Efficient View-Dependent Sampling of Visual Hulls

Wojciech Matusik Chris Buehler Leonard McMillan
Computer Graphics Group
MIT Laboratory for Computer Science
Cambridge, MA 02141

Abstract

In this paper we present an efficient algorithm for sampling visual hulls. Our algorithm computes exact points and normals on the surface of visual hull instead of a more traditional volumetric representation. The main feature that distinguishes our algorithm from previous ones is that it allows for sampling along arbitrary viewing rays with no loss of efficiency. Using this property, we adaptively sample visual hulls to minimize the number of samples needed to attain a given fidelity. In our experiments, the number of samples can typically be reduced by an order of magnitude, resulting in a corresponding performance increase over previous algorithms.

1. Introduction

Recently, shape-from-silhouettes techniques have been used in real-time, shape-acquisition applications [5, 3]. Typically, shape-from-silhouettes techniques involve computing a volume known as the *visual hull*, which is the maximal volume that reproduces the observed silhouettes. It has been found that visual hulls can be computed very quickly, and that the calculation is robust to the crude silhouettes produced by real-time segmentation algorithms.

People commonly compute visual hulls in real-time using one of two approaches: voxel carving [6, 9] and view ray sampling [3]. In voxel carving, a discrete grid of voxels is constructed around the volume of interest. Then, each voxel in the grid is checked against the input sihouettes, and any voxels that project outside the silhouettes are removed from the volume. This procedure results in a view-independent volumetric representation that may contain quantization and aliasing artifacts due to the discrete voxelization. Voxel carving can be accelerated using octree representations.

In view ray sampling, a sampled representation of the visual hull is constructed. The visual hull is sampled in a view-dependent manner: for each viewing ray in some desired view, the intersection points with all surfaces of the visual hull are computed. This procedure removes much of the quantization and aliasing artifacts of voxel carving, but it does not produce a view-independent model. View ray sampling can be accelerated by sampling the visual hull in a regular pattern, such as in a regular grid of pixels.

In this paper, we present a new algorithm for computing view ray sampling. Our algorithm is distinguished by the fact that it allows for computing *arbitrary* patterns of samples with the same efficiency as previous algorithms. We use our algorithm to accelerate visual hull construction by *adaptively* sampling the visual hull. We adjust the density of samples such that more samples are used in regions of large depth variation and fewer samples are used in smooth regions. Using this sampling procedure, we can reduce the number of samples used to construct the visual hull by more than 90% with almost no loss of fidelity. We also demonstrate how to compute surface normals at each sample point. Further, these normals can be used to direct the adaptive sampling procedure.

1.1. Previous Work

Laurentini [2] originated the idea of the *visual hull*: the maximal volume that reproduces all silhouettes of an object. In this paper, visual hulls are constructed from a finite number of silhouettes, so they are only guaranteed to reproduce those particular silhouettes. A visual hull is essentially a volume formed from the intersection of *silhouette cones*. A silhouette cone is the volume that results from extruding a silhouette out from its center of projection. It may be a complicated, non-convex object, depending on the complexity of the silhouette contours. We represent silhouette contours with line segments, so the resulting silhouette cones are facted.

Visual hulls are most often computed using voxel carving[6, 9]. If the primary use for the visual hull is to produce new renderings, then a view ray sampling approach such as the image-based visual hull technique introduced by [3] may be used. The advantage of view ray sampling algorithms is that they do not suffer from the quantization artifacts introduced by discrete volumetric representations.

A series of optimizations are discussed in [3] that reduce the computational overhead of view ray sampling on average to a constant cost per sample. At the heart of these op-

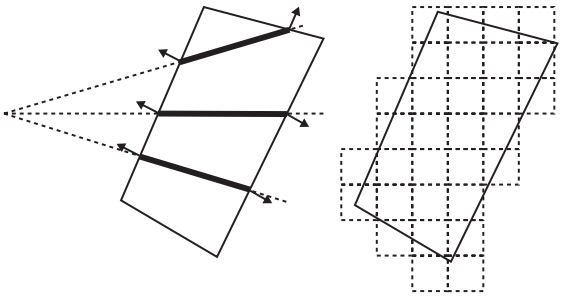


Figure 1: Our algorithms compute a representation of the visual hull that is sampled along viewing rays (left). This is in contrast to a voxel–based representation (right).

timizations is a presorting of the silhouette edges about the epipole of the desired image. This sorting allows samples to be computed in constant time if the samples are arranged in scanline order. This initial sorting also involves some additional cost.

1.2. Contributions

In this paper, we present an improved algorithm that allows the samples to be computed efficiently in any order. This flexibility makes it easy to do hierarchical or adaptive sampling of the visual hull. In addition this algorithm visits the silhouettes in a "lazy" fashion, using only the portions of the silhouettes necessary. We also describe how to compute normals for the visual hull samples, which can be useful for shading and surface reconstruction.

2. Visual Hull Sampling

One way to compute an exact sampling of a visual hull consists of two steps: (1) compute a set of polygons that defines the surface of a visual hull and (2) sample these polygons along the sampling rays to produce exact visual hull samples. The first step consists of a 3D intersection of all the extruded silhouette cones. The second step is also an intersection operation–an intersection of sampling rays with the volume. The ray intersections result in a set of occupied volume intervals along each sampling ray.

The algorithm in section 2 is conceptually simple. However, it is impractical to compute full 3D intersections between silhouette cones, especially when we are ultimately interested in samples. We would like to compute the same result without performing full volume–volume intersections. To do this, we take advantage of the commutative properties of the intersection operation. An intersection of a ray with a visual hull is described mathematically as follows:

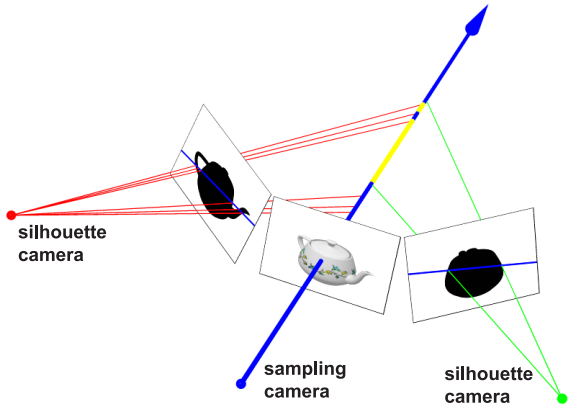


Figure 2: Instead of computing the intersection of the sampling ray with extruded silhouette cones, we can project the sampling ray onto the silhouette images, compute the intersections of the projected sampling rays with the silhouette, and lift back the resulting intervals to 3D.

$$VH(S) = \left(\bigcap_{s \epsilon S} cone(s) \right) \cap ray3D \qquad (1)$$

This operation is equivalent to:

$$VH(S) = \bigcap_{s \epsilon S} (cone(s) \cap ray3D) \qquad (2)$$

This means that we can first intersect each extruded silhouette with the 3D ray separately. This results in a set of occupied intervals along the ray. Then we compute the intersection of all these sets of intervals for all silhouettes. In this process we exchanged volume-volume and volume-line intersections for simpler volume-line and line-line intersections.

2.1. Image Space Intersections

We can further simplify the intersection process by exploiting the fact that the cross–section of the extruded silhouette remains fixed. This observation implies that instead of computing the ray intersection with the *extruded* silhouette, we can compute the intersection of the silhouette with the ray projected onto the plane that defines the cross-section. We can then backproject the results of the image space intersection onto the original 3D ray (see Figure 2). Effectively we reduce the volume-line intersections to area-line intersections. As we will see in the next section, this reduction allows us to use simple 2D spatial partitioning for accelerating ray intersections.

We can pick any plane that completely intersects the silhouette cone when we perform the area-line intersection.

However, it is simplest to perform the operation on the silhouette camera's image plane because (1) the cross-section is already defined for this plane (it is the silhouette) and (2) we avoid any possible resampling artifacts.

3. Silhouette-Line Intersections

In this section, we describe an efficient technique for computing the intersection of a set of 2D lines with silhouette contours. We impose one constraint on the sampling rays: we require that they all emanate from a single point in space, the *sampling camera*. This limitation on our algorithm is not too severe, as it is often desired to sample the visual hull from the point-of-view of an arbitrary camera or one of the silhouette cameras. Note that we do not constrain the sampling rays to be on a regular grid or in any other structured organization. Also, the algorithm can work with a set of parallel sampling rays. This case corresponds to an orthographic sampling camera with a center of projection at infinity.

Our algorithm is based on a data structure called a "Wedge-Cache." The Wedge-Cache is used to store intersection information that can be re-used when computing later intersections. The main idea behind the Wedge-Cache algorithm is based on the epipolar geometry of two views. An epipolar plane (i.e., a plane that passes through centers of projections of both cameras) intersects the image planes of both cameras in epipolar lines [1]. It is easy to see that all rays from the first camera that lie in the epipolar plane project to the exact same epipolar line in the second camera (of course, the reverse is true too). Therefore, many sampling rays will project to the same line in any given silhouette view. The Wedge-Cache algorithm takes advantage of this fact to compute the fast line-silhouette intersections. The basic idea is to compute and store the intersection results for each epipolar line the *first* time it is encountered. Then, when the same epipolar line is encountered again, we can simply look up previously computed results.

In practice (because of discretization), we rarely encounter the exact same epipolar line twice. Since we want to reuse silhouette intersections with epipolar lines, we compute and store the intersections of the silhouette with wedges (i.e., sets of epipolar lines) rather than single lines. Within each wedge, we store a list of silhouette edges that individual lines within the wedge might intersect. Then, when an epipolar line falls within a previously computed wedge, we need only intersect that line with the silhouette edges belonging to the wedge.

We discretize the silhouette image space into a set of wedges such that each wedge has exactly one pixel width at the image boundaries (see Figure 3). Depending on the position of the epipole (shown as a dot) with respect to the silhouette image, we distinguish nine possible cases of image boundary parts that need to be used. These cases are illustrated in Figure 4. There is only one special case that needs to be handled: the case in which the epipolar lines in a silhouette camera are parallel and do not converge at the epipole (i.e., the epipole is at infinity). In this case, we can modify the Wedge-Cache to use parallelogram-shaped "wedges" (see Figure 5). In some applications, this case can be avoided by a small perturbation in the orientation of the sampling camera.

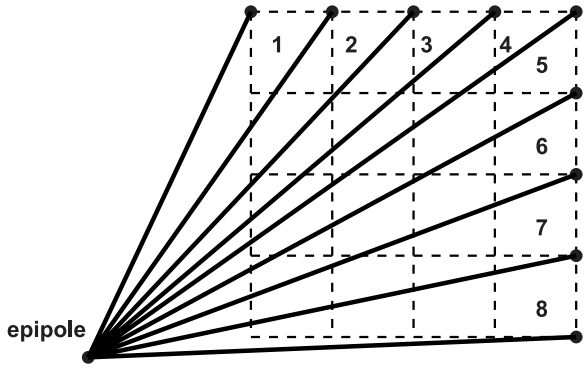


Figure 3: The silhouette image is partitioned into a set of wedges, which are the entries of the Wedge-Cache. In this simple example, the silhouette image is 4×4 pixels and there are eight Wedge-Cache entries.

Execution of the Wedge-Cache algorithm proceeds as follows. For each sampling ray we compute its epipolar line. Then we determine into which wedge it falls. If silhouette edges that lie in this wedge have not been computed, we use a Bresenham-like algorithm to traverse all the pixels in the wedge and compute these edges. Then, we test which of the computed edges in the wedge actually intersect the given epipolar line. Later, when other epipolar lines fall into this wedge we simply look up the edges contained in the wedge and test for intersection with the epipolar line. Figure 6 illustrates a simple Wedge-Cache with two wedges. The silhouette in this case is a square consisting of four edges $a, b, c,$ and d. Each wedge contains three edges as shown in the figure. In this example, the epipolar line corresponding to some sampling ray is contained in wedge 1. Checking against the three edges in wedge 1 reveals that the line intersects two edges, a and d.

One nice property of the Wedge-Cache algorithm is that it employs a lazy-computation strategy; we process pixels and edges in the wedge only when we have an epipolar line that lies in this wedge. The pseudocode for `VHsample` that uses the Wedge-Cache Algorithm is shown in Figure 7.

```
VHsample (samplingRays R, silhouettes S)
  for each silhouetteImage s in S
    compute_silhouette_edges(s)
    for each samplingRay s in R do
      r.intervals = 0..inf
    for each silhouetteImage s in S
      clear(Cache)
      for each samplingRay r in R
        lineSegment2D l2 = project_3D_ray(r,s.camInfo)
        integer index = compute_wedge_cache_index(l2)
        if Cache[index] == EMPTY
          silhouetteEdges E = trace_epipolar_wedge(index, s)
          Cache[index] = E
        intervals int2D = linesegment_isect_silhouette(l2,Cache[index])
        intervals int3D = lift_up_to_3D(int2D,r.camInfo,ry3)
        r.intervals = intervals_isect_intervals(r.intervals,int3D)
```

Figure 7: Pseudocode for Wedge-Cache algorithm.

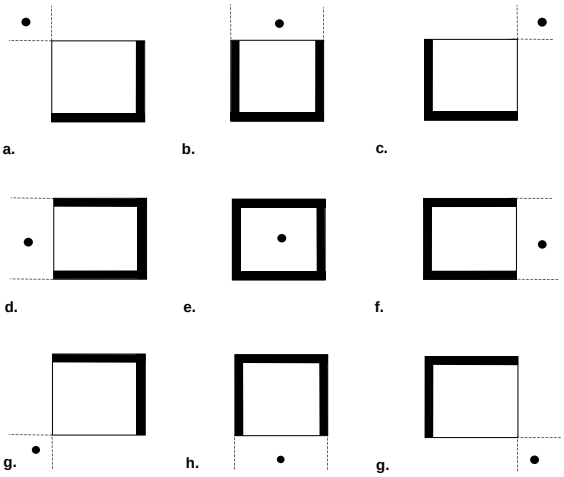


Figure 4: Depending on the position of the epipole with respect to the silhouette image boundary, we decide which parts of the silhouette image boundary (thick black lines) need to be used for wedge indexing.

3.1. Using Valid Epipolar Line Segments

Some care must be taken when implementing the `calc2Dintervals` subroutine. This is because some portions of the epipolar line are not valid and should not be considered. There are two constraints on the extent of the epipolar line: (1) it must be in front of the sampling camera and (2) it must be seen by the silhouette camera. The two constraints result in four cases that are easily distinguished; see [4] for details.

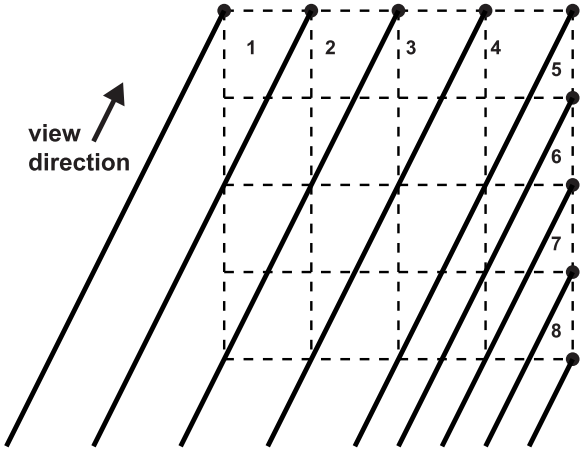


Figure 5: In the special case when the epipolar lines are all parallel, the wedges become parallelograms.

3.2. Visual Hull Surface Normals

In this section we show how to compute visual hull surface normals for each of the interval endpoints of the sampled visual hull representation. The surface normal is useful to reconstruct the surface of the visual hull, and we use normals in this manner to control the adaptive sampling procedure described in Section 4. Of course, the normal of the visual hull is not the same as the normal of the original object. However, as the number of silhouettes increases, the normals of the visual hull approach the normals of the object in non-concave regions.

Normals are simple to compute with just a little extra bookkeeping. For each interval endpoint we store a refer-

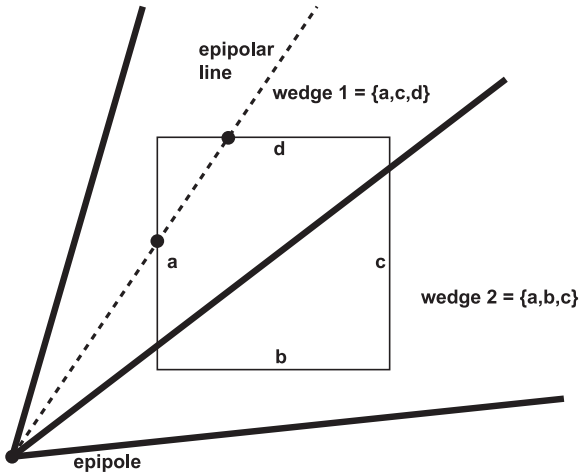


Figure 6: A simple Wedge-Cache example with a square silhouette. The epipolar line is contained in wedge 1, so it need only be compared to edges a, c, and d for intersection. In fact, the line intersects only edges a and d.

ence to the silhouette edge and the silhouette image that determine the interval endpoint. Each interval is then defined as $((depth_{start}, edge_{i,m}), (depth_{end}, edge_{j,n}))$, where i and j are the indices of the reference images and m and n are the silhouette edge indices in the reference images. The stored edges and the center of projection of the corresponding reference image determine a plane in 3D. The normal of this plane is the same as the surface normal of the point on the visual hull (see Figure 8). We can compute the plane normal using the cross-product of the two vectors on this plane. This leaves two choices of normals (differing in sign); the proper normal can be chosen based on the direction of the sampling ray and the knowledge of whether the sampling ray is entering or leaving the visual hull.

4 Adaptive Visual Hull Sampling

One advantage of the Wedge Cache algorithm is that it allows for sampling along arbitrary viewing rays in any order. We have used this property to implement an adaptive sampling procedure that can drastically reduce the number of samples required to construct an image of the visual hull from a particular view.

First, we decide upon a minimum size N of features that we expect to see in the visual hull. This choice determines the smallest features of the visual hull that our algorithm can resolve. Typically, we choose a minimum feature size of $N = 4$, which means the algorithm can potentially miss features smaller than 4×4 pixels.

Next, we perform an initial sampling of the visual hull

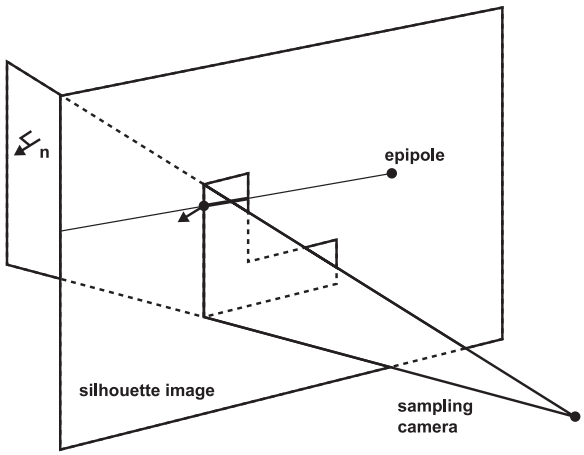


Figure 8: A silhouette edge combined with the center of projection define a plane. Visual hull interval endpoints defined by that edge have the same normal as the plane.

by sampling every N^{th} pixel in both the x and y directions. This initial sampling results in a coarse grid of samples over the image. For each square in the grid, we consider the samples at its four corners. If all four sample rays miss the visual hull, then we conclude that no rays within the square hit the visual hull, and we mark that square as empty. If some of the rays miss the visual hull and some of them do not, then we conclude that a silhouette boundary passes through this square. In this case, we sample the rest of the rays within the square to resolve the silhouette edge accurately.

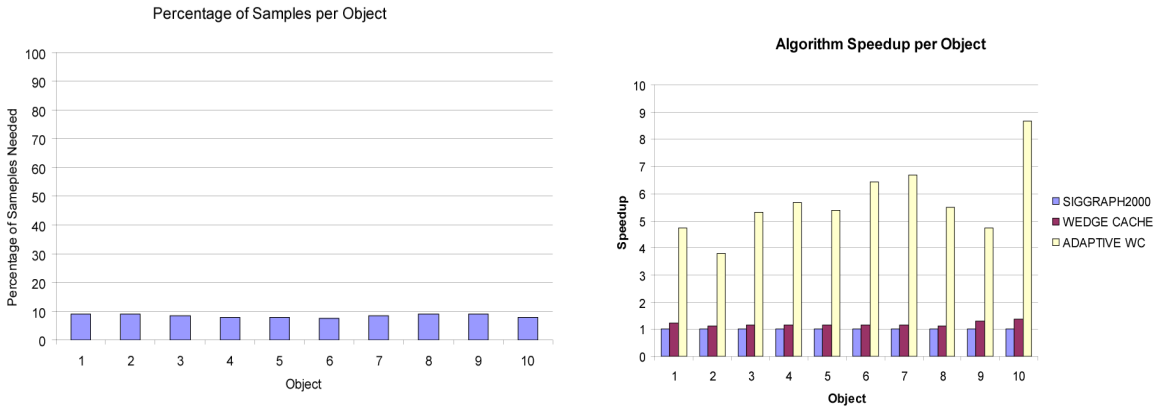
If all four sample rays hit the visual hull, then we decide whether to sample further based on a simple estimate of the surface continuity. Since we know the normals at each of the four samples, we can construct a plane through one the sample points on the visual hull. If this plane does not predict the other three sample points sufficiently well, then we sample the rest of the rays within the square. Otherwise, we approximate the depths of the samples within the square using the planar estimate. We compare the prediction error against a global threshold to decide if more sampling is necessary.

5. Results

In this section, we present some results of our adaptive sampling algorithm. In Figure 9a, we show the depth map of a visual hull sampled at every pixel. This visual hull is computed from 108 input images, and it is sampled at 1024×1024 resolution (1048576 samples). In Figure 9b, we have the same view of the visual hull, but this time it is sampled with only about 9% of the samples. The mean squared error between the two depth maps is 0.1 (the depth maps are quantized to 8 bits). Figure 9c shows the sampling

(a) (b) (c)

Figure 9: Visual hull sampling results. (a) Shows the result of sampling every pixel. (b) Shows the result of adaptively sampling only about 9% of the pixels. (c) Shows where the adaptive sampling procedure increased the sampling density.



(a) (b)

Figure 10: Wedge cache performance results. (a) Shows the percentage of samples that are computed using adaptive sampling. (b) Shows wedge cache performance speedup as compared to the algorithm in [3].

pattern used for this view of the visual hull.

We have found that adaptive sampling of visual hulls is fruitful on a wide variety of different objects. In Figure 10a we have plotted the percentage of samples needed to render a visual hull given a fixed threshold for evaluating surface continuity.

We compared the runtime performance of our algorithm to the algorithm described in [3]. The results are shown in Figure 10b. When fully sampling every pixel in the image, our wedge cache algorithm is slightly faster on all objects. When we enable adaptive sampling, the speedup is more dramatic. On average, there is a factor of five speedup, which would directly result in increased framerates.

6. Conclusions

We have presented a new algorithm for efficiently sampling visual hulls. Our algorithm computes exact samples of the visual hull surface along with surface normals. The algorithm computes these samples along an arbitrary set of sampling rays that emanate from a common point, possibly at infinity.

Using this algorithm, we have demonstrated a simple way to adaptively sample depth maps of visual hulls from virtual viewpoints. By using this adaptive sampling, the performance of the algorithm is increased on average 5 times with almost no loss in quality of the resulting depth maps. In real-time systems, for which visual hull analysis has been found useful, this performance increase translates directly into faster framerates.

References

[1] Faugeras, O. *Three-Dimensional Computer Vision.* MIT Press. 1993.

[2] Laurentini, A. “The Visual Hull Concept for Silhouette Based Image Understanding.” *IEEE PAMI* 16,2 (1994), 150-162.

[3] Matusik, W., Buehler, C., Raskar, R., Gortler, S., and McMillan, L. “Image-Based Visual Hulls,” *SIGGRAPH 2000*, July 23-28, 2000, 369-374.

[4] McMillan, L. “An Image-Based Approach to Three-Dimensional Computer Graphics,” Ph.D. Thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1997.

[5] Kanade, T., P. W. Rander, P. J. Narayanan. “Virtualized Reality: Constructing Virtual Worlds from Real Scenes,” *IEEE Multimedia*, 4, 1 (March 1997), pp. 34-47.

[6] Potmesil, M. ”Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images.” *CVGIP 40* (1987), 1-29.

[7] Roth, S. D., “Ray Casting for Modeling Solids.” *Computer Graphics and Image Processsing*, 18 (February 1982), 109-144.

[8] Snow, D., Viola, P., and Zabih, R., “Exact Voxel Occupancy with Graph Cuts,” *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition.* 2000.

[9] Szeliski, R. “Rapid Octree Construction from Image Sequences.” *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.