Noname manuscript No.
(will be inserted by the editor)

GPU-based Parallel Construction of Compact Visual Hull Meshes

Byungjoon Chang · Sangkyu Woo · Insung Ihm

Received: date / Accepted: date

Abstract Building a visual hull model from multiple two-dimensional images provides an effective way of understanding the three-dimensional geometries inherent in the images. In this paper, we present a GPU accelerated algorithm for volumetric visual hull reconstruction that aims to harness the full compute power of the many-core processor. From a set of binary silhouette images with respective camera parameters, our parallel algorithm directly outputs the triangular mesh of the resulting visual hull in the indexed face set format for a compact mesh representation. Unlike previous approaches, the presented method extracts a smooth silhouette contour on the fly from each binary image, which markedly reduces the bumpy artifacts on the visual hull surface due to a simple binary in/out classification. In addition, it applies several optimization techniques that allow an efficient CUDA implementation. We also demonstrate that the compact mesh construction scheme can easily be modified for also producing a time- and space-efficient GPU implementation of the marching cubes algorithm.

Keywords Visual hull · volumetric approach · compact mesh · GPU algorithm · CUDA implementation · marching cubes algorithm.

B. Chang, S. Woo, I. Ihm (Corresponding author)
Department of Computer Science and Engineering
Sogang University, Seoul, Korea
Tel.: +82-2-705-8493
Fax: +82-2-704-8273
E-mail: jerrun@sogagn.ac.kr, coldnight.w@gmail.com, ihm@sogang.ac.kr

1 Introduction

Since it was introduced to the computer vision community, the idea of reconstructing three-dimensional (3D) shapes from object silhouettes in two-dimensional (2D) images [1] has been applied to model static or dynamic 3D objects effectively in scenes. Given silhouette images from multiple camera views along with their viewing parameters, a visual hull can be constructed by intersecting the silhouette cones they respectively define [8], thus representing the maximal volume implied by the silhouettes.

Volumetric methods employ a fixed or adaptive volume grid representation to produce a visual hull. A set of small 3D cells that approximate the visual hull region are generated or its boundary surface is polygonized using a surface extraction technique such as the marching cubes algorithm [10] (refer to, for instance, [15] for a quick review of volumetric visual hull methods). The volumetric approach, while numerically robust, has sometimes been regarded as less accurate than the polyhedral approach, e.g. [11,5], that attempts to compute the exact intersection of silhouette cones via explicit geometry processing. Current hardware systems, however, cope easily with high-resolution grids to increase the precision of the resulting visual hull. Furthermore, thanks to efforts to sample the volume space adaptively and estimate its boundary surface more accurately, e.g. [4,9], high-quality visual hull meshes are now routinely generated by the volume-based methods.

Important advantages of the volumetric approach are the simplicity of its algorithm and its inherent parallelism in computation, which allows an efficient parallel implementation with current hardware (refer to [7] to see some previous implementations in various parallel environments). In particular, it is well suited to implementation on current GPUs, which are highly parallel, multithreaded, many-core processors. This observation naturally led to several GPU-based implementations of volumetric visual hull algorithms [7, 14, 16], particularly using the compute unified device architecture (CUDA) API from NVIDIA [12].

In this paper, we present a GPU accelerated parallel algorithm for volumetric visual hull reconstruction that, from a sequence of multiple binary silhouette images, constructs exact visual hull models effectively by fully exploiting the compute power of the many-core processor. Unlike previous marching cubes-based techniques that simply list extracted triangles with the same vertices repeated in the representation, our GPU algorithm removes such duplication and produces a triangular mesh in compact form using the *indexed face set* method so that the resulting mesh is instantly available for efficient applications. Our method extracts smooth piecewise-linear silhouette contours on the fly for a sophisticated voxel classification, which leads to significant reduction of the bumpy artifacts that often occur on the visual hull surface due to a simple binary in/out classification. For a time- and space-efficient CUDA implementation, we apply several optimization and data-parallel programming techniques including parallel prefix sum [6] and parallel radix sort [13]. In particular, as in [9], our method also estimates the exact locations of vertices on the visual hull surface using input silhouette images. However, ours is based on the concept of perspective correction, which requires fewer floating-point operations to implement. Last but not least, the presented GPU technique for the direct construction of a compact mesh can easily be modified for also producing an efficient GPU implementation of the marching cubes algorithm, as demonstrated in the paper.

2 Our methods

Our GPU computation framework has three main phases, which are explained in the following subsections. Throughout this work, we assume that the 3D computational volume is discretized into a regular grid of given resolution, where its grid points and cubes made of eight neighboring grid points are called *voxels* and *cells*, respectively. Recall that the input to our scheme is a set of multiple binary silhouette images with respective camera viewing parameters, which is repeatedly produced for every time frame (see Fig. 1 for some example input images). In particular, the object to be reconstructed is represented as black pixels in the images, and the background as white pixels.

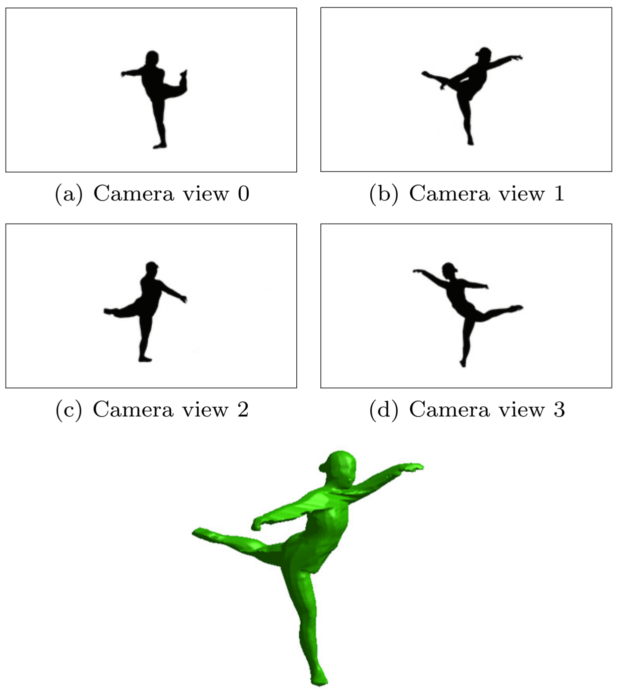

(a) Camera view 0 (b) Camera view 1
(c) Camera view 2 (d) Camera view 3
(e) The resulting model

Fig. 1 Input binary silhouette images and the generated visual hull model. Our test dataset consists of 20 binary silhouette images of $1,280 \times 720$ pixels per time frame with respective camera calibration parameters. Figures (a) to (d) show four selected input images for a given time frame, and (e) displays the created visual hull model.

2.1 **Phase 1:** Extraction of smooth piecewise-linear silhouette contours

Unlike the previous interactive visual hull techniques that are based on a simple binary classification of projected voxels, our method initially constructs smooth silhouette contours on the fly from the input binary silhouette images for a more refined classification that markedly reduces bumpy artifacts on the surface of the resulting visual hull.

The first step in our GPU accelerated scheme is to apply a Gaussian filter of given size to each input binary image, where we use the recursive filtering technique [3] that is easily implemented with separable horizontal and vertical convolutions. To improve the filtering efficiency, the binary image is partitioned into tiles of $m \times m$ pixels (for our test images having resolution of $1,280 \times 720$ pixels, the best performance was observed in our parallel implementation when 32×32 tiles were used). A CUDA kernel is then executed over the tiles to see if a given tile is on the border, i.e. if it contains both black and white pixels, marking the boundary tile and its eight neighboring tiles as valid. Then, the actual Gaussian filtering is performed tile by tile by a second kernel, where the convolution operations are carried out only with respect to the valid tiles. In our experimentation, this border-tile-only filtering strategy resulted in significant speedup over a simple CUDA implementation, performing the convolutions against the entire image pixels because the input binary images often possess a high degree of spatial coherence as those in Fig. 1(a) to (d).

As a result of the smoothing process, the input binary images are converted to grayscale images whose pixel values now vary from zero (inside) to one (outside). For efficient GPU processing in later stages, this first phase produces a 2D array for every input image, each of whose elements corresponds to a *square* region, formed by four adjacent pixels of the filtered silhouette image (note that the square in the screen space is the 2D version of the cell in the 3D volume space). For this, another group of parallel threads are spawned, one for each square, where each thread classifies the four corners of the assigned square using a given threshold value, i.e. a given iso-value. When all the four intensity values are less than (greater than) the iso-value, the square is simply marked *inner* (*outer*). Otherwise, it is marked *boundary*, and stored with a line segment (or segments) extracted through linear interpolation using a simple 2D version of the marching cubes algorithm (see Fig. 2).

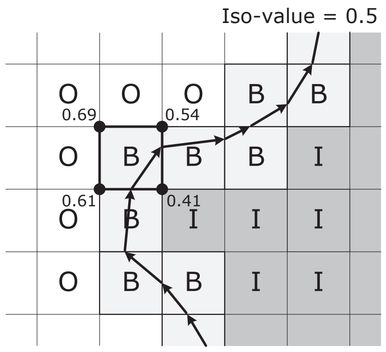


Fig. 2 Square classification and extraction of piecewise-linear silhouette contour. Each square region, made of four adjacent pixels of a Gaussian-filtered silhouette image, is classified as inner (I), outer (O), or boundary (B) according to the pixels' intensities and a given threshold value. For a boundary square, an oriented line segment (or segments) is additionally stored so that the 3D voxel classification in the second phase of our method can be made efficiently.

The extracted line segments together form a smooth piecewise-linear silhouette contour. However, it should be emphasized that the connection information between line segments is not recorded explicitly while they are independently generated by parallel threads on the GPU. Instead, we store each line segment with an *orientation* in the boundary square such that the interior region always locates on the right side, which allows an easy in/out classification for the boundary square region. Once this 2D inner/outer/boundary classification is over for each input silhouette image, in the next stage, the decision whether a voxel in the 3D volume space is contained in a silhouette cone, generated by a given input image, can be made efficiently by projecting the voxel onto the corresponding silhouette image and checking, using the simple 2D classification data, if the projected voxel resides in the interior area of the extracted silhouette contour.

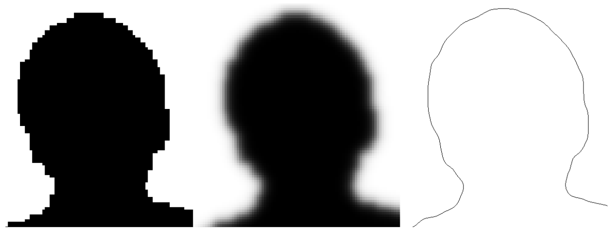
Fig. 3 shows three example sets of a binary silhouette, a Gaussian-filtered silhouette, and an extracted contour (from left to right, respectively), where it is demonstrated that the Gaussian filter successfully removed the bumpy silhouettes in the original binary images to generate smooth but feature preserving silhouette contours.

2.2 **Phase 2:** Construction of compact mesh structure

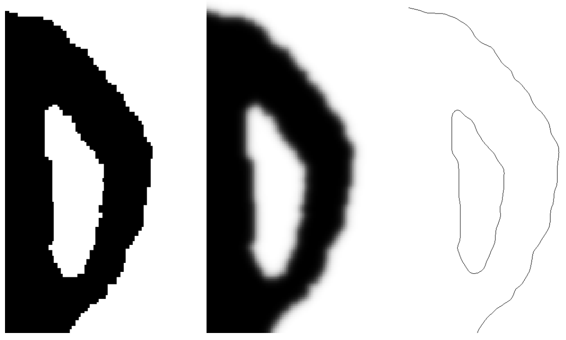
Given the 2D classification data for extracted silhouette contours, we start to build a visual hull model. We aim to represent the model compactly in the *indexed face set* format, where the triangular mesh structure consists of a simple list of vertex coordinates and a list of triangles that index the vertices they use. In this phase, the mesh structure is constructed only partially using temporary vertex information (refer to Fig. 8), and the actual vertex coordinates of the model are computed in the final phase.

The first step in this phase is to identify the *boundary* cells with which the visual hull surface intersects. For efficient GPU computation, we first linearize the 3D voxel grid into a one-dimensional (1D) array called a *voxel array (VA)* with a simple address calculation, in which chunks of contiguous voxel elements form CUDA blocks of threads (see Fig. 4). Each thread executes a voxel classification kernel which projects the corresponding voxel onto input silhouette images and uses the 2D inner/outer/boundary classification information to check whether it resides inside of the respective silhouette contours. Since the visual hull is the intersection of all silhouette cones, the voxel is marked *inner* only if it is found to exist within all silhouette contours.

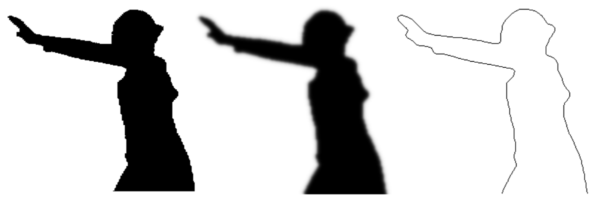
Similarly, the 3D cells in the volume space are enumerated linearly into a 1D array called a *boundary cell array (BCA)*, which is for marking the boundary cells. We also allocate another array of the same dimension, called a *triangle count array (TCA)*, for remembering the number of triangles generated in each boundary cell. A cell classification kernel is then executed by each



(a) Example 1



(b) Example 2



(c) Example 3

Fig. 3 Construction of smooth piecewise-linear silhouette contours (left and center: binary and filtered silhouette images, right: extracted contour). We find that, for input binary images of $1,280 \times 720$ pixels, a 7×7 Gaussian filter with threshold value 0.5 usually enables to generate satisfactory contours.

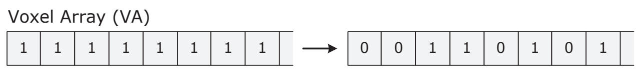


Fig. 4 Inner/outer classification of voxels. In the beginning of the second phase, the voxels in the 3D volume space, linearized into a 1D array on the GPU memory, are classified as *inner* (1) or *outer* (0) according to whether they are inside of the visual hull, i.e. whether they are inside of all the silhouette cones, generated by the extracted silhouette contours.

thread that, using the information in the VA, classifies the cell it handles and marks the corresponding BCA element if it intersects the visual hull surface, i.e. if the inner/outer classifications of the eight incident voxels do not coincide. In that case, the thread additionally calculates how many triangles are created in the cell, and stores the count in the corresponding TCA element. At this moment, only the triangle count is calculated quickly by referring to the marching cubes table without generating the actual triangles. Fig. 5 illustrates an example in which the second, fourth, fifth, and sixth cells (counting from zero) are classified as boundary cells and their triangle counts are accordingly stored (see the second row).

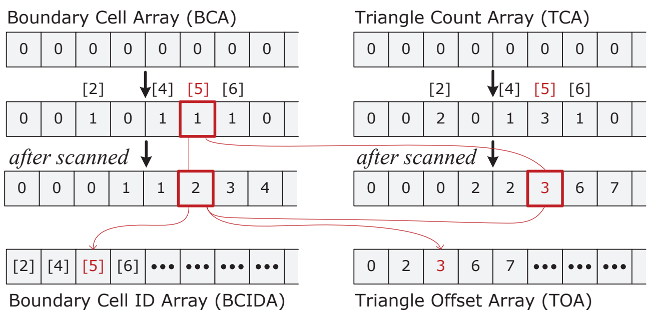


Fig. 5 Extraction of boundary cell information. Using the voxel classification information in the VA, each cell of the 3D grid, initially linearized into a 1D array on the GPU memory, is first checked if it is a boundary cell. If it is, the number of triangles that will be created in the cell by the marching cubes algorithm is also stored (the two arrays in the second row). Then, through the help of the exclusive-scan operation, only the boundary cells are packed into an array (BCIDA). At the same time, the accumulated number of triangles created before a current boundary cell is also recorded in an extra array (TOA) for a proper address calculation in a later stage.

We next perform a data-parallel prefix sum (scan) operation [6] on both the BCA and the TCA. Note that an application of the *exclusive scan* operation on a sequence $(a_0, a_1, a_2, \cdots)$ returns another sequence $(0, b_1, b_2, \cdots)$ such that $b_i = \sum_{j=0}^{i-1} a_j$, $i = 1, 2, \cdots$. So, for a boundary cell, the scanned BCA contains the number of boundary cells that precede it in the cell enumeration, and hence the offset for storing the boundary cell's ID in the compacted *boundary cell ID array (BCIDA)*. Likewise, the corresponding element of the scanned TCA indicates the number of all triangles generated in the preceding boundary cells. By storing this information using the same offset in another array called a *triangle offset array (TOA)*, the triangles from the boundary cell can be generated and stored in the proper location by a parallel CUDA thread in a later stage. See an example in Fig. 5, where the ID of the second boundary cell in the *BCA* is 5, and its three triangles should be stored in the triangle list of the mesh structure, starting from the third element.

We are now prepared to fill in the triangle and the vertex lists of the triangular mesh structure. Fig. 6 illustrates a list of triangles, called a *triangle list (TL)*, that initially contains a sequence of triples, one per triangle, in which each (i, j), encoded in a 32-bit unsigned integer, indicates the jth vertex of the ith triangle. In addition, we use another list called an *edge ID list (EIDL)*; its elements, mapped one to one with those of the TL, will hold temporary vertex information. Here, a trian-

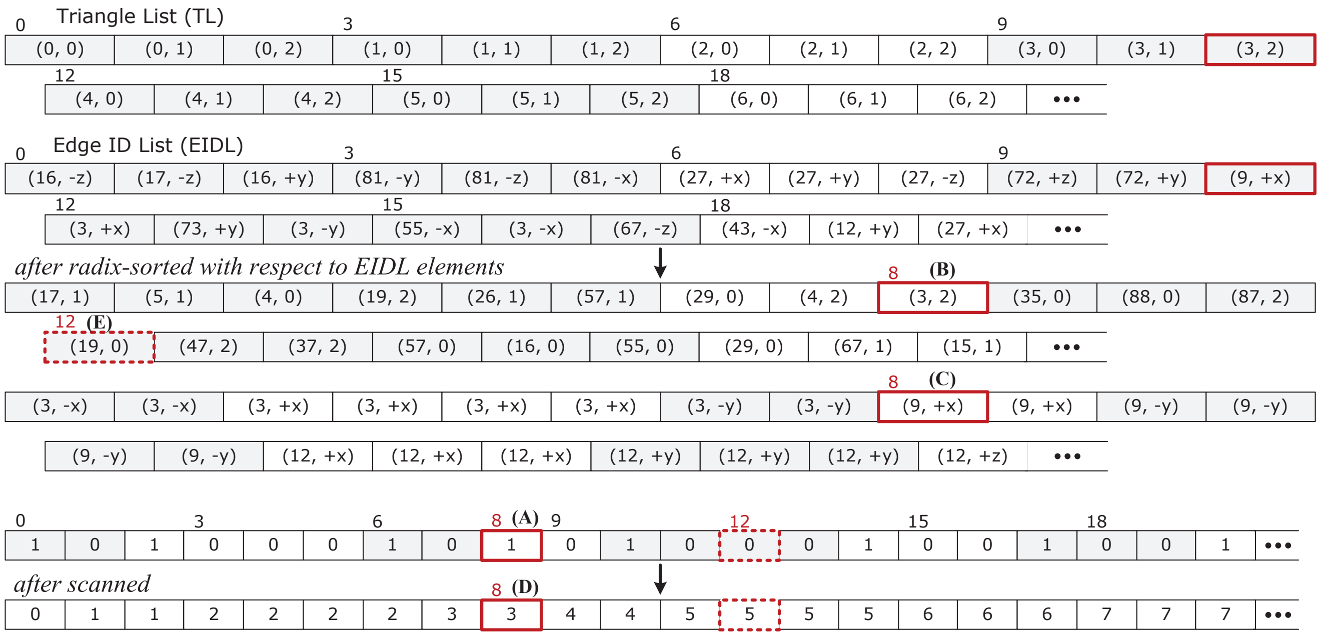


Fig. 6 GPU-based removal of duplicate vertices generated by the marching cubes algorithm. Here, the 'on' flag of the eighth element of the extra array (marked by (A)) indicates that the second vertex of the third triangle (B) is $(9, +x)$ (C), and should be stored in the third slot (D) of the redundancy-free vertex list of the compact triangular mesh structure. The 0/1 array in the below indicates the head vertex of each redundant vertex group. Note that the index of a temporary vertex to the vertex list can be calculated by subtracting one from the sum of the corresponding values in the 0/1 array and its scanned sequence. For instance, the index of the vertex $(19, 0)$ in the twelfth slot of the sorted TL (E) is 4 $(= 0 + 5 - 1)$.

gle's vertex that exists along an edge of a boundary cell is temporarily denoted by a pair made of the ID of the inner voxel and the axis direction from the base voxel. For instance, the triangle in Fig. 7 is represented by a triple of vertices: $((16, -z), (17, -z), (16, +y))$.

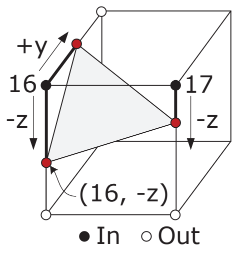
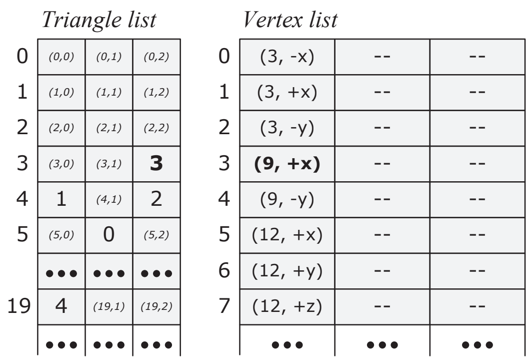


Fig. 7 Temporary vertex representation. A vertex along an edge of a boundary cell, whose exact location is not known yet, is temporarily denoted by the ID of the inner voxel and the axis direction from the voxel. Its actual coordinates are calculated in the third phase.

Then, each CUDA thread, launched one per boundary cell in the BCIDA, again refers to the marching cubes table to compute the temporary vertex pairs of all triangles that are created from the cell, storing them in the proper place in the EIDL using the offset information found in the TOA. After that, the TL and EIDL are simultaneously sorted by the EIDL's values, which have been encoded in a 32-bit unsigned integer, using the parallel radix sort algorithm [13]. As a result, the duplicate vertices, shared by adjacent triangles, are placed in a contiguous region of the EIDL with corresponding vertex identifications in the TL. Now, a CUDA thread, generated one per sorted EIDL element, checks whether the assigned element's vertex is the first one in the same vertex group (this can be done easily by comparing with the preceding element in the EIDL), and marks the result in an extra array. This 0/1 array, together with an additional sequence obtained through an exclusive scan, provides the offset information with which the locations of (temporary) vertices in the redundancy-free vertex list of the mesh structure are easily determined (refer to Fig. 6 again to see how to decide the index of an arbitrary vertex to the compacted vertex list).

Finally, a CUDA kernel is executed on the threads spawned with respect to the sorted TL elements, where, for the corresponding vertex (i, j), each thread calculates the index, d, to the vertex list of the triangular mesh (*Vertex list* in Fig. 8), and records d as the jth index of the ith triangle in the triangle list (*Triangle list*). Also, to reduce memory bandwidth consumption, only the thread for the head vertex, i.e. the thread corresponding to the first one in the same vertex group, stores the temporary vertex information of the vertex (i, j) in the x-coordinate field of the dth vertex in the



Triangle list

0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)
3	(3,0)	(3,1)	**3**
4	1	(4,1)	2
5	(5,0)	0	(5,2)
	•••	•••	•••
19	4	(19,1)	(19,2)
	•••	•••	•••

Vertex list

0	(3, -x)	--	--
1	(3, +x)	--	--
2	(3, -y)	--	--
3	**(9, +x)**	--	--
4	(9, -y)	--	--
5	(12, +x)	--	--
6	(12, +y)	--	--
7	(12, +z)	--	--
	•••	•••	•••

Fig. 8 The compact triangular mesh structure in the indexed face set representation after the second phase of our method. In the following third stage, a thread is spawned for each x-component value of *Vertex list* to compute the actual xyz coordinates, and store them in the corresponding slot, completing the construction of a compact triangular mesh structure. Note that *Triangle list* in this figure has only been partially filled with the example data from Fig. 6.

vertex list. For example, in Fig. 6, the index of the vertex $(3, 2)$ in the eighth slot of the sorted TL (B) is 3 $(= 1 + 3 - 1)$. Hence, the second index (counting from zero) of the third triangle becomes 3 (see Fig. 8). Also, since the vertex is a head vertex, the corresponding thread stores its temporary vertex information $(9, +x)$ (C) in the x component of the third vertex of the vertex list.

2.3 **Phase 3:** Efficient generation of exact vertex coordinates using the idea of perspective correction

To complete the construction of the compact mesh structure, a parallel thread, spawned with respect to each temporary vertex information in the x component of the vertex list, finds the actual location of the corresponding vertex, and overrides the temporary vertex information with its xyz coordinate vector. In this process, we exploit the idea of perspective correction, which has been effectively applied in 3D graphics for correct texture mapping, to efficiently estimate the exact intersection in the world space, i.e. in the volume space, between an edge of a boundary cell and a silhouette cone formed by an extracted silhouette contour.

For the edge containing the temporary vertex of a current thread, let $\mathbf{p}_c^i = (x_c^i \; y_c^i \; z_c^i)^t$ and $\mathbf{p}_c^o = (x_c^o \; y_c^o \; z_c^o)^t$ be the camera space coordinates of its two end voxels $\mathbf{p}_w^i$ and $\mathbf{p}_w^o$ in the world space, between which the intersection $\mathbf{p}_w^s = \alpha \cdot \mathbf{p}_w^o + (1-\alpha) \cdot \mathbf{p}_w^i$ is to be computed (see Fig. 9). For each camera view, the thread projects $\mathbf{p}_w^i$ and $\mathbf{p}_w^o$ into the screen space where the input silhouette image exists, and finds the exact intersection $\mathbf{p}_s^s$ between the projected edge $(\mathbf{p}_s^i, \mathbf{p}_s^o)$ and the piecewise-linear silhouette curve using an extended version of Bresenham's line-drawing algorithm [2], which reveals the ratio $t = \frac{|\overline{\mathbf{p}_s^i \mathbf{p}_s^s}|}{|\overline{\mathbf{p}_s^i \mathbf{p}_s^o}|}$. Because of the perspective projection, t is in general different from the needed ratio α. In fact, it can be shown that $\alpha = \frac{t/z_c^o}{t/z_c^o + (1-t)/z_c^i} = \frac{tz_c^i}{tz_c^i + (1-t)z_c^o}$, which requires only two additions, two multiplications, and one division to calculate from t (refer to Appendix for the correctness of the formula and Fig. 10 for a pseudocode for computing the α value).

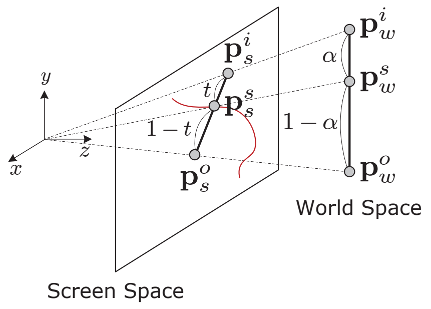


Fig. 9 Efficient computation of exact intersection in the world space. For a given camera view, the intersection $\mathbf{p}_w^s$ in the world space between an edge of a boundary cell, $(\mathbf{p}_w^i, \mathbf{p}_w^o)$ and the corresponding silhouette cone can only be calculated by finding the intersection $\mathbf{p}_s^s$ in the screen space between the projected edge $(\mathbf{p}_s^i, \mathbf{p}_s^o)$ and the piecewise-linear silhouette contour (the curve on the plane), extracted in the first phase of our method.

```
Input: two voxel coordinates Pi_w & Po_w in volume
   space and a camera ID.
Output: alpha, the distance ratio from Pi_w to the
   point Ps_w on the silhouette cone.
Begin
  SI := the current silhouette image;
  Pi_s := the projection of Pi_w onto SI;
  Po_s := the projection of Po_w onto SI;
  March from Pi_s to Po_s in SI until a boundary
      square BS is met;
  Ps_s := the intersection between the BS's
      line segment(s) and (Pi_s, Po_s);
  t := the distance ratio from Pi_s to Ps_s;
  zi_c := the z coordinate of Pi_w in camera space;
  zo_c := the z coordinate of Po_w in camera space;
  tmp := t*zi_c;
  alpha := tmp/(tmp + (1-t)*zo_c);
End
```

Fig. 10 Pseudocode for the computation of the α value for a given camera view.

Then, taking the smallest of the α values from all camera views, we can locate the intersection point on the visual hull boundary. It should be mentioned that the exact intersection was also proposed in previous work based on a matrix computation [9]. Our technique finds the same intersection with fewer floating-point operations, which allows a marked performance enhancement for nontrivial numbers of cameras and volume resolutions.

3 Implementation results

3.1 Construction of visual hull meshes from input silhouette images

We implemented our GPU algorithm using the CUDA API [12], and evaluated its performance using several example images, generated by the real-time 3D modeling system at the Electronics and Telecommunications Research Institute in Korea. Fig. 11 shows the visual hull models constructed from three representative test datasets, named Woo, Bboy, and Girl, respectively. Each dataset consists of 20 binary silhouette images of $1,280 \times 720$ pixels with the corresponding camera calibration parameters.

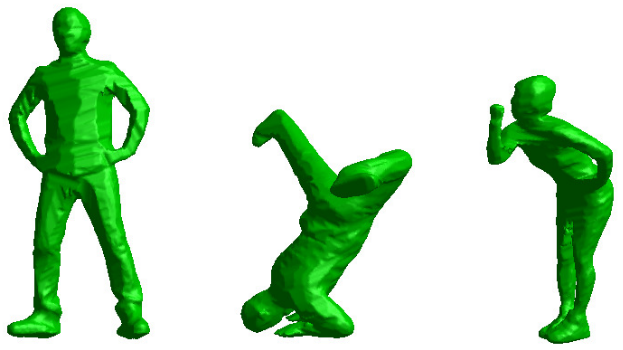


Fig. 11 Visual hull models created from three test datasets: Woo, Bboy, and Girl (from left to right).

Table 1 shows statistics measured on an NVIDIA GeForce GTX 580 GPU with 1.5 GB of graphics memory with respect to three different volume resolutions, where each row reveals the size of the produced triangular mesh, represented in the indexed face set, and the total GPU time along with the relative overhead of the three computational phases. Here, the figure in parentheses in the 'Vertices' column denotes the value of three times the number of faces divided by the number of vertices, which indicates the degree of vertex redundancy in the simple mesh representation that simply enumerates the vertices of triangles. Interestingly, the observed ratios, including those from our marching cubes implementation (refer to Table 3) were quite consistent, and the removal of vertex duplication in the mesh representation resulted in the time- and space-efficient GPU implementation.

The timing results show that the computation time taken by our method greatly depends on the voxel resolution. When a lower-resolution volume grid was selected, the first phase for extracting a smooth silhouette contour required a relatively significant period of the time. However, as the volume resolution increased, the GPU implementation became dominated by the second phase since the numbers of voxels and cells to be processed increased with the cube of the volume resolution. On the other hand, the third phase of finding the exact vertex locations consumed only a moderate amount of time because of our efficient calculation framework that only processed a relatively small number of boundary cells. In particular, launching one CUDA thread per one and only one unique temporary vertex in the compacted vertex list avoided unnecessary thread divergence, leading to an increase in the GPU occupancy during the vertex coordinate calculation.

Recall that the two major computations carried out in the first phase are the application of the Gaussian filter and the extraction of the piecewise-linear silhouette contour, which implies that the computation time of this phase is basically dependent on the image resolution and the complexity of the contour curve. As can be seen in the timings in the 'Phase I' columns of Table 1 and 2, relatively small amounts of time were spent applying the 7×7 Gaussian filter to 20 binary images of $1,280 \times 720$ pixels thanks to our GPU accelerated filtering scheme that applies the separated convolution only to the pixels neighboring to the silhouette contour. In particular, when a nontrivial volume resolution (e.g. $256 \times 256 \times 256$) was chosen, the additional cost for the Gaussian filtering was just small compared to the entire computation time.

It should be mentioned that, while the visual hull construction speed was slightly improved without the Gaussian filtering, the gain only came with unsightly bumpy artifacts on the visual hull surfaces. When no smoothing filter was applied, the extracted silhouette contour almost coincided with the boundary of the binary silhouette. Hence, the aliases on the surfaces became unavoidable, although the exact intersections between the respective silhouette cones and the edges of the 3D cells were computed in the third phase. Fig. 12 compares the outputs produced for the volume resolution of $256 \times 256 \times 256$ voxels without and with the Gaussian filter applied, in which we clearly observed that the undesirable surface effects were nicely smoothed out through the refined voxel classification. We also observed that the elaborate extraction of the piecewise-linear silhouette contours created less visual artifacts

Dataset	Volume resolution	Size of mesh		Computation time (ms)			
		Vertices	Faces	Phase I	Phase II	Phase III	Total
Woo	64^3	3,222 (6.004)	6,448	12.03 (74.9%)	3.12 (19.4%)	0.92 (5.7%)	16.07
	128^3	13,118 (6.000)	26,236	12.12 (60.7%)	6.78 (34.0%)	1.06 (5.3%)	19.96
	256^3	53,194 (5.999)	106,376	12.04 (29.5%)	26.83 (65.7%)	1.97 (4.8%)	40.84
Bboy	64^3	2,986 (5.996)	5,968	12.43 (75.9%)	2.96 (18.1%)	0.99 (6.0%)	16.38
	128^3	12,302 (6.000)	24,604	12.40 (61.5%)	6.65 (33.0%)	1.12 (5.5%)	20.17
	256^3	50,010 (6.000)	100,016	12.41 (29.7%)	27.44 (65.7%)	1.92 (4.6%)	41.77
Girl	64^3	2,290 (6.000)	4,580	11.55 (78.8%)	2.25 (15.4%)	0.85 (5.8%)	14.65
	128^3	9,368 (6.000)	18,736	11.48 (61.9%)	6.12 (33.0%)	0.95 (5.1%)	18.55
	256^3	38,104 (6.000)	76,208	11.53 (29.8%)	25.70 (66.4%)	1.47 (3.8%)	38.70

Table 1 Performance statistics on our GPU accelerated visual hull construction method. Three different datasets, each made of 20 binary silhouette images of $1,280 \times 720$ pixels, were tested with respect to three volume resolutions, where a 7×7 Gaussian filter was applied in Phase I. The numbers of vertices and faces of the generated triangular meshes represented in the indexed face set format are shown. The figures in parentheses in the 'Vertices' column denote the ratios 3*(# of faces)/(# of vertices), indicating the degree of redundancy of vertices in the simple mesh representation, in which the vertex coordinates of extracted triangles are simply enumerated with the same vertex repeated. The phase-by-phase dissection of computation times required by the GPU is also provided.

Dataset	Volume resolution	Size of mesh		Computation time (ms)			
		Vertices	Faces	Phase I	Phase II	Phase III	Total
Woo	64^3	2,972 (6.004)	5,948	10.98 (74.3%)	2.99 (20.2%)	0.81 (5.5%)	14.78
	128^3	12,476 (6.000)	24,952	10.98 (60.2%)	6.26 (34.3%)	1.01 (5.5%)	18.25
	256^3	51,520 (5.998)	103,012	10.97 (27.9%)	26.48 (67.2%)	1.93 (4.9%)	39.38
Bboy	64^3	2,816 (5.979)	5,612	10.97 (74.7%)	2.90 (19.7%)	0.82 (5.6%)	14.69
	128^3	11,660 (5.990)	23,280	10.96 (58.6%)	6.64 (35.5%)	1.1 (5.9%)	18.70
	256^3	47,858 (5.996)	95,660	10.97 (27.6%)	26.86 (67.7%)	1.86 (4.7%)	39.69
Girl	64^3	2,136 (6.000)	4,272	11.00 (78.5%)	2.27 (16.2%)	0.75 (5.3%)	14.02
	128^3	8,864 (6.001)	17,732	11.00 (60.6%)	6.15 (33.9%)	1.01 (5.5%)	18.16
	256^3	36,574 (6.000)	73,144	11.01 (29.3%)	25.11 (66.9%)	1.44 (3.8%)	37.56

Table 2 Performance statistics without a Gaussian filter applied. As can be identified in the computation times taken by the first phases of the two implementations with and without the application of the Gaussian filter, the computational burden of the Gaussian filtering was alleviated significantly through the parallel convolution computation only on the border regions of the silhouette contours. Note that, when no Gaussian filter was applied, somewhat smaller triangular meshes were produced in slightly less time, but only at the expense of ugly looking artifacts on the surfaces of the visual hull objects.

when the reconstructed objects were rendered with various texture images.

3.2 Application to the marching cubes algorithm

The presented GPU technique for directly generating compact triangular meshes in the form of indexed face set can be easily modified for implementing the frequently used, marching cubes algorithm [10] on the GPU. Given a volumetric dataset and an iso-value, the second phase of our method builds, as before, the triangle index list and the temporary vertex list, initially containing the vertex offset information. Then, in the following stage, the actual vertex coordinates (optionally with normal coordinates) are generated by simple linear interpolation along the edges of boundary cells, storing them in the corresponding locations of the compact vertex array.

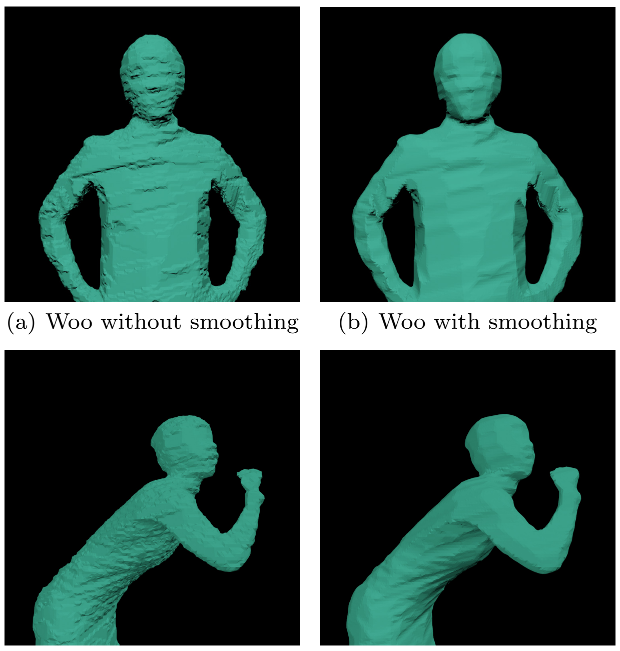
The experimental results in Table 3, obtained with respect to four volumetric datasets (see Fig. 13) with two different volume resolutions, compare our GPU implementation with a conventional implementation which simply classifies boundary cells and lists the vertex and normal coordinates of each triangle extracted from them. Obviously, our implementation is more complicated than the simple implementation because ours should go through an additional GPU stage that builds the indexed face set structure. Interestingly, however, our method turned out to be faster significantly on the NVIDIA GeForce GTX 580 GPU as demonstrated in the table.

Notice that, in the NVIDIA's Fermi architecture, memory operations are issued per warp (32 threads), and it is critical to the performance of CUDA applications to have each warp access global memory as coalesced as possible. The major difference between the two GPU implementations is the amount and locality of the

(Time: ms, Memory: MB)

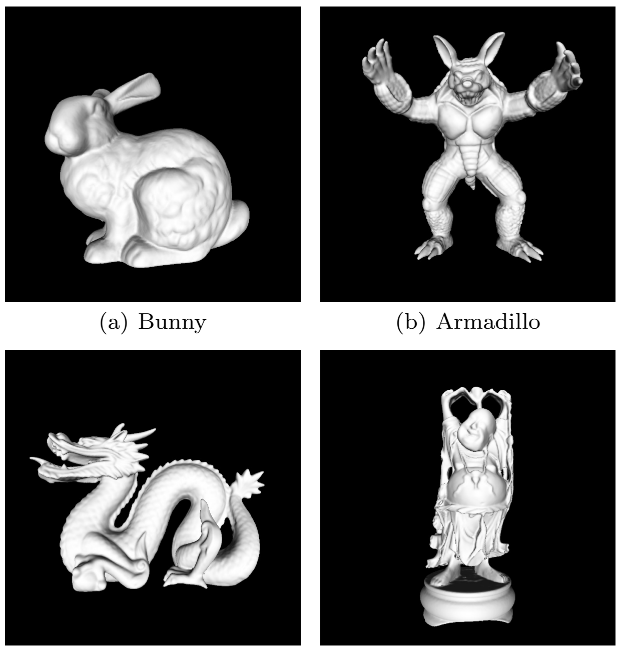
Dataset	Volume resolution	Size of mesh		Ours		Conventional	
		Vertices	Faces	Time	Memory	Time	Memory
Bunny	128^3	33,392 (5.974)	66,491	4.84	1.53	7.22	4.57
	256^3	132,883 (5.987)	265,194	15.58	6.08	29.20	18.21
Armadillo	128^3	23,528 (5.999)	47,052	4.36	1.08	7.18	3.23
	256^3	95,004 (6.000)	190,004	15.50	4.35	28.77	13.05
Dragon	$128^2 \times 256$	41,473 (5.979)	82,659	7.14	1.90	12.50	5.68
	$256^2 \times 512$	167,684 (5.990)	334,796	26.62	7.67	55.94	22.99
Happy Buddha	$128^2 \times 256$	31,833 (5.990)	63,565	6.87	1.46	12.44	4.37
	$256^2 \times 512$	129,815 (5.994)	259,361	26.12	5.94	55.56	17.81

Table 3 Statistics on the two GPU implementations for the marching cubes algorithm. In this table, 'Ours' represents the implementation produced based on the presented GPU technique, while 'Conventional' corresponds to the classic implementation that simply lists the vertex and normal coordinates of extracted triangles with the same vertex information repeated in the representation. Again, the figures in parentheses in the 'Vertices' column indicate the degree of vertex redundancy (refer to Table 1 for an explanation of these values).


(a) Woo without smoothing (b) Woo with smoothing

(c) Girl without smoothing (d) Girl with smoothing

Fig. 12 Effect of the sophisticated voxel classification. To clearly see the effect of the smooth contour extraction carried out in the first phase, the triangular meshes are rendered with flat shading such that the individual triangles are visible.


(a) Bunny (b) Armadillo

(c) Dragon (d) Happy Buddha

Fig. 13 Triangular meshes produced by the marching cubes algorithm from four test datasets.

mesh data that each warp must write in parallel into the global memory of the GPU. In the final stage of the conventional implementation, each thread of a warp writes the coordinate data of all triangles extracted from a boundary cell assigned to it. While the most efficient situation for the Fermi architecture is that each warp requests 32 aligned, consecutive 4-byte words within a single, aligned 128 byte-long segment of global memory, there is a high probability that the memory access from the warps are very scattered in the conventional implementation, leading to a marked performance decrease, since each triangle consumes 72 bytes (6 floats per vertex and 3 vertices per triangle). In contrast, in the new implementation, the global memory access is relatively less scattered as only the coherent vertex array region is accessed in this stage, enabling a better timing performance in spite of the additional computation for building the indexed face set structure. This experimental result strongly implies that constructing compact visual hull meshes is equally important for efficient GPU processing.

4 Concluding remarks

We have presented an effective parallel volumetric visual hull construction algorithm that employs a novel technique for creating a compact triangular mesh of the reconstructed visual hull model as well as several optimization techniques for producing a time- and space-efficient GPU implementation. To the best of our knowledge, our computation scheme is the first parallel algorithm that, fully run on the GPU, generates smooth high-resolution visual hull meshes in compact form, based on a refined voxel classification.

We have also shown that the presented GPU technique allows an easy modification for another important problem, that is, the GPU implementation of the marching cubes algorithm. Through our experiments, it was demonstrated that, with the current GPU architecture, it is undoubtedly worthwhile to develop GPU schemes that facilitate compact data representation by reducing wasteful data redundancy. Of course, this statement is also true when visual hull meshes are to be constructed on the GPU.

Acknowledgements This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant no. 2012R1A1A2008958), and by the strategic technology development program of MCST/MKE/KEIT (Development of Full 3D Reconstruction Technology for Broadcasting Communication Fusion (KI001798)).

References

1. Baumgart, B.: Geometric modeling for computer vision. Ph.D. thesis, Stanford University (1974)
2. Bresenham, J.: Algorithm for computer control of a digital plotter. IBM Systems Journal **4**(1), 25–30 (1965)
3. Deriche, R.: Recursively implementing the Gaussian and its derivatives. Unité de Recherche INRIA-Sophia Antipolis, Tech. Rep. No. 1893 (1993)
4. Erol, A., Bebis, G., Boyle, R., Nicolescu, M.: Visual hull construction using adaptive sampling. In: Proc. of the 7th IEEE Works. on Application of Computer Vision, vol. 1, pp. 234–241 (2005)
5. Franco, J.S., Boyer, E.: Exact polyhedral visual hulls. In: Proc. of British Machine Vision Conf., pp. 329–338 (2003)
6. Harris, M.: Parallel prefix sum (scan) with CUDA. In: H. Nguyen (ed.) GPU Gems 3, chap. 39, pp. 851–876. Addison Wesley (2008)
7. Ladikos, A., Benhimane, S., Navab, N.: Efficient visual hull computation for real-time 3D reconstruction using CUDA. In: Proc. of the Conf. on Computer Vision and Pattern Recognition Works., pp. 1–8 (2008)
8. Laurentini, A.: The visual hull concept for silhouette-based image understanding. IEEE Trans. PAMI **16**(2), 150–162 (1994)
9. Liang, C., Wong, K.Y.: Exact visual hull from marching cubes. In: Proc. of the 3rd Int. Conf. on Computer Vision Theory and Applications, vol. 2, pp. 597–604 (2008)
10. Lorensen, W., Cline, H.: Marching Cubes: A high resolution 3D surface construction algorithm. Proc. of ACM SIGGRAPH **21**, 163–169 (1987)
11. Matusik, W., Buehler, C., McMillan, L.: Polyhedral visual hulls for real-time rendering. In: Proc. of the 12th Eurographics Works. on Rendering Techniques, pp. 115–126 (2001)
12. NVIDIA: NVIDIA CUDA C Programming Guide (Version 3.2) (2010)
13. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proc. of the 2009 IEEE Int. Symp. on Parallel & Distributed Processing, pp. 1–10 (2009)
14. Shujun, Z., Cong, W., Xuqiang, S., Wei, W.: Dream World: CUDA-accelerated real-time 3D modeling system. In: Proc. of the IEEE Int. Conf. on Virtual Environments, Human-Computer Interfaces and Measurement Systems, pp. 168–173 (2009)
15. Slabaugh, G., Culbertson, B., Malzbender, T., Schafer, R.: A survey of methods for volumetric scene reconstruction from photographs. In: Proc. of Int. Works. on Volume Graphics, pp. 81–100 (2001)
16. Waizenegger, W., Feldmann, I., Eisert, P., Kauff, P.: Parallel high resolution real-time visual hull on GPU. In: Proc. of the 16th IEEE Int. Conf. on Image Processing, pp. 4301–4304 (2009)

Appendix: Proof of perspective-corrected ratios

We want to reveal the relation between the points $\mathbf{p}_w^i$, $\mathbf{p}_w^s$, $\mathbf{p}_w^o$ in the world space and the mapped points $\mathbf{p}_s^i$, $\mathbf{p}_s^s$, $\mathbf{p}_s^o$ in the screen space (see Fig. 9 again). Note that the transformation from the normalized image space to the screen space is an affine transformation because it involves only translation, scaling, and possibly shearing. So is the view transformation that converts points from the world space to the camera space. Since the affine transformations preserve the ratios of distance along a line, it is enough to consider the mapping between the corresponding points $\mathbf{p}_c^i$, $\mathbf{p}_c^s$, $\mathbf{p}_c^o$ in the camera space and $\mathbf{p}_n^i$, $\mathbf{p}_n^s$, $\mathbf{p}_n^o$ in the normalized image space (see Fig. 14).

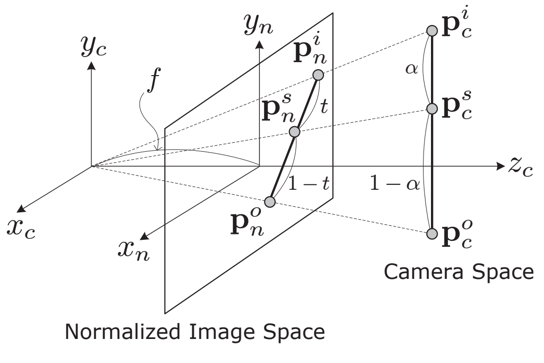


Fig. 14 Mapping between the camera space and the normalized image space.

Assume that $\mathbf{p}_c^i = (x_c^i\ y_c^i\ z_c^i)^t$, $\mathbf{p}_c^s = (x_c^s\ y_c^s\ z_c^s)^t$, $\mathbf{p}_c^o = (x_c^o\ y_c^o\ z_c^o)^t$, where

$$\mathbf{p}_c^s = \alpha \begin{pmatrix} x_c^o \\ y_c^o \\ z_c^o \end{pmatrix} + (1-\alpha) \begin{pmatrix} x_c^i \\ y_c^i \\ z_c^i \end{pmatrix}.$$

Similarly, let $\mathbf{p}_n^i = (x_n^i\ y_n^i)^t$, $\mathbf{p}_n^s = (x_n^s\ y_n^s)^t$, $\mathbf{p}_n^o = (x_n^o\ y_n^o)^t$. When $\mathbf{p}_c^s$ is transformed into the normalized image space by multiplying the perspective transformation matrix, we get

$$\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_c^s = \begin{pmatrix} f\alpha x_c^o + f(1-\alpha)x_c^i \\ f\alpha y_c^o + f(1-\alpha)y_c^i \\ \alpha z_c^o + (1-\alpha)z_c^i \end{pmatrix},$$

which, via perspective division, leads to

$$\mathbf{p}_n^s = \frac{f\alpha}{\alpha z_c^o + (1-\alpha)z_c^i} \begin{pmatrix} x_c^o \\ y_c^o \end{pmatrix} + \frac{f(1-\alpha)}{\alpha z_c^o + (1-\alpha)z_c^i} \begin{pmatrix} x_c^i \\ y_c^i \end{pmatrix}.$$

By the same perspective transformation, it becomes that $(x_n^i\ y_n^i)^t = \left(\frac{f x_c^i}{z_c^i}\ \frac{f y_c^i}{z_c^i}\right)^t$ and $(x_n^o\ y_n^o)^t = \left(\frac{f x_c^o}{z_c^o}\ \frac{f y_c^o}{z_c^o}\right)^t$. From these, we obtain that

$$\begin{aligned} \mathbf{p}_n^s &= \frac{\alpha z_c^o}{\alpha z_c^o + (1-\alpha)z_c^i} \begin{pmatrix} x_n^o \\ y_n^o \end{pmatrix} + \frac{(1-\alpha)z_c^i}{\alpha z_c^o + (1-\alpha)z_c^i} \begin{pmatrix} x_n^i \\ y_n^i \end{pmatrix} \\ &= \frac{\alpha z_c^o}{\alpha z_c^o + (1-\alpha)z_c^i} \mathbf{p}_n^o + \frac{(1-\alpha)z_c^i}{\alpha z_c^o + (1-\alpha)z_c^i} \mathbf{p}_i^o. \end{aligned}$$

This implies that $t = \frac{\alpha z_c^o}{\alpha z_c^o + (1-\alpha)z_c^i}$, from which we are led to $\alpha = \frac{t z_c^i}{t z_c^i + (1-t) z_c^o}$. □