH rendering algorithms [11] can be used, as
long as they satisfy the following condition: At the rasterization step 3D coordinates of the fragments are available. Step 2.1 ensures that the rendered fragment lies between the first and the second depth layer of some reference view. This is done in the same way as projective shadow mapping works. The fragment’s 3D coordinates are first transformed into the coordinate frame of the reference view, after which its depth is compared to the two depth layers.

Step 2.1.1 checks the fragment against the first depth layers of all reference views. If there is a reference view where the fragment is closer than the depth map, it does not belong to the DH, so it is discarded. Step 2.1.2 checks against the second depth layers. If the fragment is closer than (or lies on) the second depth layer of at least one reference view, it is enough to conclude that it belongs to the RDH. Exactly one action, either discard or accept, is performed on a fragment, after which the processing proceeds to the next fragment.

For each reference view, fragments of the DH polygons originating from that view lie on the boundaries of the view’s depth map. This may cause inconsistencies during the depth comparison against the layers of the same view and lead to rendering artifacts. To avoid this, it is advisable to dilate the depth layers to extend them a little.

5.5 The VH case

The VH is constructed using less information than is available in the DH case. Thus, the reduced visual hull, or the RVH, might not be unique, as illustrated in Figure 5.5. Here the scene consists of two disjoint objects, thus each view sees two unconnected silhouettes. The VH consists of the union of the two blue regions (containing the
true geometry) and two red phantom regions. The RVH would be either the two blue regions or the two red regions, depending on how the two silhouettes in each image are paired. Interestingly, in this situation, if the two objects have the same color the Voxel Coloring algorithm [55] would incorrectly reconstruct the two red regions, because it is closer to the volume containing the cameras.

To resolve possible ambiguities in the RVH, additional information is required. A possible source of additional information is the correspondence between the silhouettes, which may, for example, be obtained from the colors of objects.

Algorithm 5.2 presents an outline of RVH rendering that takes advantage of the object correspondence, which is specified by labeling each object image (silhouette) with a unique ID.

The algorithm checks for inclusion only between a fragment and a silhouette that both belong to the same object. Consider two reference views $O_1$ and $O_2$ in Figure 5.6. $I_1(A)$ and $I_1(B)$ are, respectively, the silhouettes of objects $A$ and $B$ in the reference

\[\text{Figure 5.5: Visual hull ambiguity. The actual objects may be contained either in the green area of the VH or in the red.}\]
5.5. THE VH CASE

RVH

foreach camera \( i \)

foreach object \( k \)

foreach polygon

foreach fragment \( f \)

foreach camera \( j \neq i \)

if \( T_j(f).xy \notin I_j(k) \)

discard \( f \)

accept \( f \)

\( I_j(k) \) – image of the object \( k \) in the reference view \( I_j \).

**Algorithm 5.2**: Reduced Visual Hull rendering algorithm.

![Reduced Visual Hull](image)

**Figure 5.6**: Reduced Visual Hull rendering using correspondence information.

images \( I_1 \) and \( I_2 \). When rendering the face that is created by the bottom-left edge of \( I_2(A) \) the correspondence information is used to test the points of that face for inclusion only in \( I_1(A) \). For example, the point \( P \) of the hull will be accepted because \( T_1(P).xy \) lies in \( I_1(A) \). This is not true for \( Q \), so it will be discarded.

An example of RVH rendering can be seen in Figure 5.10.
CHAPTER 5. REDUCED DEPTH AND VISUAL HULLS OF COMPLEX SCENES

5.6 Depth and correspondence

Our main observation in the previous section was that information akin to depth could be obtained from correspondence information. This is hardly surprising, as it is well known that correspondence information between pixels in images taken from similar viewpoints can lead to depth information at that pixel, using stereo techniques.

But the opposite is also true. Sparse depth information can lead to correspondence information, see Figure 5.7, which is useful in the rendering of the RVH, as described in the previous section, even if the depth information has low quality and resolution and even has partial occlusions. The only condition is that at least one depth value is available for each separate object in the scene. For this purpose a very small, low-resolution and thus low-cost, 3D sensor can be used, e.g. that of Canesta [3].

Furthermore, in a mixed setup where some of the views come from regular cameras and some have depth information, it is possible to use both types of information to compute a reduced hull, which is somewhere inbetween the RVH and the RVH. In this case, explicit correspondence information is not required between the video views. It suffices that the depth camera ”sees” all the objects in the scene. The rendering algorithm for this hybrid reduced hull is described in Algorithm 5.3. Figure 5.10 shows such a hull compared to a RVH and RDH.

5.7 Occlusions

When viewing a scene from a reference view, the objects in the scene may occlude each other in many ways. Here we elaborate on the output of our RDH algorithm in the different scenarios. The most straightforward case (Figure 5.8 (a)) is when the object is fully occluded in all reference views. This is treated both by RDH and RVH
5.7. OCCLUSIONS

HybridRDH

1. foreach camera $i$
   Construct second depth layer $D_i^2$

2. foreach camera $i$
   foreach silhouette polygon
     foreach fragment $f$
     foreach video camera $j \neq i$
     if $T_j(f).xy \notin \Omega(I_j)$
        discard $f$

     foreach depth camera $j \neq i$
     if $T_j(f).z < D_j(T_j(f).xy)$
        discard $f$

     foreach depth camera $j \neq i$
     if $T_j(f).z \leq D_j^2(T_j(f).xy)$
        accept $f$
     discard $f$

$\Omega(I_j)$ – the silhouette in the reference image $I_j$.

Algorithm 5.3: Hybrid RDH rendering algorithm.

as if there were no object at all. Indeed, from the reference views it is impossible
to tell whether the completely occluded space contains any objects or not. Another
simple case is when the object is occluded in some reference views, but in at least
one depth view it is completely visible. This case is handled without any problem by
both RDH and HybridRDH algorithms. For RVH the object must be seen in at least
two reference views.

A less obvious case of occlusion is when the object is partially visible in just one
reference view (Figure 5.8 (b)). In this case, the RDH will reconstruct only the part
of the object that corresponds to its depth layers that can be computed from this
reference view. In the same scenario, the RVH will still not be able to reconstruct
any part of the occluded object (Figure 5.8 (d)).
Figure 5.7: Computing correspondences from a depth view. Projections of the same 3D point \( P \) in different images fall into corresponding silhouettes. \( O_3 \) is the center of projection of the depth camera.

The last occlusion scenario is when the object is partially visible in several reference views. For the RDH, it is a combination of several one-view partial occlusion cases (Figure 5.8 (c)). RVH will be able to reconstruct only those parts of the object that are seen by at least two reference views (Figure 5.8 (e)).

5.8 Experimental Results

We implemented the RDH rendering algorithm as part of our depth-camera based free-viewpoint video system. It is written in C++ with OpenGL; the GPU part is written in Cg. All buffer operations like transformations, depth peeling and morphological operations on the depth maps are performed in the GPU. The RVH and the hybrid algorithms were tested in semi-automatic fashion, meaning that the different parts of the algorithms were not integrated into a single framework, rather executed separately.
Figure 5.8: Occlusion scenarios in two reference views of a scene of 3 objects. One reference view observes the scene from the bottom and the second reference view from the right. Two (large) objects are always visible, and the visibility of the third (smaller) object varies between scenarios. The light green region is the RDH; dark green – the part of DH which is not RDH. Analogously, the light blue region is the RVH and the dark blue – the part of VH which is not RVH. (a) (small) object is occluded in all views. (b) and (d) – object is partially visible in only one view of RDH and RVH, respectively. (c) and (e) – object is partially visible in more than one view of RDH and RVH, respectively.
### Table 5.1: Rendering performance of DH vs. RDH of different test scenes. Measured on a system with: GPU - NVidia GeForce Go 6800, CPU - Intel Pentium M 1.6 GHz, RAM - 1 GB.

<table>
<thead>
<tr>
<th>Scene</th>
<th># objects</th>
<th># views</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Men</td>
<td>2</td>
<td>2</td>
<td>55</td>
</tr>
<tr>
<td>Hand</td>
<td>5 (fingers)</td>
<td>2</td>
<td>55</td>
</tr>
<tr>
<td>Paper prisms - circle</td>
<td>3</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>Paper prisms - row</td>
<td>3</td>
<td>3</td>
<td>25</td>
</tr>
</tbody>
</table>

Surprisingly, in spite of the computational overhead that comes from the preparation of the second depth layers, the RDH actually outperforms the DH, as can be seen in Table 5.1. We believe this is because the RDH eventually produces much less fragments and thus reduces the total rendering load.

The results of applying the RDH algorithm to different scenes can be seen in Figures 5.9–5.12. The reference views in Figure 5.10 were produced by a simulated depth-camera, the scene was composed of two polygonal models, constructed from 3D scans of real people, publicly available on InSpeck’s web-site [5]. The reference views in Figures 5.9, 5.11, 5.12 were obtained using Z-Snapper depth-cameras from Vialux [6].

### 5.9 Conclusions

We have presented rendering algorithms for phantom-free depth and visual hulls. In our experiments using the RDH algorithm led to improved rendering quality and performance. We also showed that a small amount of complementary depth information suffices to eliminate phantom geometry from the VH. This can be used to improve existing multi-view VH systems, especially real-time systems, by supplementing the
5.9. CONCLUSIONS

video cameras with a small low-cost depth-sensing component.
Figure 5.9: Hand example. (a,b) – two reference video views and their corresponding (c,d) depth images. Notice the "phantom" fingers produced by the undesired part of the DH.
5.9. CONCLUSIONS

Figure 5.10: Men example. (a) and (c) – two reference views; (b) and (d) – the depth images of the reference views. Notice that the undesired parts of the DH and the VH use textures from different objects. For the hybrid hull only the depth information of the first reference view was used. Still, this was enough to remove the extra geometry. Notice that the left sides of both objects are more flat, like in the RVH, whereas the right sides are like in the RDH.
Figure 5.11: Three paper prisms arranged in a circle. Top row – three reference views. Bottom row – Depth Hull and Reduced Depth Hull.
Figure 5.12: Three paper prisms arranged in a row. Top row – three reference views. Bottom row – Depth Hull and Reduced Depth Hull.
References


REFERENCES


עיבוד מידע ביער מתודת ערב לפני לشم
יצירות מחקריות וخلافיות מודעות של צמח

אלכסנדר בוגומילקוב
עיבוד נתונים ברצועת המעבד גליפי לשם
יזירת מבנים ודמויות מיריאו של זומק

חיבור על מחקרים

לש מולי חלקי של הדרישות לקבליות התוכנה
ודקטרفلוסופית

אלכסנדר ברגמא产业基地

גורש לעסק התוכנות – מוכן טכנולוגיה לישראל
2008

חיפה

עברית תשס”ח
הברת תודה

ברצוני להודות פרופ' חינו גסטמן על הנингה המצלמה, הטרيقة והעידה
בכל שכליו המחקך.

אני מצה לטעמי על התמינה הכסיית הנדיבות בשหลายๆותיניות.
# חוכמ בּגַנִים

<table>
<thead>
<tr>
<th>מספר</th>
<th>תכנית</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>מבוא</td>
</tr>
<tr>
<td>3</td>
<td>מסקנה 1</td>
</tr>
<tr>
<td>10</td>
<td>העבירה 1.1</td>
</tr>
<tr>
<td>13</td>
<td>רקע 2</td>
</tr>
<tr>
<td>13</td>
<td>כונן העבירה הפריפי 2.1</td>
</tr>
<tr>
<td>16</td>
<td>הפ.paginator 벤EFI equipo 2.2</td>
</tr>
<tr>
<td>17</td>
<td>מיפוי טקסטוריו פרימティブי 2.3</td>
</tr>
<tr>
<td>18</td>
<td>מיפוי אל lié פארמאטיבי 2.4</td>
</tr>
<tr>
<td>20</td>
<td>כלל שבב העבירה 2.5</td>
</tr>
<tr>
<td>22</td>
<td>פишוט מפת שֶפֶכָד בערת המכבי הגרפ 3</td>
</tr>
<tr>
<td>23</td>
<td>מבוא 3.1</td>
</tr>
<tr>
<td>24</td>
<td>עבידות קדמואות 3.2</td>
</tr>
<tr>
<td>26</td>
<td>פישוט מפת עמק 3.3</td>
</tr>
<tr>
<td>28</td>
<td>האלגוריתמים המיטופרים 3.4</td>
</tr>
<tr>
<td>30</td>
<td>פישוט עם חומרים הגרפים 3.5</td>
</tr>
<tr>
<td>30</td>
<td>kiện ושקדקטובים בقيقة 3.5.1</td>
</tr>
</tbody>
</table>
הוכן טכいます

31  מגרריך חישוב 3.5.2
36  בנין רשת 3.5.3
36  תחאורת 3.6
38  דיני 3.7

ריאור תמך חוסש מחזיר טעמים עמק
40  מבוא 4.1
44  עבדות קדורת 4.2
46  תרומת 4.3
47  קליפת ויוארת 4.4
47  הליהור 4.4.1
48  הפתקה קליפת ויוארת 4.4.2
51  הפתקה פורימקטיית משפרת 4.4.3
53  קליפת השמכ 4.5
54  הפתקה קליפת עמק 4.6
56  תוראה נהגית 4.7
56  סדרת אנרגמות 4.7.1
58  ביצועים שונים 4.7.2
59  מדיקט והיד 4.8

קליפת טכני קליפת ויוארת מצרימות של טכנולוגיה ורבים
63  מבוא 5.1
67  עבדות קדורת 5.2
68  הליהור 5.3
71  אלגוריתמים הפתק של קליפת שמק מ сразמות 5.4
73  הפתקה של קליפת ויוארת 5.5
ודגנים

76  .................................................. עוקב והודאות 5.6
76  .................................................. הסתרות 5.7
78  .................................................. ת:noneת 5.8
80  .................................................. מסכיות 5.9

56  רשימה מקורות

1  תכשיר
הוכן שנים
רשימת אрафים

14. אטרים נוספיםajuICE2.1
17. הפ stadכי גמרセットרזעמעברפ.2.2
18. מיפור טקסטורחרת פרימריים.2.3
19. מיפורǼילה פרימריים.2.4
20. כלים שעביםת העמק.2.5

23. דוגמאות של מוף גובה עומק.3.1
26. שילוח מוף עומק.3.2
26. תסריט קדקודים.3.3
28. חסום הרה-כ.3.4
29. רמות של קדקודים.3.5
32. אاحتجות הקדקודים ב zamówienia.3.6
34. מעבר חישוב 1-4.3.7
35. מעבר חישוב 5.3.8
39. דוגמאות פיפוטים של מופ גובה.3.9
39. דוגמאות של מופ גובה מפוסטתר.3.10
42. דוגמאות של הפסקת מבט חсим.4.1
48. השוואת קפítulo בד-עומד.4.2
רשימה אידים

<table>
<thead>
<tr>
<th>מספר</th>
<th>תítulo</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>השוואת אידות הפכה</td>
</tr>
<tr>
<td>57</td>
<td>סידור המצלמות</td>
</tr>
<tr>
<td>59</td>
<td>השוואת אלגוריתמים הפכה של קליפת שעמק</td>
</tr>
<tr>
<td>60</td>
<td>רץ של די.</td>
</tr>
<tr>
<td>61</td>
<td>רץ של ראשים</td>
</tr>
<tr>
<td>64</td>
<td>דוגמה הפכה של קליפת שעמק</td>
</tr>
<tr>
<td>69</td>
<td>הנדרה של מפת שעמק</td>
</tr>
<tr>
<td>70</td>
<td>מותח מנצל של מבנים</td>
</tr>
<tr>
<td>70</td>
<td>קליפת שעמק ויכולת של העמק הטובה ביותר</td>
</tr>
<tr>
<td>74</td>
<td>דמיון מאגנטס בклиיפה ויוזראלי</td>
</tr>
<tr>
<td>75</td>
<td>הפקר קליפת ויוזראלי המתבצעת том' שצומח במדע על ותואמת</td>
</tr>
<tr>
<td>78</td>
<td>חישוב האמצע המבוסס על שעמק</td>
</tr>
<tr>
<td>79</td>
<td>התיקרות</td>
</tr>
<tr>
<td>82</td>
<td>דוגמה קליפת שעמק ומוצמדませ עם די.</td>
</tr>
<tr>
<td>83</td>
<td>דוגמה של גז אריזה קליפת שעמק מוסוגת שעימה</td>
</tr>
<tr>
<td>84</td>
<td>מסרורת ניר במעגל</td>
</tr>
<tr>
<td>85</td>
<td>מסרורת ניר בשורה</td>
</tr>
</tbody>
</table>
תקציר

בחזרה לקצין הפាវס של מבטיס דודים, יזם יערית הtrzymałות אוצר מחוזות מיזוגים שנות לאלה שאלים. הקטן הصيدית ההומגנה ייצוג אוחז ואיתו 얘기 אצליום מביאים תמציתו המבוססת בין שחת.fv

יוואל מספר הפילו הידיעה הלאור בוחса בחרס הידיעה הידיעה בחרס המצאית שנבכרע על ידי המדסה ו"ויאן אינטראקציה" הוא יער STREAM. מסלמה ורטואלארט אוצר שולחנת על ידי

המדסה בק.

ישנא מספר "חושם" אחר עיסוי לחרבס על היכל והשל הפקת הידיעה המבוססת דודים. דודים, בלשון יcallee זה מלה tritur הידיעה המבוססת,DODS.

יכצל ושוחח עם יcallee יcallee בחררס מבהיסGIT בחרס הידיעה המבוססת שיתפויות. ב Yahuda וידיעה אוזן שופ החולה Calvin Compact שפל שפל לשנה המחנה הידיעה."חימה ישראלית" המ谲פת המחנה מתכנתת מתכנתת דודים.

בטסגורת עבודה, האנג ניסקינד בודים בברט שדרור, הידיעה שיפלת ו"מכפרות" שיפל פורטינ. האט יcallee הרבח מראות הידיעה מחזק הפקת בטיסים דודים. ליין, שליביAEA קשת י öde, שליבי הידיעה, מזרעים פעול ההיפוגה, הידיעה להביעה במראה בהובך הבנק בוי שלדישות המתוגות הגדול. שיסת מרססות המחנה מתכנתת מתכנתת תכליתית, די ישיוות מחומת הידיעה. המחנה הפקת בחרס לכל, מחומת המחנה מחומת בחררס. די קול האת המחנה מחומת. בפייקס בחררס הפלס, קול האת מחזק המחנה המחנה מחומת

מחומת מחומת הפקת.
השיטות האלקטומטריות לשחזור מראה של כמות של アイオメトリיה שלש קושי

מדרשון, שיתוף פעולה. בדרכי זה, כדי לה PriorityQueue את תוצאות ה-דיסקית שלכתי, ניתן במאית
שמירת את הנוסחהembreיטים, בשני הקדימה משוחかつ מובססו תוצאות ושורת
משפרים על ההמומנה כלים. עם זאת, במדのではない התוכנה של דירקשון וב/dis
aryawan פס הרגיש, עלי קוך ל-ט מftime את הדיסקית שלחר

שיטות מבוססו על-ארטופורייה מש بتاريخ קורות ל-ארטופורייה של הר妥协סית בשתי

ופשיטוט נגזר מהחלקה, ובאשר זה מהווה מתוכן וארטופורייה שמתחא את התוקה/ה יוץ התוקה.

בשיטות א-ארטופורייה שלב התוקה עלי בוחר, כי הוא יכול לצל את התוקה/ה הזניחית.

אולא יש קיוסים מחודד של מראה ת合一 מובט על פי קוך על המ susceptים היגי"ם מכבר.

טיטס

קושי תוכן, שיום בחוש הלולע בכיסוי אוארטופורייה של הסצנת, העו קוך תועד במלילי

וא בЄסיו התוקה/ה אנא לא נמשר על ד"וי המצלמות. כל מים, יש ספין מ קיוס תודת

בכיסול המדידה או השחור של א-ארטופורייה המצלמות. חף כל הדיבר ה-ארטומתים.

וטו ה-ארטופורייה שישה ייסול רמה שב לוסק את ה-העטיפת הדירקסי שלverity על במנה Außen.

ההפלס ב תומכת על י למספר המצלמות ב גאפת, בתכתס שהזוהה. מספר המצל-

موت עשיית החשוניות בכヨרי הצלד. מ-3-2 עד 80 יונים. שיווק ממסכות תומנה, למסק.

דוירשת מספר רב ש섯-צלמות. בתיגה תמונת של הסצנת, הרובו אתלורתיה ני-ארטופה.

יו מתחילה בשחור של א-ארטופורייה של הסצנת. ד"ו ההודורס תמא מחיוב בר

החלף כל והטוען באיתנו ממדידה אחר ייסול לשספור-ארטופורייה להזינו שיש.

סֹּרִיך לייזר. יד הע גאות, סורק לייזר ד"ו ולא מספיס מהדרים ד"וי לוסק קרבן ודרים.

לֹּפִיכָס, הפתרון הוא שيوم במעל המצלמות עמק, והי ה-טולפוגה ייסית תודש, שמייפר

למסגטים וידיאו למסק בנסף תמונת עמק על הסצנת. קאיו הוא מתכנת_parallelו

הלוקה תמיימו בלוקה מסגרת שיאת מבוקד אחד של המצלמות.بقל רוג מתקבלה

הเสมอ זב, שיום בלוט ספק시키ות, התמונת עמק, שמייפרושו גיאופורית. בורב וה-ש

יוסי. האבע הלוחם נרמישו לי יאנה סכוסר א לhaitו לי"סנס呒י מושיכים על ואמן

Technion - Computer Science Department - Ph.D. Thesis PHD-2008-01 - 2008
התקיר

צי אופט, תלטсты בא記錄 השני ונוחה מד והיא משלמה. קיימים משלמות שמשתמשים
ולוח לכסף ועד 60 במונה שוק עבש על שקית. Hóaית ביוור ו המקומ המקובלים
עלظم התחרות של יצר אינפרה-או. אולס, מביתו הרוך עדזמ. אל פעות בשתי.
 gslות השלמות של האינטגרציה בין יתר המשמשות בישוע ואור מרב הזורות
ובברב מקא יח. והируו שלשלות числе בעבר.

החקיפה ה.),ישולית של האביהוות היא תۂ וזרה בינוני על אפקטל בול. כמ. הראו.

יקי עטב. בונה זה הנשה שיםש בר, בשוחרת תהליך-מידי, לי, סימק סירטס על צל.
לילות. קיימים אולריצולים פרמי לברית קDeltaTimeי והጽור, מחוק דרומת ייחס
ולמי.nier. שעון ישונת. שעון רונר, והחקיפה ה.),ישולית לפואנדית ה.),ישולית
שוחרר מבית של אינפיטרי. שיחעים העולם מיתור בטורטת מסמך לצליבה
.psiיון בעוות של והידס. והזרה ה.),ישולית של משלמות ה),ה וכס קפוק ו Stück.
צורה בחרות בך, ה),ישולית בשתייה. חוקיפיה ה.),ישולית של
אותונט בסיס של דר הלטס פרמיים קיימים ה),ישולית מסמך לצליבה
מאית וידאון וול.

תקיפה ה.),ישולית מאפרחת צורה עכ ונומדכל ל-,ימונטיר האפידונית של האוריונט.
םשר ורותר למספר גול שיים כת ק),סדי ליגוי לאיבוד המונה סטיב. בנסו תקיפה
ויואליות אבוף ניאורטלי למסגורת לחות מריאה לגבי ממסים. תודים פיתואים
שהמסירות ק),סדי. ה),סד בארו וה),דא קלפיים של קלפיות מהפיכת
במסגרת עבוך. וה),רותרת את המשמיעת התקיפה ה.),ישולית על תקיפה
ה),סדים. בן, שברב קומת האליטה וזו העשם הדומית השלמה והדומית הנפגעים
ולמי. אנחת טיאטס צרכו כדי.Small, את הנעבバック ההופך נשיבת השיחוד של הפסק
מביסים, עד להוטה בום אט בדושת תמונות עים. השיאל הניה להעבגיה ממ. שיאור המים
למעון, גרושי, ע. מעט לשלב בדועי מרכז את זות ההיתור של. שומדאות באגעל,
וכם להעבバック הריאה.
תקציר

אצ ההומוכן של העבירה הגרעיה נינו לחלק ממפרת חלקי עיקרים. בחלקהวาง
של העבירה ואת מטרותיהם acquitted השם חלקי ההפיסות של מפת עומק בשירה
שימשר בהמפה הנראית. כל שיש משלמות רוב מפעלי בליפה של עריכת מבטיפת-
ים. פשיטי עלール מפת עומק ט>(), שירコミ שים כי טימEntryPointים לחシステム.
זם אפקט
פגי הקורון נימי אומן. קלש לשל סימפליטע טיסה. גצות התאומים אגוניותшим
שווהקלים העניקים של ההעלאה מבעל הראה.
בחלקהวาง הונcerer על העבירה הנוסחי בחלקי ההפיסת הגיאומטרית של קפלת
עומק הנקה השושב במעבד הגרעיה. עם מפת עומק של הונ rte ירוקי עגרת שילוח
ות בצarious👨‍💻.dep. אנד בשנייה רגע לא מוסתרת תפרה של החלקוי השניים לנה
אואד ששל. סוכקת כשל מיקום עכלים בתורה ממתה שיטה של치료ים החלקויスピיקט
העומק. החלקוי שנאמרה ההפיסת המרינה. בנסף, בחלקהวาง, מצר חזרה, בנזרתwayneמשפר החלקוי.
קלאיפס יחואלנץ ראש הגוי חיס משפרים בין אימים ומוריית ההפיסת.
חלקהวาง השולש של העבירה,apus בונהולון מרובע ייז. במכרים הכלה, הקפלת
והוריאליות הקפלת העמק. כפ שורדהו במקו, הביבלה החלקוי שבועבש איכז מני
ᠭ平常 גואלמהור של גויך קימיים בפועת. חליפה אלה גולטת משילבר של
גופים שטנים בבשנה, אלא של חליפה שטנה של ארזים, נמי מרבורב, אחר הגרות והמשי-
ונגו קפלת עמק מפמפרת קפלת יחואלנץفعاليات. שימורה ש biên מבלשיה ויוג
גופים שקבוע קימיים בשקן. גוז מימיי ארגונטימיות חיים שחליפה של הקפלת
האנטומוניות, אשר מבצעים באלפיניות המעבד הראיתי.