Real-Time High-Resolution Sparse Voxelization with Application to Image-Based Modeling

Charles Loop*
Microsoft Research

Cha Zhang†
Microsoft Research

Zhengyou Zhang‡
Microsoft Research

Figure 1: *Several views of a model, with and without texture, reconstructed from eight cameras. This surface contains 2.5 million out of* 1024^3 *voxels. Computation plus rendering took 24 milliseconds.*

Abstract

We present a system for real-time, high-resolution, sparse voxelization of an image-based surface model. Our approach consists of a coarse-to-fine voxel representation and a collection of parallel processing steps. Voxels are stored as a list of unsigned integer triples. An oracle kernel decides, for each voxel in parallel, whether to keep or cull its voxel from the list based on an image consistency criterion of its projection across cameras. After a prefix sum scan, kept voxels are subdivided and the process repeats until projected voxels are pixel size. These voxels are drawn to a render target and shaded as a weighted combination of their projections into a set of calibrated RGB images. We apply this technique to the problem of smooth visual hull reconstruction of human subjects based on a set of live image streams. We demonstrate that human upper body shapes can be reconstructed to giga voxel resolution at greater than 30 fps on modern graphics hardware.

CR Categories: I.4.8 [IMAGE PROCESSING AND COMPUTER VISION]: Scene Analysis—Surface fitting;

Keywords: image-based modeling, sparse voxelization, visual hull

*e-mail: cloop@microsoft.com
†email: chazhang@microsoft.com
‡e-mail: zhang@microsoft.com

HPG 2013, July 19 – 21, 2013, Anaheim, California.

1 Introduction

Reconstructing 3D models on-the-fly is an important problem in the context of telepresence and live free viewpoint performance capture, among others. Given calibrated multiple camera configurations, dense pixel data must be merged into a new 3D model each frame, in real-time. Voxel methods have proven useful for reconstructing such models in the off-line setting [Curless and Levoy 1996]. Recently, voxel based surface reconstruction methods have been applied in real-time applications [Zhou et al. 2011; Izadi et al. 2011]. However, these methods are not directly applicable to our problem domain, where our goal is to build and render high-resolution voxel models of dynamic scenes at interactive rates (i.e., 1024^3 at greater than 30 fps). To this end, we present a straightforward GPU based technique for rapidly reconstructing and rendering such models.

We represent voxels as index triples corresponding to the 3D spatial location of a cell within a discrete voxel lattice (grid). A voxelization is an array (list) of such voxels. We begin with an initial low resolution voxel grid and *cull* voxels in parallel that cannot belong to an object's surface defined by the image data. The remaining voxels are split into 8 sub-voxels in parallel, and tested again for surface intersection. This process repeats until the projected voxels are on the order of a pixel in size. We render the axis-aligned, screen space bounding rectangle of a voxel with shading based on the input RGB images.

Currently, the geometry of our models are based entirely on silhouettes and a *visual hull* reconstruction. Such models are not highly accurate geometrically. This results in ghosting artifacts when blending input images to create view dependent texture. Our intent with this work is to demonstrate components of a high performance real-time capture and reconstruction system using a voxel based approach, and not to create a perfect reconstruction. We believe that geometric accuracy can be improved by incorporating depth sensor data as that technology matures.

We justify using only a silhouette-based visual hull reconstruction by the following. First, current structured infrared light based depth sensing technologies suffer from inter-sensor interference resulting in errors or loss of data. This currently limits their usefulness in multi-sensor configurations. Second, depth sensors tend to fail on hair, due to IR light scattering. One possible way to overcome this difficulty is to incorporate silhouette information into the modeling process, and construct the visual hull as a conservatively correct approximation to a head with hair. Finally, we are able to benchmark a complete capture system to demonstrate the viability of a GPU-based voxel approach in a live studio setting.

This paper is organized as follows. We review related previous work in Section 2. The voxel data structure and basic algorithm we use are presented in Section 3. In Section 4 we present details of our smooth visual hull reconstruction. We give a performance analysis in Section 5, discuss future work in Section 6, and offer conclusions in Section 7.

2 Previous Work

Sparse voxelizations partition space hierarchically and eliminate sub-trees that do not contain a part of a model. A node within such hierarchies can correspond to a *brick* of size m^3. When $m = 1$, the hierarchy corresponds to an *octree*. Octrees have long been used for isosurface extraction [Wilhelms and Gelder 1992]. Recent works have utilized voxel octree methods on the GPU. Octree textures [Benson and Davis 2002; Lefebvre et al. 2005; Kniss et al. 2005] encode a voxel hierarchy in 2D texture maps by storing child pointers as texture table offsets. A fragment program can efficiently texture a mesh model with volume color data using this technique. In *gigavoxels*, bricks corresponding to hardware accelerated 3D textures with mip-map hierarchies are streamed from the CPU when needed [Crassin et al. 2009]. A ray casting algorithm, driven by hardware rasterization, is used to efficiently render massive precomputed, static voxel scenes at interactive rates. Along these lines, [Laine and Karras 2011] advance the premise that the advantages of voxel processing are beginning to challenge traditional 2D textured triangle models for representing and rendering complex, highly detailed models. They reduce tree depth by storing surface contour information in terminal nodes, and report impressive results for a ray cast renderer. However, these methods are concerned with rendering existing models, not on-the-fly surface reconstruction. Solving a somewhat different problem, efficient high-resolution solid voxelization of triangle mesh models has been performed on the GPU [Schwarz and Seidel 2010; Crassin and Green 2012] using the hardware rasterizer.

Using a particle-based passive stereo approach, [Hornung and Kobbelt 2009] developed an interactive system for free viewpoint rendering of a static scene from a collection of pre-processed images. Kinect Fusion builds a voxel model of a static scene in real-time using a single Kinect depth sensor that is moved through space [Izadi et al. 2011]. Our goal is to build a dynamic voxel model from multiple live input image streams. A similar voxel data structure can be found in[Zhou et al. 2011], but their goal is to accelerate Poisson 3D surface reconstruction from point cloud data, not on-the-fly image-based 3D surface reconstruction. To facilitate this, they maintain significant auxiliary data structures to enable fast neighborhood queries. Also similar, is the *space filling curve* linearization of a voxel octree reported in [Ajmera et al. 2008]. They build a sparse voxelization on the GPU in a bottom-up fashion; ours is top-down.

Recent telepresence systems reconstruct novel 3D views of a captured scene from several depth and RGB cameras. Using multiple Kinect sensors, [Maimone and Fuchs 2011; Kuster et al. 2011] combine several triangle meshes constructed by connecting neighboring pixels in depth maps for 3D rendering. This approach does not attempt to *fuse* these meshes into a coherent 3D model, and is therefore affected by interference between the Kinect sensors. A similar approach can be found in [Zhang et al. 2013], but instead of Kinect sensors, a novel structured IR light based stereo approach is used to overcome inter-sensor interference.

Visual hulls have been used in computer vision applications for some time [Laurentini 1994]. Real-time visual hulls have be computed efficiently using CSG operations on epipolar lines of calibrated images [Matusik et al. 2000]. More recently, GPU algorithms to carve visual hulls from a voxel grid have appeared. A hierarchical approach very similar to ours has been reported in [Ladikos et al. 2008], while a [Schick and Stiefelhagen 2009] directly cull voxels from a fine grid. While these works could be used to obtain results similar to those we describe in this paper; our scheme is more flexible in generating isosurface based models with arbitrary smoothness (see Figure 5). The ghosting artifacts resulting from the poor fit of visual hull geometry can be mitigated using optical flow estimation [Eisemann et al. 2008]. Silhouette images of human subjects have also been valuable in developing marker-less motion capture systems [Carranza et al. 2003; de Aguiar et al. 2008].

3 GPU Based Sparse Voxelization

We begin with a voxel representation and a collection of processing steps. Each processing step is implemented as a compute kernel that executes in parallel on a programmable GPU. We first present the basic voxel processing approach, and defer details that apply to the specific problem of visual hull reconstruction until Section 4.

Voxels $\mathcal{V}$ are represented as an array of unsigned integer triples $\{ix, iy, iz\}$ corresponding to the coordinates of a voxel corner within a 3D grid of dimensions $2^k \times 2^k \times 2^k$, where k is the *level* within a voxel hierarchy $\mathcal{V}^k$. Each voxel within the voxel hierarchy is fully defined by its integer coordinates and level. Note that we do not allocate space for the 2^{3k} voxels of the grid; rather, the existence of a voxel is determined by the presence of its indices within the list $\mathcal{V}^k$. The number of voxels $|\mathcal{V}^k|$ we store is proportional to the surface area of an object, not the number of voxels in the voxel grid. The maximum number of levels, i.e. the *depth*, of our hierarchy is determined by the number of bits used for each unsigned integer coordinate.

While we store the integer coordinates of a voxel internally, we convert to *normalized voxel coordinates* (NVC) via the transformation

$$nx = \sigma_k\, ix - 1, \quad ny = \sigma_k\, iy - 1, \quad nz = \sigma_k\, iz - 1$$

where $\sigma_k = 2^{1-k}$ is the scale factor associated with level k. An NVC voxel grid spans the cube $[-1, 1]^3$. We use this NVC cube as the world coordinate system, and maintain all camera and viewing transforms with respect to these coordinates. We place this cube (virtually) around the capture subject and voxelize the visible surfaces within the enclosed space.

3.1 Algorithm Overview

Figure 2 illustrates a flow diagram of our voxel processing steps. Each box corresponds to a GPU compute kernel, with the exception of **render** where the graphics pipeline is invoked. We leverage the parallelism of modern, massively multi-threaded GPUs by assigning a single lightweight thread per voxel within each compute kernel. These processing step are described below.

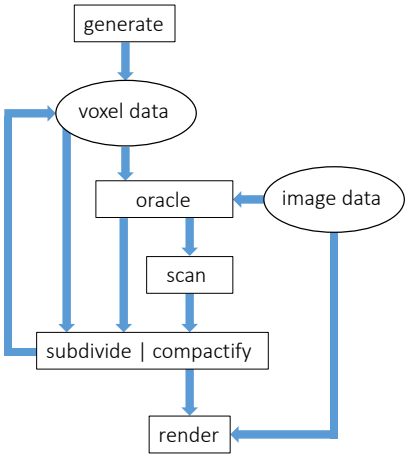

Figure 2: *Flow diagram of our voxel processing steps.*

Generate: An initial low resolution voxel grid, i.e., a list of voxels, is generated. In our implementation, we begin at level $k = 4$ and create a 16^3 grid containing 4096 voxels. In principle we could begin at level $k = 0$ with a single voxel. However, several iterations would be needed before all available cores of a GPU are fully utilized, reducing efficiency. Each thread writes unique voxel indices to a location corresponding to its thread ID. These indices are computed as a function of thread ID, as shown in the following code snippet.

```
voxel v;
v.ix = threadID/256;
v.iy = (threadID/16)%16;
v.iz = threadID%16;
VoxelList[threadID] = v;
```

Oracle: The oracle kernel is the most important and flexible processing step. Its job is to decide if a given voxel should be culled from the voxel list. The oracle will have access to a variety of image data and associated coordinate transforms that project NVC voxels to image pixels. Ignoring for the moment *how* the oracle makes a culling decision, its basic function is to write a zero (cull) or a one (keep) to a list in the position corresponding to the thread ID of a voxel. We present an oracle kernel for smooth visual hull reconstruction in Section 4.

Scan: The list of voxel culling decisions generated by the oracle is passed to a prefix sum scan kernel [Blelloch 1990]. The scan kernel maps the input list one-to-one to an output list in parallel. The value of the output list at position i, is defined to be the sum of all elements of the input list less than i (*exclusive scan*). The power of this operation is that, for each kept voxel, the scan output list provides that voxel's location within a sequential list of kept voxels. This means that culled voxels can be eliminated from the voxel list independently and in parallel. We use this in the next stage of processing. A detailed description of how parallel prefix scan is efficiently implemented on modern GPUs can be found in [Harris et al. 2007].

Subdivide: If the (user defined) maximum number of iterations has not been reached, then the subdivide kernel is invoked. The subdivide kernel splits each voxel into 8 sub-voxels, creating a new grid whose resolution has doubled in all 3 dimensions. The indices of the new sub-voxels are found by doubling the voxel indices and adding zero or one in all 8 combinations. These 8 new sub-voxel indices are written to a new voxel list at a location found as 8 times the value of the scan output list in the position correspond to the thread ID. The code snippet below provides details of the subdivide kernel.

```
if (oracleOutput[threadID] != 0)
{
    voxel vIn = VoxelListIn[threadID];
    uint baseIdx = 8*scanOutput[threadID];
    for (uint i = 0; i < 8; i++)
    {
        voxel vOut;
        vOut.ix = 2*vIn.ix + i/4;
        vOut.iy = 2*vIn.iy + (i/2)%2;
        vOut.iz = 2*vIn.iz + i%2;
        VoxelListOut[baseIdx+i] = vOut;
    }
}
```

Compactify: If the maximum number of iterations has been reached, then the compactify kernel is invoked rather than the subdivide kernel. The compactify kernel simply removes culled voxels from the voxel list. The indices of a voxel are written to a new list at the location corresponding to the value of the scan output list at the position correspond to the thread ID. The code snippet below details the compactify kernel.

```
if (oracleOutput[threadID] != 0)
{
    voxel vOut = VoxelListIn[threadID];
    uint baseIdx = scanOutput[threadID];
    VoxelListOut[baseIdx] = vOut;
}
```

Render: After a sufficient number of iterations, the output of the compactify kernel is the final list of voxels that intersect the desired surface. The projection of these voxels should be roughly the size of pixels. We render theses voxels using the graphics pipeline as splats corresponding to their axis-aligned, screen space bounding rectangles. We describe this process in more detail for our application in Section 4.4.

4 Smooth Visual Hull Reconstruction

We now describe how to voxelize a surface that corresponds to a smoothed visual hull of an object as seen by a collection of n cameras. We form this surface as the product of n filtered silhouette images. Let

$$\mathcal{I}_i \; : \; \mathbb{R}^2 \rightarrow [0,1]$$

denote the filtered silhouette image as seen from camera i, such as those shown in Figure 3. Further, let

$$\mathcal{C}_i \; : \; \mathbb{R}^3 \rightarrow \mathbb{R}^2$$

be the projective transform from NVC space to image space associated with camera i. We form the product of silhouette images

$$\mathcal{S}\left(\mathbf{p}\right) \; = \; \prod_i \mathcal{I}_i \left(\mathcal{C}_i \left(\mathbf{p}\right)\right),$$

where $\mathbf{p}$ is a point in NVC space. The surface we voxelize is

$$\mathcal{S}\left(\mathbf{p}\right) \; = \; \tfrac{1}{2}.$$

Note that the gradient $\bigtriangledown \mathcal{S}\left(\mathbf{p}\right)$ is normal to $\mathcal{S}\left(\mathbf{p}\right)$. If the silhouette images $\mathcal{I}_i$ are smooth, then $\bigtriangledown \mathcal{S}\left(\mathbf{p}\right)$ provides a smooth normal field over $\mathcal{S}\left(\mathbf{p}\right) \; = \; \frac{1}{2}$.

4.1 Silhouette Image Acquisition and Filtering

We use chroma keying to segment foreground and background pixels. Foreground pixels are assigned a value of zero; while background pixels are assigned a value of one. In our implementation, we upload raw Bayer pattern data to the GPU each frame, storing these in a 2D texture array. Since Bayer pattern data is only 8 bits per pixel, significant camera bus and CPU/GPU data bandwidth are reduced. We launch a single compute kernel to simultaneously remove radial lens distortion, demosaic, and chroma key these image data to produce calibrated silhouette images.

Next, we filter these images using a separable Gaussian kernel with unit integral. Increasing the width of this kernel increases the amount of blur in the silhouette images, resulting in a *smoother* surface $\mathcal{S}(\mathbf{p})$, see Figure 5. While separability is not necessary, it greatly increases convolution performance on the GPU since an entire image row (column) can fit within shared memory using a thread block per row (column) strategy.

4.2 Visual Hull Oracle

Our oracle kernel is assigned a single thread per voxel, and each voxel culling decision is independent of any other. The oracle has access to the n filtered silhouette images, as well as n mip pyramids used to accelerate and bound voxel/silhouette coverage queries; we discuss these in Section 4.3. In addition, the oracle knows about the n coordinate transforms from NVC space to silhouette image space, as well as the coordinate transform from NVC space to screen space for the novel view currently being generated.

The oracle must make a binary culling decision, based on whether or not a given voxel intersects the surface $\mathcal{S}(\mathbf{p}) = \frac{1}{2}$. Clearly, a voxel will intersect the surface if $\mathcal{S}(\mathbf{p}_1) > \frac{1}{2}$, and $\mathcal{S}(\mathbf{p}_2) < \frac{1}{2}$ for some points $\mathbf{p}_1$ and $\mathbf{p}_2$ within the voxel. Our strategy is to find bounds on the value of $\mathcal{S}(\mathbf{p})$, over all points $\mathbf{p}$ within the voxel. This can be done by finding bounds on the projection of a voxel into each filtered silhouette image $\mathcal{I}_i$, and taking the product of these bounds. We are able to find these bounds within images using the mip pyramids described in the next section. These are used to implement the function `sampleSilhouette(v, i)`, that will return a `float2` value containing a conservative bound on the projection of voxel `v` into image `i`. The following code snippet outlines the logic of our smooth visual hull oracle kernel.

```
DECISION = CULL;
voxel v = VoxelListIn[threadID];

if (!outsideFrustum(v))
{
    bounds = float2(1.0, 1.0);
    for (int i = 0; i < numCameras; i++)
        bounds *= sampleSilhouette(v, i);

    if (bounds.x < 0.5 && bounds.y > 0.5)
      DECISION = KEEP;
}
oracleOutput[threadID] = DECISION;
```

In the code above, `bounds.x` and `bounds.y` are the lower and upper limits of the desired bounds. Note that the product of `float2` variables is the product of their components, and the function `outsideFrustum(v)` returns `true` if a voxel `v` lies outside the viewing frustum of the novel view.

In order to accelerate processing, we run the oracle kernel in a hierarchical, coarse-to-fine manner (see Figure 2). To facilitate this, and avoid missing small features or introducing holes, we developed an acceleration structure using a mip pyramid over each silhouette image.

4.3 Hierarchical Voxelization Acceleration Structure

The acceleration structure that we use is inspired by the well-known hierarchical z-buffer [Greene et al. 1993]. At each pyramid level, two channel texels are determined by finding the minimum and maximum possible values of the 4 texels at next lower level. This will allow us to quickly find a conservative bound on $\mathcal{I}_i$ over the projection of `v`. In the function `sampleSilhouette(v, i)`, we project the 8 NVC space corners of `v` into silhouette image space. Next, we find the axis aligned bounding rectangle as `minX, minY, maxX` and `maxY`. We determine the sampling level in the silhouette image mip pyramid by

```
floor(log2(max(maxX - minX, maxY - minY)));
```

At this level, the bounding rectangle of `v` will cover at most a 2×2 block of texels. We sample these texels at the corners of the axis aligned bounding rectangle. The minimum and maximum of these values gives us a conservative bound on the value of $\mathcal{I}_i(\mathcal{C}_i(\mathbf{p}))$, for all $\mathbf{p}$ within a voxel. By taking the product of the minimums (similarly maximums) over the images, we get a conservative bound on the value of $\mathcal{S}(\mathbf{p})$ for all $\mathbf{p}$ within a voxel. This means that we will not miss small features or create holes in our voxelization, since our bounds are conservatively correct. Furthermore, our silhouette image space acceleration structure greatly reduces the amount of computation and data accesses needed by the oracle kernel to make a culling decision for a given voxel. The n image pyramids are generated by a compute kernel that is launched immediately after silhouette image smoothing.

4.4 Voxel Rendering

We render voxels as screen space bounding rectangle splats. That is, we project voxels to screen space, find their axis aligned bounding rectangles, and render these quads as a pair of triangles. For typical views that include the upper body and head of the capture subject, giga voxel resolution splats project to roughly the size of a pixels at HD resolution. While GPU rasterization is not currently optimized for such single pixel quads, the rendering performance we experience is still acceptable, see Section 5. Image quality suffers from minor aliasing artifacts; though no more than is typical of non anti-aliased renderings of triangle models. Our splat based rendering approach does show significant artifacts when a voxelized model is viewed at close range. In such cases, voxel splats will appear as rectangular blocks covering many pixels. However, voxel splats still project to a small pixel area in the RGB images, so close up views will also show pixelation artifacts induced by the finite resolution of the RGB images. However, close-up views are not needed for our intended application: live 3D human head and upper body capture and visualization.

We texture voxel splats by weighting the contribution of the voxel's projection into each of the color images. We determined this weighting as the product of two weight terms; a view dependent weight, and a normal dependent weight. The code snippet below shows how we compute these weights. The variable `VtoEye` represents the unit vector from the voxel center to the novel view eye point, `Normal` is the unit normal to the surface at the voxel center, and `VtoCamera[i]` is the unit vector from the voxel center to the position of camera `i`.

```
dotV = dot(VtoEye, VtoCamera[i]);
dotN = dot(Normal, VtoCamera[i]);

weightV = dotV>0.0 ? pow(dotV, 4) : 0.0;
```

Figure 3: *Top row: images captured with our eight camera rig for a single frame. Bottom row: corresponding smoothed silhouette images using a 41 pixel wide Gaussian kernel.*

```
weightN = dotN>0.0 ? pow(dotN, 4) : 0.0;
weight[i] = weightV * weightN;
```

Finally, the values of `weight` are normalized by dividing by their sum. The calls to `pow()` in the code above helps to reduce artifacts in transitions between dominant textures.

5 Experimental Results

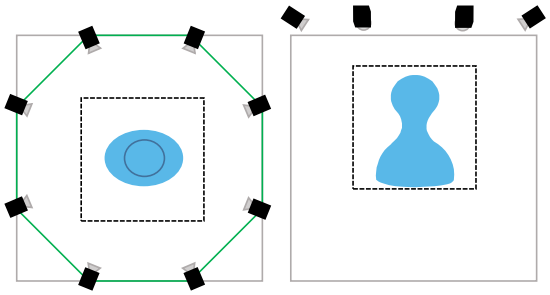

Figure 4: *Left: Top view of our capture rig layout showing 8 cameras, position of green-screen (green octagon), and human subject within (virtual) NVC cube. Right: Capture rig side view.*

We validated our voxelization and smooth visual hull reconstruction using an eight camera rig. The rig consisted of a 5x5x5 foot cube with the cameras mounted in portrait orientation pointed slightly downward on the top edges in an octagonal configuration, see Figure 4. The cameras were calibrated once at rig setup time using Zhang's method [Zhang 2000]. A human subject was seated at the center of the rig. We placed a green-screen curtain within the rig to surround the subject. We used eight Point Grey Flea2 cameras with shutters synchronized by an external trigger. The cameras were connected in two groups of four to a pair of IEEE 1394 hubs, which are in turn connected to a single IEEE 1394 PCI card. With this equipment, we were limited to eight 800x600 8 bit images streams running at 30 hertz. This data bandwidth limitation is currently the bottleneck of our capture system. Figure 3 shows a set of eight captured frames (top row), and the corresponding smoothed silhouette images (bottom row).

We placed a monitor above the the top edge of the rig and gave subjects a wireless game controller so they could see and rotate their reconstructed model in real-time. Figure 1 shows a reconstructed 3D model textured using the input images. Since the reconstructed model is based entirely on silhouette images, it is not highly accurate geometrically. This inaccuracy leads to the *ghosting* artifacts especially visible around the ear in the far left of Figure 1. We show our capture rig hardware and the results of several more capture sessions in the companion video. Additional examples models are shown in Figure 6.

	GeForce Titan		Radeon 7970GE	
level	9	10	9	10
demosaic, etc.	0.46	0.46	0.43	0.43
convolution	2.49	2.49	1.56	1.56
mip generation	0.44	0.44	0.58	0.58
generate	0.06	0.06	0.11	0.11
oracle	1.66	5.27	2.09	6.47
scan	1.48	3.19	1.01	2.69
subdivide	0.87	2.33	0.98	2.71
compactify	0.23	0.71	0.21	0.61
compute normals	0.49	1.74	0.56	1.97
render	1.57	6.30	4.47	17.87
total	9.75	23.86	14.64	35.00
fps	112	46	83	30

Table 1: *Detailed timing analysis for constructing and rendering a voxel model for a single frame. All timings are given in milliseconds. For the test frame level 9 contained 606660 voxels; level 10 contained 2538713 voxels. We used a* 1080×1920 *render target.*

5.1 Performance

We used DirectX 11 and HLSL to code all compute and graphics shaders. Since DirectX is cross-platform, we ran experiments using two different high end graphics cards. an nVidia GeForce Titan, and an AMD Radeon 7970GE. We present a detailed timing analysis of the various processing stages of our algorithm in Table 1. We measured performance on the single frame of image data shown in Figure 3 held in CPU memory. For each processing step measured, we flushed the GPU pipeline and took 100 frame moving averages. The Titan outperformed the Radeon overall, but not for all kernels. Rendering was significantly slower on the Radeon, and generally the most costly processing step. We used block sizes of 1024 voxels for all voxel processing compute kernels except **Generate**, where we used a block size of 128.

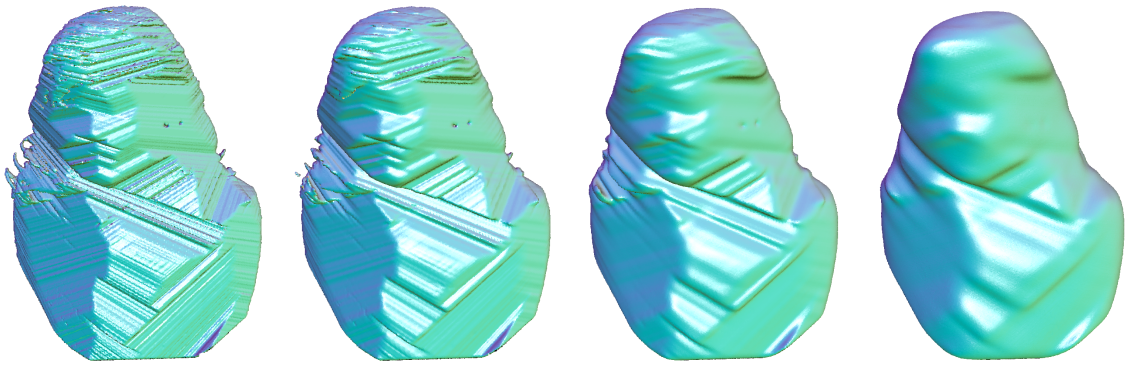

Figure 5: *We voxelize a smoothed visual hull of a human subject using multiple live image streams. We show the effect of image smoothing kernel width on the reconstructed 3D model. Kernel width (in pixels) from left to right: 11, 21, 41, 81.*

6 Future Work

In the interest of simplicity, and because the benefit for our capture/visualization scenario would not be great, we pursued a *uniform* voxel subdivision strategy rather than an *adaptive* one. Adaptive subdivision in the list-based, breadth-first GPU paradigm was originally done in [Patney and Owens 2008], with several improvements appearing in [Eisenacher et al. 2009]. Rather than maintaining a single oracle *decision* list, we would add a second *done* list. In addition to a culling decision, the oracle would also decide if the stopping criterion had been reached, and then write the decision to the appropriate list. Processing would stop when there are no more voxels to subdivide; voxels in the done list would then be rendered. In addition to voxel indices, we would need store the level n of a voxel as well.

We used the graphics pipeline for rendering despite the well-known fact that current GPUs do not efficiently rasterize pixel sized triangles. We did this for convenience, and to avoid dealing with the complexities of color and depth buffer write hazards that a software voxel rasterizer would need to handle. Regardless, it is not clear if a GPU-based software voxel rasterizer would outperform hardware, but this issue is worth exploring.

We also plan to experiment with other image based modeling oracles. Among these are:

Photo Consistency Oracle: If a voxel's projection into two or more images is similar (photo consistent), then it is likely the voxel is part of an object's surface and should be kept. This is the idea underlying *space carving* [Kutulakos and Seitz 2000]. Our voxelization method could be used to implement GPU hierarchical space carving similar to [Hornung and Kobbelt 2006].

Incorporating Depth Data: In order to improve the geometric fidelity of our models, we plan to incorporate depth sensor image data. Adding a single depth sensor could help to improve our models, especial faces. However, synchronizing this senor with our existing camera rig remains a challenge. Ideally, we would like to use a multiple depth sensor configuration. Current structured IR light based depth sensor technologies suffer from inter-sensor interference. We are looking at stereo based depth sensing as an alternative, and plan to adapt the oracle kernel to improve the quality of our 3D models.

7 Conclusions

We have presented an efficient GPU based surface voxelization system for image based modeling applications. The approach operates over voxels in parallel utilizing an *oracle kernel* to decide if a given voxel, and all subsequent child voxels, should be kept (1), or culled (0), from the voxel list. The voxels are subdivided or compacified based on a parallel prefix sum scan of culling decisions. This process repeats until a sufficiently hi-resolution voxelization is reached. The resulting pixel sized voxels are rendered as splats, textured by a weighted combination of the input color images.

We demonstrated the effectiveness of this system on the problem of smooth visual hull reconstruction. While this problem has been solved in the computer vision literature for some time, we presented a novel method for smoothing the visual hull and providing a normal field and showed that a very high resolution voxelization can be carried out in real time. Our results also suggest that silhouette information can be useful in creating a smooth proxy geometry for applying blended hair images, resulting in a photo-realistic reconstruction of the a human scalp.

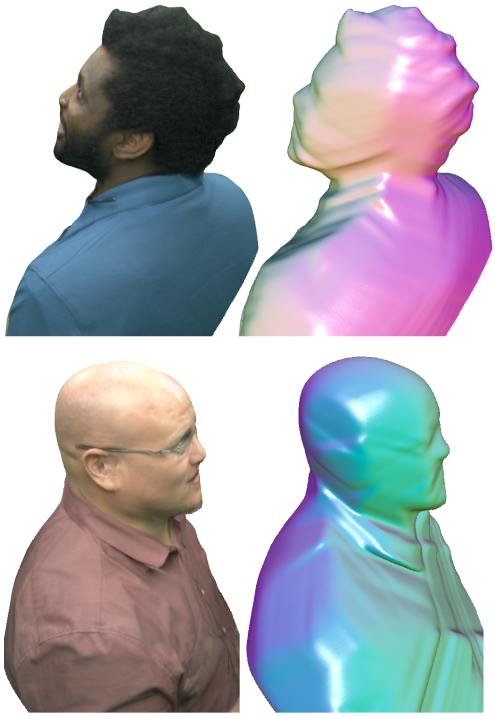

Figure 6: *Additional examples of models reconstructed with, and without texture*

Acknowledgements

We thank the volunteer test subjects who allowed us to use their images as examples in this paper and video.

References

AJMERA, P., GORADIA, R., CHANDRAN, S., AND ALURU, S. 2008. Fast, parallel, gpu-based construction of space filling curves and octrees. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, 10.

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *SIGGRAPH*, vol. 21, ACM, 785–790.

BLELLOCH, G. E. 1990. *Vector models for data-parallel computing*. MIT Press.

CARRANZA, J., THEOBALT, C., MAGNOR, M. A., AND SEIDEL, H.-P. 2003. Free-viewpoint video of human actors. *ACM Transactions on Graphics (TOG) 22*, 3, 569–577.

CRASSIN, C., AND GREEN, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights* (July), 303–319.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 15–22.

CURLESS, B., AND LEVOY, M. 1996. A volumetric method for building complex models from range images. In *SIGGRAPH*, ACM, 303–312.

DE AGUIAR, E., STOLL, C., THEOBALT, C., AHMED, N., SEIDEL, H.-P., AND THRUN, S. 2008. Performance capture from sparse multi-view video. *ACM Trans. Graph. 27*, 3.

EISEMANN, M., DE DECKER, B., MAGNOR, M. A., BEKAERT, P., DE AGUIAR, E., AHMED, N., THEOBALT, C., AND SELLENT, A. 2008. Floating textures. *Comput. Graph. Forum 27*, 2, 409–418.

EISENACHER, C., MEYER, Q., AND LOOP, C. T. 2009. Real-time view-dependent rendering of parametric surfaces. In *SI3D*, 137–143.

GREENE, N., KASS, M., AND MILLER, G. S. P. 1993. Hierarchical z-buffer visibility. In *SIGGRAPH*, 231–238.

HARRIS, M., SENGUPTA, S., AND OWENS, J. D. 2007. Parallel prefix sum (scan) with cuda. *GPU gems 3*, 39, 851–876.

HORNUNG, A., AND KOBBELT, L. 2006. Hierarchical volumetric multi-view stereo reconstruction of manifold surfaces based on dual graph embedding. In *CVPR (1)*, 503–510.

HORNUNG, A., AND KOBBELT, L. 2009. Interactive pixel-accurate free viewpoint rendering from images with silhouette aware sampling. *Comput. Graph. Forum 28*, 8, 2090–2103.

IZADI, S., KIM, D., HILLIGES, O., MOLYNEAUX, D., NEWCOMBE, R. A., KOHLI, P., SHOTTON, J., HODGES, S., FREEMAN, D., DAVISON, A. J., AND FITZGIBBON, A. W. 2011. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *UIST*, 559–568.

KNISS, J., LEFOHN, A., STRZODKA, R., SENGUPTA, S., AND OWENS, J. D. 2005. Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches*, ACM, 16.

KUSTER, C., POPA, T., ZACH, C., GOTSMAN, C., AND GROSS, M. H. 2011. Freecam: A hybrid camera system for interactive free-viewpoint video. In *VMV*, 17–24.

KUTULAKOS, K. N., AND SEITZ, S. M. 2000. A theory of shape by space carving. *International Journal of Computer Vision 38*, 3, 199–218.

LADIKOS, A., BENHIMANE, S., AND NAVAB, N. 2008. Efficient visual hull computation for real-time 3d reconstruction using cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, IEEE, 1–8.

LAINE, S., AND KARRAS, T. 2011. Efficient sparse voxel octrees. *IEEE Trans. Vis. Comput. Graph. 17*, 8, 1048–1059.

LAURENTINI, A. 1994. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell. 16*, 2, 150–162.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Octree textures on the gpu. *GPU gems 2*, 37.

MAIMONE, A., AND FUCHS, H. 2011. Encumbrance-free telepresence system with real-time 3d capture and display using commodity depth cameras. In *ISMAR*, 137–146.

MATUSIK, W., BUEHLER, C., RASKAR, R., GORTLER, S. J., AND MCMILLAN, L. 2000. Image-based visual hulls. In *SIGGRAPH*, 369–374.

PATNEY, A., AND OWENS, J. D. 2008. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph. 27*, 5, 143.

SCHICK, A., AND STIEFELHAGEN, R. 2009. Real-time gpu-based voxel carving with systematic occlusion handling. In *DAGM-Symposium*, 372–381.

SCHWARZ, M., AND SEIDEL, H.-P. 2010. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph. 29*, 6, 179.

WILHELMS, J., AND GELDER, A. V. 1992. Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3, 201–227.

ZHANG, C., CAI, Q., CHOU, P. A., ZHANG, Z., AND MARTIN-BRUALLA, R. 2013. Viewport: A distributed, immersive teleconferencing system with infrared dot pattern. *IEEE MultiMedia 20*, 1, 17–27.

ZHANG, Z. 2000. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell. 22*, 11, 1330–1334.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2011. Data-parallel octrees for surface reconstruction. *IEEE Trans. Vis. Comput. Graph. 17*, 5, 669–681.