ICS

TECHNICAL REPORT

Depth-Mesh Objects: Fast Depth-Image Meshing and Warping

Renato Pajarola, Miguel Sainz, Yu Meng

UCI-ICS Technical Report No. 03-02
Department of Information & Computer Science
University of California, Irvine

February 2003

Information and Computer Science
University of California, Irvine

Depth-Mesh Objects: Fast Depth-Image Meshing and Warping

Renato Pajarola*, Miguel Sainz†, Yu Meng*

*Computer Graphics Lab
Information & Computer Science Department
University of California Irvine
pajarola@acm.org, ymeng@ics.uci.edu

†Image Based Modeling and Rendering Lab
Electrical and Computer Engineering Department
University of California Irvine
msainz@ece.uci.edu

Abstract

In this paper we present a novel and efficient depth-image representation and warping technique based on a piece-wise linear approximation of the depth-image as a textured and simplified triangle mesh. We describe the application of a hierarchical triangulation method to generate view-dependent triangulated depth-meshes efficiently from reference depth-images, and propose a new hardware accelerated depth-image rendering technique that supports per-pixel weighted blending of multiple depth-images in real-time. Applications of our technique include image-based object representations and the use of depth-images in large scale walk-through visualization systems.

Keywords: image based rendering, depth-image warping, multiresolution triangulation, level-of-detail, hardware accelerated blending

1. Introduction

In recent years a new rendering paradigm called *Image Based Rendering* (IBR) [DBC+99], that is based on the reuse of image data rather than geometry to synthesize arbitrary views, has attracted growing interest. Since IBR works on sampled image data, and not on geometric scene descriptions, the rendering cost is independent of the scene complexity, and depends only on the resolution of the sampled data. In fact, one of the goals of IBR is to de-couple 3D rendering cost from geometric scene complexity to achieve better display performance in terms of interactivity, frame-rate, and image quality. Target applications include interactive rendering of highly complex scenes, display of captured natural environments, and rendering on time-budgets.

In this paper we expand on the technique of depth-image warping [McM95]. Images with depth(s) per-pixel have been used to represent individual objects [Max96, SGHS98, OB99] or to approximate parts of a large scene in interactive walk-through applications [RP94, SLS+96, SS96, AL99, ACW+99, QWQK00]. We present an improved depth-image warping technique based on adaptive triangulation and simplification of the depth-buffer, and rendering this *depth-mesh* with the color texture of the depth-image (see also [MMB97]). In addition to reducing the rendering cost from the geometric scene complexity to the resolution of the depth-image, adaptively triangulating the depth-map further reduces the rendering cost down to the complexity of the depth-variation within this image.

1.1 Main contributions

Our method offers several improvements and alternatives compared to previous depth-image warping techniques. The main contributions include:

- A technique where the simplification of the triangulated depth-buffer is performed view-dependently at interactive frame-rates for high-resolution depth-images (i.e. with 250,000 pixels or more).
- Depth-image warping is efficiently performed by rendering a comparatively small and bounded set of textured triangle-strips instead of warping a large number of individual pixels.
- A novel technique for hardware accelerated per-pixel positional weighted blending of multiple reference depth-meshes in real-time.

Due to its efficiency, our approach is applicable in various rendering systems such as [RP94, SLS+96, SS96, ACW+99] which update image-based scene representations frequently at run-time.

1.2 Organization

The remainder of the paper is organized as follows. In Section 2 we briefly review the most related methods in depth-image warping. Section 3 describes our depth-meshing and segmentation, Section 4 explains the rendering algorithm, and in Section 5 we provide experimental results supporting our claims. Finally, Section 6 concludes the paper.

2. Related work

The notion of *depth* per pixel has been introduced as disparity between images in [CW93] and used for image synthesis by interpolation between pairs of input images. The depth information – distance from the center of projection along the view direction to the corresponding surface point – allows to re-project pixels from a depth-image to arbitrary new views. In [McM95], a unique evaluation order is presented to guarantee back-to-front drawing order when (forward) warping pixels from the input depth-image to the frame buffer of a new view.

An extension to depth-images is presented in [SGHS98] called a layered depth-image (LDI) which can store multiple depth and color values per pixel. The use of a LDI allows improved depth-image warping with fewer exposure artifacts – exposure of regions not visible in the reference image. The use of precalculated, multi-layer depth-images has previously been discussed in [Max96] for rendering of

complex trees. LDIs are also used in [AL99] together with an automatic preprocess image placement method to support interactive rendering at guaranteed frame rates, and in [OB99] to represent objects using a bounding box of LDIs. The idea of LDIs has further been extended in [CBL99] to LDI-trees for improved control over the sampling rate.

Point based approaches such as [PZvBG00, RL00, ZPvBG01, CN01] eliminate exposure artifacts due to undersampling and zooming in on a fixed set of discrete samples by rendering points as disks, surface elements with non-zero extent. When rendered, the tightly packed surface elements appear to represent a smooth surface.

Another approach to cope with exposure artifacts is to represent depth-images as triangle meshes [MMB97]. The triangulated depth-buffer provides a connected surface approximating the 3D scene and supports automatic pixel interpolation for exposed or stretched regions as well as hardware acceleration warping by rendering the textured triangle mesh. The approach presented in [DSV97, DCV98] creates simplified irregular triangulations of depth-images used as cubical environment maps. Both approaches use multiple reference depth-images to limit exposure artifacts for new views. In [DSSD99] multiple layers of triangulated depth-images are proposed to solve exposure problems.

3. Depth-image meshing

3.1 Overview

The depth values of a depth-image can be considered to be a 2.5-dimensional (projective) height-field data set similar to terrain elevation models. Given the depth values in the z-buffer for a particular reference image, we can calculate for each pixel (i, j) its corresponding 3D coordinate $P_{i,j} \in \mathrm{R}^3$ in the viewing coordinate system. Using the coordinates of points $P_{i,j}$, a quadtree based multiresolution triangulation hierarchy [Paj02] can be constructed on the grid of pixels of the reference depth-image. We call this triangulation of a depth-image a *depth-mesh*, and the representation of an object by multiple depth-image triangulations a *depth-mesh object*. In the following we explain how a single depth-mesh is initialized from a given depth-image, and how an adaptively triangulated depth-mesh is generated at rendering-time.

3.2 Multiresolution triangulation

We use the restricted quadtree triangulation method presented in [Paj98] to generate a simplified triangulation of the z-buffer, called a depth-mesh. Figure 1 shows the basic refinement steps for this hierarchical triangulation method. Based on vertex dependency relations [LKR+96, Paj98] an adaptively refined and crack-free triangle mesh can efficiently be extracted from this multiresolution hierarchy. For basic details on the vertex selection and triangulation process we refer to [LKR+96, Paj98]. The view-dependent vertex selection and triangulation is outlined in Section 3.3.

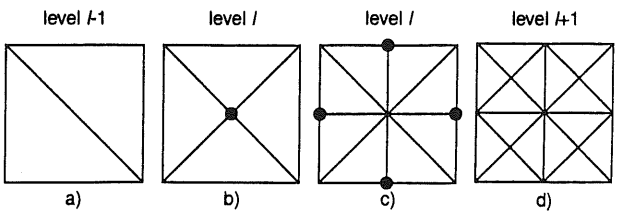

FIGURE 1. Recursive quadtree subdivision and triangulation. Refinement points are shown as grey circles in b) for a diagonal edge bisection and c) for vertical and horizontal edge bisections.

For multiresolution modeling, each depth-mesh point needs to determine its approximation error. As error metric we use a point-to-surface distance as often used for terrains. The refinement point's error is its distance along the elevation axis to the refined edge, thus a point-to-line distance function. For performance reasons, vertical and horizontal refinement points shown in Figure 1 c) use an approximative 2D distance function instead of the more complicated 3D point-to-line distance. The approximation error d of a refinement point p, bisecting a vertical (horizontal) edge between two points a and b of a coarser LOD, is calculated as the 2D distance of p to the line $\overline{ab}$ in the projection on the y,z-plane (x,z-plane). Figure 2 shows an example configuration of projecting points a, b and p from pixels within the same column onto the y,z-plane. Thus using the line equation $z = m \cdot y + b$ for $\overline{ab}$, with $m = (z_b - z_a)/(y_b - y_a)$, the approximation error for vertical refinement points (and analogously for horizontal refinement points) can be evaluated by:

$$d = \frac{m(y_p - y_a) - (z_p - z_a)}{\sqrt{1 + m^2}} \qquad \textbf{(EQ 1)}$$

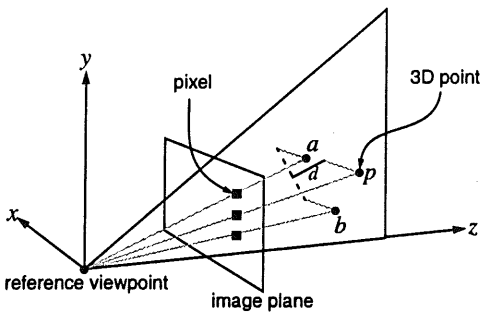

FIGURE 2. Approximation error of vertical refinement point p is calculated as point-to-line distance in the projection on the y,z-plane.

For refinement points p bisecting a diagonal edge between points a and b as in Figure 1 b), the approximation error is calculated as the actual 3D distance:

$$d = \frac{|(b - a) \times (b - p)|}{|b - a|} \qquad \textbf{(EQ 2)}$$

For field-of-view angles of less than 50°, Equation 1 introduces an error of less than $1.0 - \cos 25° = 9.4\%$.[1] Note that at most one third of the refinement points are center vertices as shown in Figure 1 b), all others are vertical or horizontal refinement points as in Figure 1 c). Therefore,

because only 33% of points are diagonal refinement points, we achieve a significant speed-up using Equation 1 instead of Equation 2 for vertical and horizontal refinement points. Note that this error metric is maximized such that center vertices (Figure 1 b)) store the maximal error of all points within that subtree of the quadtree hierarchy. To avoid expensive square root evaluations the distance is actually used as squared value d^2.

Additionally for meshing and rendering purposes we compute and store the following information in the hierarchy. For each depth-mesh vertex p we store the surface normal n_p, and we calculate a per-vertex quality measure $\rho_p = |n_p \bullet (0, 0, 1)^T|$ with the vector (0,0,1) being the view direction in the local camera coordinate system. A per-vertex quality measure allows smooth interpolation and blending over depth-mesh triangles in contrast to the per-triangle quality measure proposed in [DCV98]. Furthermore, for each center vertex of a quadtree block (Figure 1 b)) we store a bounding sphere radius r_p that includes all depth-mesh vertices within that block.

3.3 Segmentation

The triangulation of the z-buffer may introduce surface interpolations between different object surfaces and the background as shown in Figure 3 that causes artificial occlusion or *rubber sheet* [MMB97] artifacts when warped to new viewpoints. Therefore, it is important to determine whether triangles represent rubber sheets or not.

a) b)

FIGURE 3. Rubber sheet artifacts showing in a) are removed in b) by appropriate segmentation.

If performed at run-time, this segmentation of the triangulated depth-mesh has to be done very quickly. We perform an efficient per-triangle segmentation on the full resolution depth-mesh once during the depth-mesh initialization phase and store the results in the multiresolution hierarchy. In [PMS02] we discussed several fast segmentation alternatives including *connectedness* [MMB97], *disparity* [OB98] and *orthogonality* [PMS02]. Any of these three methods can be used efficiently within our proposed approach. Here we only want to briefly outline the *orthogonality* test that we used in our current implementation.

We can observe that the rubber sheet triangles introduced during the triangulation of the depth-image have the following property: the triangle normal is almost perpendicular to the vector from the viewpoint to the center of the triangle as shown in Figure 4. Let v be the vector from viewpoint to the center of triangle t and n_t be the normal of t. The following inequality using an angular threshold ω can be used to determine if a triangle is a rubber sheet triangle:

$$\left| \frac{v}{|v|} \bullet n_t \right| < \cos(90° - \omega) \qquad \textbf{(EQ 3)}$$

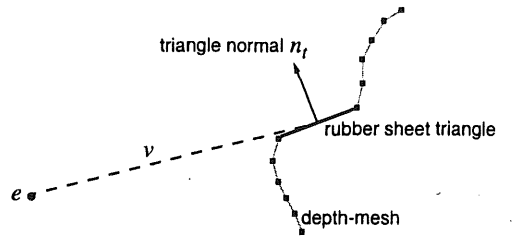

FIGURE 4. Rubber sheet triangles in the depth-mesh.

Additional care should be taken not to remove very small triangles which represent rough surface features but do not constitute a discontinuity. Therefore, in addition to Equation 3 we also consider the depth-range Δz of triangles at distance z in the camera coordinate system and we only remove triangles which span a depth-range larger than some threshold λ:

$$\frac{\Delta z}{z} > \lambda \qquad \textbf{(EQ 4)}$$

The result of the segmentation process, if a triangle is considered a rubber sheet, is stored as an 8-bit boolean flag for the triangles $a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2$ of each quadtree node as shown in Figure 5 a). For coarser triangles $t = a$ (and b, or c) within that node as shown in Figure 5 b), the segmentation is determined by the expression $t_1 \vee t_2$. Similarly, for a node on a level l-1 the flag of triangle a_1 is recursively set to $a \vee b$ from the child node on level l (and analogously for the other triangles) as shown in Figure 5 c). Thus the segmentation of one triangle causes all parent triangles in the quadtree hierarchy to be segmented as well. After initialization, segmentation can be determined for each node at rendering-time by a simple boolean expression.

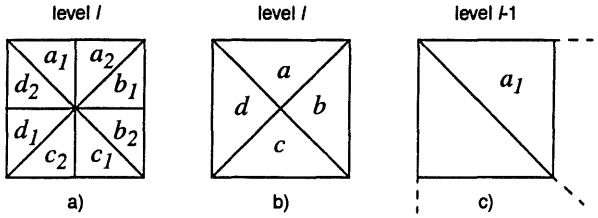

FIGURE 5. a) Segmentation flags for the full-resolution triangles of a quadtree node are stored as an 8-bit boolean field, and b) for the simplified triangles are expressed as boolean 'or' combinations. c) Flags on level l-1 are recursively calculated from level l.

3.4 Real-time meshing

At rendering time, for each frame an adaptively triangulated depth-mesh can be extracted according to the current viewpoint and frustum. Given the viewpoint e in the depth-

1. Thus this could conservatively be taken into account and added to the result of Equation 1 if desired.

mesh's local camera coordinate system and an image-space error tolerance τ, a depth-mesh vertex p with geometric error d inside the view frustum is selected if

$$\left(\frac{d}{|p-e|}\right)^2 > \tau. \qquad \text{(EQ 5)}$$

This vertex selection is performed recursively top-down in the quadtree hierarchy with the bounding sphere information of each node used for view-frustum culling. The vertex dependency rules explained in [LKR+96, Paj98] are used to guarantee a crack-free triangulation. The triangle-strip construction method proposed in [Paj98] is used and modified to incorporate segmentation of rubber sheet triangles as shown in Figure 6. While creating the triangle strip sequence for a selected set of vertices, the segmentation flags of each visited quadtree node are checked, and if necessary the strip is broken up into multiple smaller triangle strips.

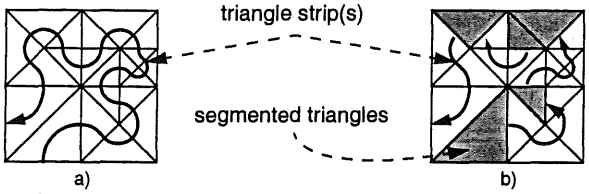

FIGURE 6. a) Triangle strip sequence of an adaptively triangulated quadtree, and b) split into multiple shorter strips due to segmentation of triangles.

Figure 7 illustrates the preprocess stages to generate the multiresolution depth-mesh data structures and Figure 11 shows some examples of adaptively triangulated and segmented depth-meshes.

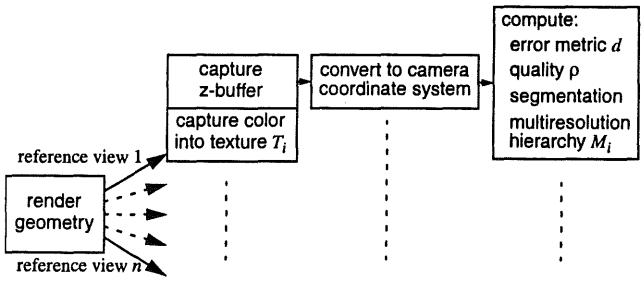

FIGURE 7. Depth-mesh generation preprocess.

4. Depth-mesh rendering

4.1 Overview

Depth-image warping can efficiently be performed by hardware supported rendering of textured polygons instead of projecting every single pixel from a reference depth-image to new views. The approximate depth-image consisting of a segmented triangulation of the depth-buffer, as outlined in the previous section, is rendered using the color values of the reference frame-buffer as texture. Rendering a depth-image with a resolution of $2^k \times 2^k$ pixels involves warping of 2^{2k} pixels with traditional depth-image warping techniques (or rendering about $2 \cdot 2^{2k}$ triangles with [MMB97]). With the proposed technique, instead of warping the $2^{18} = 262144$ pixels of a 512×512 reference image, a textured depth-mesh with only a few thousand triangles can be rendered using hardware accelerated 3D graphics at a fraction of the cost of per-pixel image warping.

The depth-mesh generation and segmentation is performed in the reference view coordinate system. Whenever a depth-mesh has to be rendered, the coordinate system transformation of that reference view is used as model-view transformation to place the depth-mesh correctly in the world coordinate system.

To reduce exposure artifacts, most image warping techniques render multiple reference images that have to be merged to synthesize a new view. We present a novel and highly efficient blending algorithm that exploits graphics hardware acceleration and that supports per-pixel weighted blending of reference depth-images. Blending of n reference depth-meshes to synthesize a new view consists of the following basic steps:

1. Select n reference depth-meshes M_i ($i = 1...n$) and textures T_i to be used for the current view, and calculate their positional blending weights w_i with respect to the current viewpoint e.
2. Adaptively triangulate the depth-meshes M_i for the current viewpoint e, and generate the segmented triangle strip representations S_i.
3. Render the triangle strips S_i without illumination and texturing to synthesize the final z-buffer Z_e of the current viewpoint e.
4. Render the triangle strips S_i again with their textures T_i and per-vertex quality ρ as alpha values enabled.[1] The result is rendered into separate color-with-alpha frame-buffers C_i. Depth-buffer evaluation using Z_e is set to read-only at this stage.
5. Synthesize the new image I from buffers C_i using positional weights and alpha-blending:

$$I = \sum_{i=1...n} w_i \cdot C_i.$$

6. The image I contains the per-pixel weighted result. Note that the final alpha blending factor per pixel may be less than 1.0 at this stage and a normalization of the corresponding color yields the final image.

Figure 8 illustrates the data flow and rendering stages of our algorithm. The following sections explain the different stages in more detail and show how our algorithm exploits hardware acceleration.

1. The per-vertex quality measure ρ will be Gouraud interpolated across triangles and yield a per-pixel quality measure in the alpha-channel.

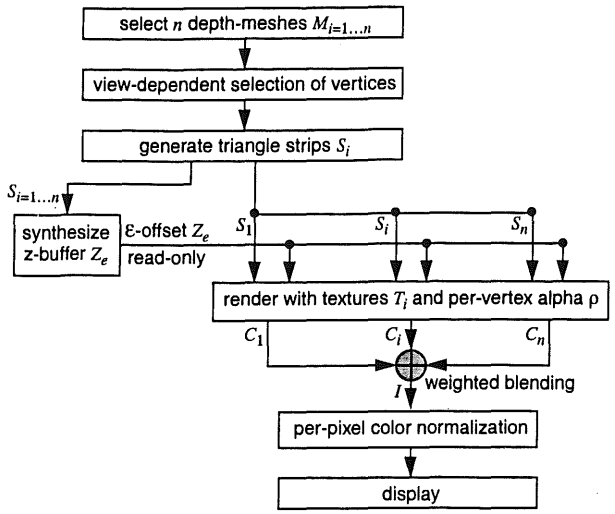

FIGURE 8. Depth-mesh rendering and blending stages.

The proposed rendering algorithm is tested and explained in more detail below for an *image-based object* (IBO) representation but is applicable to other rendering systems that make use of depth-image warping as well. In [OB99] an IBO is constructed by generating six layered depth images taken from the six sides of a cubic bounding box surrounding a single object. Similarly, we extract six textured depth-meshes around the axis-oriented bounding box of the object and store them as the reference views.

4.2 Depth-mesh selection and triangulation

During rendering, for each frame we first select the three depth-meshes $M_{i=1,2,3}$ out of the six reference views that face the novel rendering position e. The blending weights $w_{i=1,2,3}$ are calculated based on the euclidean distances from the depth-mesh viewpoints to the novel viewpoint, and are normalized such that $\sum_{i=1,2,3} w_i = 1.0$.

Each selected depth-mesh M_i is simplified view-dependently for the new viewpoint e according to an image-space geometric error tolerance of τ pixels. This is performed by a recursive top-down traversal of the quadtree triangulation hierarchy of M_i as outlined in Section 3. The resulting triangulation is segmented and represented by a set of triangle strips S_i which contain vertices with texture coordinates into the reference view texture T_i and per vertex quality values ρ used as RGBA components.

4.3 Rendering using an ε-z-buffer

To achieve a smooth blending between overlapping depth-meshes $M_{i=1,2,3}$ the rendering must allow some tolerance in the z-buffer visibility test. Depth-meshes that for a pixel cover the same surface region within some tolerance ε should be blended together, and only if the z-buffer values are sufficiently different the front-most depth-mesh determines the final color of that pixel.

This ε-z-buffer rendering is achieved by first drawing the triangle strips S_i without any shading, illumination or texturing enabled to initialize the z-buffer to Z_e for the desired viewpoint e. Since the buffers require to be cleared during the following steps, the stencil buffer is set to store which areas of the frame buffer will be overwritten by the IBO. From here on we will assume that the stencil test is activated to block rendering in areas of the original color and depth buffer where the IBO is not present.

In a second pass and using the previously computed z-buffer Z_e in read-only mode, the meshes S_i are rendered again individually into color buffers C_i at a slight negative offset in the view-direction. Initially a background quad is rendered with its alpha channel set to the corresponding positional weight w_i for the reference view. Now shading, alpha-blending, texturing and per vertex color components are enabled. Since each vertex' RGBA color is set to its quality measure (ρ,ρ,ρ,ρ), Gouraud shading and texture modulation with T_i is enabled, the resulting image C_i contains the per-pixel weighted colors of the warped depth-mesh multiplied with positional weight w_i of the view. Then the color buffer is copied into the reference texture T_i.

At this point a pixel p in the frame buffer C_i with interpolated quality $\bar{\rho}_p$ from Gouraud shading and texture coordinates s,t will finally store the desired weighted color $(w_i \cdot \bar{\rho}_p \cdot Red(T_i(s,t)),\ w_i \cdot \bar{\rho}_p \cdot Green(T_i(s,t)),\ w_i \cdot \bar{\rho}_p \cdot Blue(T_i(s,t)),\ \bar{\rho}_p)$, or short $(w_i \cdot \bar{\rho}_p \cdot R_p,\ w_i \cdot \bar{\rho}_p \cdot G_p,\ w_i \cdot \bar{\rho}_p \cdot B_p,\ w_i \cdot \bar{\rho}_p)$.

Note that rendering the selected depth-meshes S_i twice is not very costly as can be seen in Section 5 because rendering of triangle strips with only a few thousand triangles is extremely efficient. On the other hand, the more costly operation of view-dependent vertex selection and segmentation is only done once.

4.4 Per-pixel blending and normalization

As outlined above, the images C_i now contain the quality and positional weighted contributions of the selected depth-meshes M_i. The final rendering stages must now perform the image composition and normalization of the color values. The two steps involved are the summation of the weighted colors by $I = \sum C_i$ followed by normalizing each color component. The image composition operation can be performed efficiently by alpha-blending n quadrilaterals using C_i as textures on these quads.

The sum yields an image I with pixel colors $(\alpha \cdot R,\ \alpha \cdot G,\ \alpha \cdot B,\ \alpha)$. These color values constitute the proportionally correctly blended values, however, the α values need not be 1.0 as required. To get the final desired color $(R, G, B, 1.0)$ it is obvious that each color component has to be multiplied with α^{-1}. Without any hardware extensions to perform complex per-pixel manipulations this normalization step has to be performed in software on the main CPU and the resulting image has to be uploaded to the graphics hardware frame buffer for display. However, widely available graphics accelerators now offer per-pixel shading operators that can be used more efficiently. In our current implementation, we perform this normalization in hardware using nVIDIA's OpenGL *Texture Shader* extension [DS01].

To compensate the illumination deficiency we perform a remapping of the R, G and B values based on the value of α.

During initialization time we construct a texture encoding in (s,t) of a look-up table off transparency and luminance values respectively, from 0 to 256 possible values. The pixels of the textures encode the new luminance (t) compensated with the transparency (s).

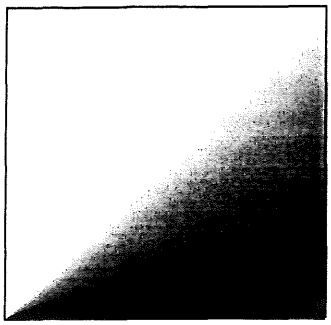

FIGURE 9. Alpha-Luminance map.

Based on this alpha-luminance map, we will proceed to correct each of the R,G and B channels of every pixel of the final image. Using the texture shader extension operation GL_DEPENDENT_AR_TEXTURE_2D_NV the video card can easily remap the output using the R and A values as texture coordinates for our alpha-luminance map. At this point, rendering a quadrilateral with the image I as texture-one and the alpha-luminance map as texture-two and setting the color mask to block the G, B and A channel would compensate the red channel with the α^{-1} amount. This process is shown in Algorithm 2.

However, the actual capabilities of the video card do not allow to use the rest of the color components with the texture shader extension. A preprocess must be performed before using the shaders to generate two additional textures that have the green and blue channels copied into the red channel, so the texture shader can be used to compensate the intensity in the complete image I.

The preprocess step starts with the final image I in the frame-buffer and reuses the textures T_i to store the modified color buffers. The first texture T_1 is a plain copy of the image I. For the second and third textures, we use the nVIDIA register combiners to copy the G and B channels into the R component. This is done using a single combiner and two passes that involve rendering two textured quadrilaterals. The result of each pass is stored in T_2 and T_3. This preprocess to remap the color channels is given in Algorithm 1.

The final rendering using the texture shader is done by incrementally rendering three quadrilaterals with the modified textures, remapped using the alpha-luminance map and blocking the appropriate color channels.

4.5 Background rendering

The last step in the rendering process consist of covering small holes from the front meshes that are caused during the segmentation process. Although the total area of these holes is very small, it generates disturbing artifacts if they are not properly colored.

This rendering is performed using a low quality non-view dependant textured depth-mesh, created during initialization and stored as a compiled display list. The overhead introduced by this final step is negligible but the visual appearance of the final image improves considerably.

```
//-------------------------------------------------------------------------
// Copy original image into T0 (R,G,B,A)
glBindTexture(GL_TEXTURE_2D, T0);
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,0,0,w,h);

// setup the register combiners to use 1 general combiner
glCombinerParameter NV(GL_NUM_GENERAL_COMBINERS_NV, 1);

// Insert colors into the Constant Color 0 register
glCombinerParameterfvNV(GL_CONSTANT_COLOR0_NV, color1); // (0,1,0,0)
glCombinerParameterfvNV(GL_CONSTANT_COLOR1_NV, color2); // (0,0,1,0)

// Set the final combiner
glFinalCombinerInputNV(GL_VARIABLE_A_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glFinalCombinerInputNV(GL_VARIABLE_B_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glFinalCombinerInputNV(GL_VARIABLE_C_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glFinalCombinerInputNV(GL_VARIABLE_D_NV, GL_SPARE0_NV,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glFinalCombinerInputNV(GL_VARIABLE_G_NV, GL_TEXTURE0_ARB,
GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

glEnable(GL_REGISTER_COMBINERS_NV);

//-------------------------------------------------------------------------
// Get the green texture T1 (G,G,G,A)
// setup combiner 1
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
GL_TEXTURE0_ARB, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
GL_CONSTANT_COLOR0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerOutputNV(GL_COMBINER0_NV, GL_RGB, GL_SPARE0_NV,
GL_DISCARD_NV, GL_DISCARD_NV, GL_NONE, GL_NONE, GL_TRUE, GL_FALSE,
GL_FALSE);

// Render the original texture (is in T0)
glBindTexture(GL_TEXTURE_2D, T0);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glBegin(GL_QUADS);
        glTexCoord2d(0,0); glVertex2d(0,0);
        glTexCoord2d(1,0); glVertex2d(w,0);
        glTexCoord2d(1,1); glVertex2d(w,h);
        glTexCoord2d(0,1); glVertex2d(0,h);
glEnd();

// Copy original image into T1
glBindTexture(GL_TEXTURE_2D, T1);
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,0,0,w,h);

//-------------------------------------------------------------------------
// Get the blue texture T2 = (B,B,B,A)
// Modify the settings of the combiner
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
GL_TEXTURE0_ARB, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
GL_CONSTANT_COLOR1_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV, GL_ZERO,
GL_UNSIGNED_IDENTITY_NV, GL_RGB);
glCombinerOutputNV(GL_COMBINER0_NV, GL_RGB, GL_SPARE0_NV,
GL_DISCARD_NV, GL_DISCARD_NV, GL_NONE, GL_NONE, GL_TRUE, GL_FALSE,
GL_FALSE);

glBindTexture(GL_TEXTURE_2D, colorID[0]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glBegin(GL_QUADS);
        glTexCoord2d(0,0); glVertex2d(0,0);
        glTexCoord2d(1,0); glVertex2d(w,0);
        glTexCoord2d(1,1); glVertex2d(w,scrY);
        glTexCoord2d(0,1); glVertex2d(0,scrY);
glEnd();

// Copy original image into T2
glBindTexture(GL_TEXTURE_2D, T1);
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,0,0,w,h);

glDisable(GL_TEXTURE_2D);
glDisable(GL_REGISTER_COMBINERS_NV);
```

ALGORITHM 1. Preprocess for color remapping.

```
// Normalize illumination
// Here rendering the quad would generate the R channel normalized.
// Texture 0 is the original image
// Texture 1 is the alpha_lum map

glEnable(GL_TEXTURE_SHADER_NV);

glActiveTextureARB( GL_TEXTURE1_ARB );
glBindTexture(GL_TEXTURE_2D, alpha_luminance_map);      // alpha_lum map
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
GL_DEPENDENT_AR_TEXTURE_2D_NV );
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_PREVIOUS_TEXTURE_INPUT_NV,
GL_TEXTURE0_ARB);

// Normalize red channel using T0
glActiveTextureARB( GL_TEXTURE0_ARB );
glBindTexture(GL_TEXTURE_2D, T0);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glColorMask(GL_TRUE,GL_FALSE,GL_FALSE,GL_FALSE);
glBegin(GL_QUADS);
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,0); glVertex2d(0,0);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,0); glVertex2d(w,0);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,1);glVertex2d(w,scrY);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,1); glVertex2d(0,scrY);
glEnd();

// Normalize green channel using T1
glActiveTextureARB( GL_TEXTURE0_ARB );
glBindTexture(GL_TEXTURE_2D, T1);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glColorMask(GL_FALSE,GL_TRUE,GL_FALSE,GL_FALSE);
glBegin(GL_QUADS);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,0); glVertex2d(0,0);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,0); glVertex2d(w,0);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,1); glVertex2d(w,h);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,1); glVertex2d(0,scrY);
glEnd();

// Normalize blue channel using T2
glActiveTextureARB( GL_TEXTURE0_ARB );
glBindTexture(GL_TEXTURE_2D, T2);
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glColorMask(GL_FALSE,GL_FALSE,GL_TRUE,GL_FALSE);
glBegin(GL_QUADS);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,0); glVertex2d(0,0);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,0); glVertex2d(w,0);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1,1); glVertex2d(w,scrY);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0,1); glVertex2d(0,scrY);
glEnd();
```

ALGORITHM 2. Color normalization.

5. Experiments

In this section we report experimental results of our depth-meshing and warping algorithm for different polygonal models. For each model we create an image-based object (IBO) similar to [OB99] using a bounding box of six depth-meshes around the object.

Table 1 shows timing results for the generation of depth-meshes with our approach given a depth-image size of 513^2 = 263169 pixels. The timing was performed on a Dell 2.2GHz Pentium4 using a nVIDIA GeForce4 4600Ti with the Detonator 4.0 drivers. The table shows the size of the test models, the time it takes to render the geometry, the cost to capture and convert the z-buffer into 3D points, and the cost to initialize the depth-mesh M. This initialization includes the calculation of the error metric, per-vertex quality measure, triangle segmentation flags, and other multiresolution parameters as described in Section 3. Not shown due to negligible cost is the capturing of the color texture T corresponding to a particular depth-mesh M. The segmentation angle tolerance ω was set to 5° and the depth-range threshold λ to 5/100 of the object's max extent. As expected, one can observe that the depth-mesh initialization is constant for a given reference image resolution. The quadtree hierarchy construction grows with $O(n \cdot \log n)$ for a reference depth-image of n pixels.

Model	Triangles	Render	z-buffer	Quadtree
David	8254150	4040 ms	39 ms	495 ms
Dragon	871414	430 ms	36 ms	405 ms
Happy	100000	49 ms	36 ms	409 ms
Female	605086	811 ms	36 ms	407 ms
Male	641560	849 ms	36 ms	408 ms

TABLE 1. Depth-mesh construction cost for different polygonal models. Given is the model size, rendering cost, z-buffer capture and conversion, and multiresolution model construction.

Table 2 shows the rendering performance of our depth-mesh image based objects, and the corresponding screen shots are given in Figure 10. We used an image-space error tolerance τ of 1 pixel for the David model, 4 pixels for the Female and Male models, 16 pixels for the Dragon, and 64 pixels for the Happy model. The selection of three reference views has negligible costs and is omitted here. The synthesis cost of generating the z-buffer Z_e includes the view-dependent selection of vertices from the depth-meshes M_i, as well as rendering the segmented triangle strips S_i once (Steps 2 and 3 from Section 4.1). The rendering of images C_i corresponds to Step 4, and blending includes Steps 5 and 6 from Section 4.1.

It can be observed from Table 2 that our depth-meshing and warping algorithm provides stable and interactive frame rates for arbitrary complex objects, and that the rendering performance is completely independent from the size of the input polygonal model. The total depth-mesh rendering and blending cost is in the order of 40 to 60 ms. The per-pixel weighted blending of the individual warped depth-images C_i is extremely efficient and scales to arbitrary number of input images C_i as long as the pixel fill rate of the graphics hardware is not exceeded. One can also see that depth-meshing is advantageous in terms of complexity of per-pixel depth-image warping. As can be seen from Table 2, the view-dependently triangulated depth-meshes S_i only consist of up to 30K textured triangles which represent three 512^2=262144 pixel depth-images.

Model	Geometry		IBO					
	time	FPS	Triangles of $S_{i=1,2,3}$	Synthesize z-buffer Z_e selection	rendering	Rendering of $C_{i=1,2,3}$	Blend and normalize	FPS
David	4029 ms	0.25	18447	13.6 ms	15.2 ms	15.5 ms	0.08 ms	22
Dragon	430 ms	2.4	30635	16.6 ms	23.3 ms	23.5 ms	0.08 ms	15
Happy	50 ms	20	22316	13.2 ms	16.9 ms	17.1 ms	0.08 ms	21
Female	813 ms	1.2	29735	13.3 ms	19.9 ms	20.2 ms	0.1 ms	18
Male	863 ms	1.2	22423	11.4 ms	15.9 ms	16.2 ms	0.08 ms	21

TABLE 2. Depth-mesh rendering performance of image based objects. It shows the number of triangles rendered from the selected depth meshes $S_{i=1,2,3}$ and the timing of the different rendering stages.

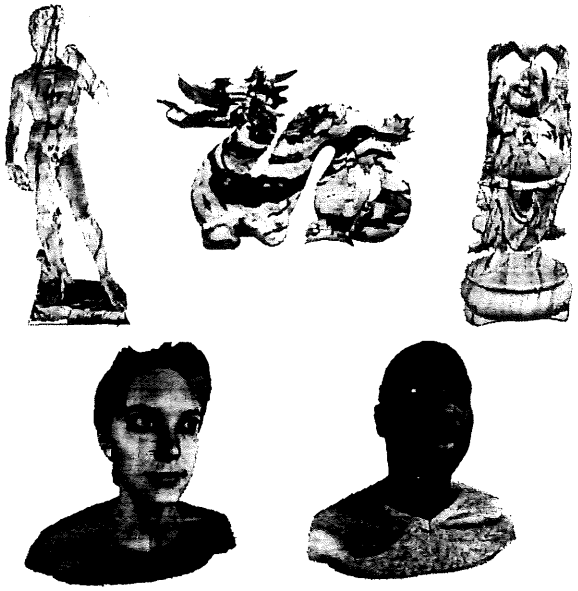

FIGURE 10. Renderings of complex polygonal objects from six reference depth-meshes using our per-pixel weighted blending algorithm.

Figure 13 illustrates results from our per-pixel weighted blending algorithm. We compare the rendering results to a simple binary merge of depth-meshes, where the depth-mesh closest to the viewpoint determines the pixel color, and to our previous positional weighted blending algorithm presented in [PMS02]. The images show the polygonal object, the binary merged depth-meshes, the positional weighted blending [PMS02], and the new per-pixel weighted blending. Furthermore, Figure 12 shows the occupancy image in pseudo coloring that shows the per-pixel blending. Each color in this occupancy image specifies how many and which depth-meshes are used to render a particular pixel. The colors red, green and blue specify the individual depth-meshes, and any combination thereof determines which depth-meshes are blended together.

6. Discussion

We presented an efficient view-dependent depth-image meshing and segmentation method as well as a novel real-time depth-image blending algorithm that exploits hardware graphics acceleration. The proposed depth-buffer triangulation approach significantly reduces the complexity of depth-image warping by view-dependent simplification of the triangulated depth-mesh and rendering textured triangle strips. Our novel blending technique provides extremely efficient per-pixel weighted blending of depth-images.

Our approach improves over previous depth-image warping methods [McM95, MMB97, DSV97, DCV98] in particular in the following aspects: efficient generation of depth-image based object representations, and weighted blending of multiple reference views at interactive frame rates. The presented approach can be used as a rendering component in visualization systems such as large scale walk-through applications [RP94, SLS+96, SS96, ACW+99].

The main limitation of the presented approach is the limited ability to cover exposure artifacts in comparison to layered approaches [SGHS98, DSSD99].

Future work in this context will include extension of the proposed approach to a multi-layer depth-mesh representation, application of our blending algorithm to other IBR methods, and taking advantage of per-pixel and per-vertex programming features of modern graphics hardware accelerators.

Acknowledgements

This work was partially supported by NSF grant CCR-0119053.

References

[ACW+99] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenneth Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings Symposium on Interactive 3D Graphics*, pages 199–206. ACM SIGGRAPH, 1999.

[AL99] Daniel Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *Proceedings SIGGRAPH 99*, pages 307–316. ACM SIGGRAPH, 1999.

[CBL99] Chun-Fa Chang, Gary Bishop, and Anselmo Lastra. Ldi tree: A hierarchical representation for image-based rendering. In *Proceedings SIGGRAPH 99*, pages 291–298. ACM SIGGRAPH, 1999.

[CN01] Baoquan Chen and Minh Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proceedings IEEE Visualization 2001*, pages 45–52, 2001.

[CW93] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings SIGGRAPH 93*, pages 279–288. ACM SIGGRAPH, 1993.

[DBC+99] Paul Debevec, Christoph Bregler, Michael Cohen, Leonard McMillan, Francois Sillion, and Richard Szeliski. Image-based modeling and rendering. SIGGRAPH 99 Course Notes 39, 1999.

[DCV98] Lucia Darsa, Bruno Costa, and Amitabh Varshney. Walkthroughs of complex environments using image-based simplification. *Computers & Graphics*, 22(1):25–34, 1998.

[DS01] Sébastien Dominé and John Spitzer. Texture shaders. Developer Documentation, 2001.

[DSSD99] Xavier Decoret, Gernot Schaufler, Francois Sillion, and Julie Dorsey. Multi-layer impostors for accelerated rendering. In *Proceedings EUROGRAPHICS 99*, pages 61–72, 1999.

[DSV97] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating static environments using image-space simplification and morphing. In *Proceedings Symposium on Interactive 3D Graphics*, pages 25–34. ACM SIGGRAPH, 1997.

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings SIGGRAPH 96*, pages 109–118. ACM SIGGRAPH, 1996.

[Max96] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer z-buffers. In *Proceedings Eurographics Rendering Workshop 96*, pages 165–174. Eurographics, 1996.

[McM95] Leonard McMillan. A list-priority rendering algorithm for redisplaying projected surfaces. Technical Report UNC-95-005, University of North Carolina, 1995.

[MMB97] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In *Proceedings Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, 1997.

[OB98] Manuel M. Oliveira and Gary Bishop. Dynamic shading in image-based rendering. Technical Report UNC-98-023, University of North Carolina, 1998.

[OB99] Manuel M. Oliveira and Gary Bishop. Image-based objects. In *Proceedings Symposium on Interactive 3D Graphics*, pages 191–198. ACM SIGGRAPH, 1999.

[Paj98] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization 98*, pages 19–26,515, 1998.

[Paj02] Renato Pajarola. Overview of quadtree-based terrain triangulation and visualization. Technical Report UCI-ICS-02-01, Information & Computer Science, University of California Irvine, 2002. submitted for publication.

[PMS02] Renato Pajarola, Yu Meng, and Miguel Sainz. Fast depth-image meshing and warping. Technical Report UCI-ECE-02-02, The Henry Samueli School of Engineering, University of California Irvine, 2002. submitted for publication.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings SIGGRAPH 2000*, pages 335–342. ACM SIGGRAPH, 2000.

[QWQK00] Huamin Qu, Ming Wan, Jiafa Qin, and Arie Kaufman. Image based rendering with stable frame rates. In *Proceedings IEEE Visualization 2000*, pages 251–258. Computer Society Press, 2000.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings SIGGRAPH 2000*, pages 343–352. ACM SIGGRAPH, 2000.

[RP94] Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. In *Proceedings SIGGRAPH 94*, pages 155–162. ACM SIGGRAPH, 1994.

[SGHS98] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered depth images. In *Proceedings SIGGRAPH 98*, pages 231–242. ACM SIGGRAPH, 1998.

[SLS+96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings SIGGRAPH 96*, pages 75–82. ACM SIGGRAPH, 1996.

[SS96] Gernot Schaufler and Wolfgang Stürzlinger. A 3d image cache for virtual reality. In *Proceedings EUROGRAPHICS 96*, pages 227–236, 1996.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings SIGGRAPH 2001*, pages 371–378. ACM SIGGRAPH, 2001.

FIGURE 11. Examples of adaptively triangulated and segmented depth-meshes. From left to right: rendering from polygonal model, depth-mesh in wire frame and segmented textured depth-mesh.

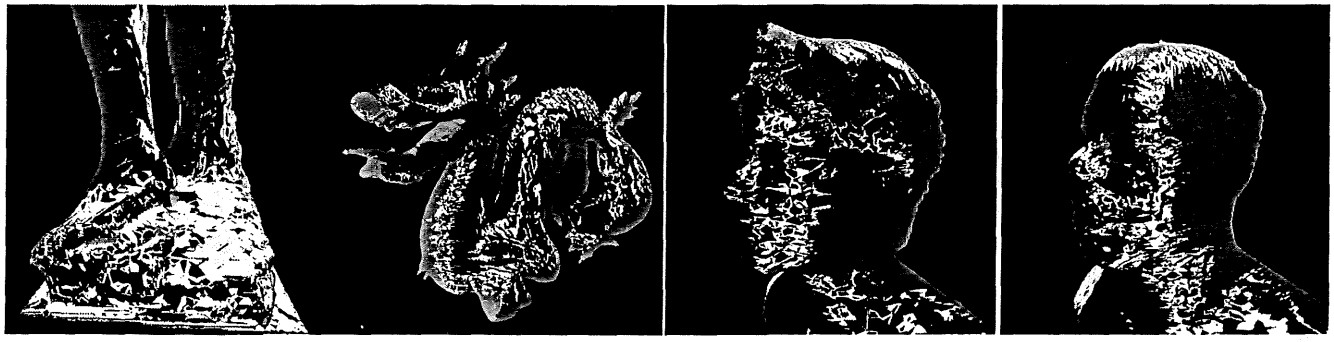

FIGURE 12. Occupancy pseudo colored images. The RGB primary colors correspond to the three different blended depth-meshes. Any other mixed color illustrates the combination of multiple depth-meshes.

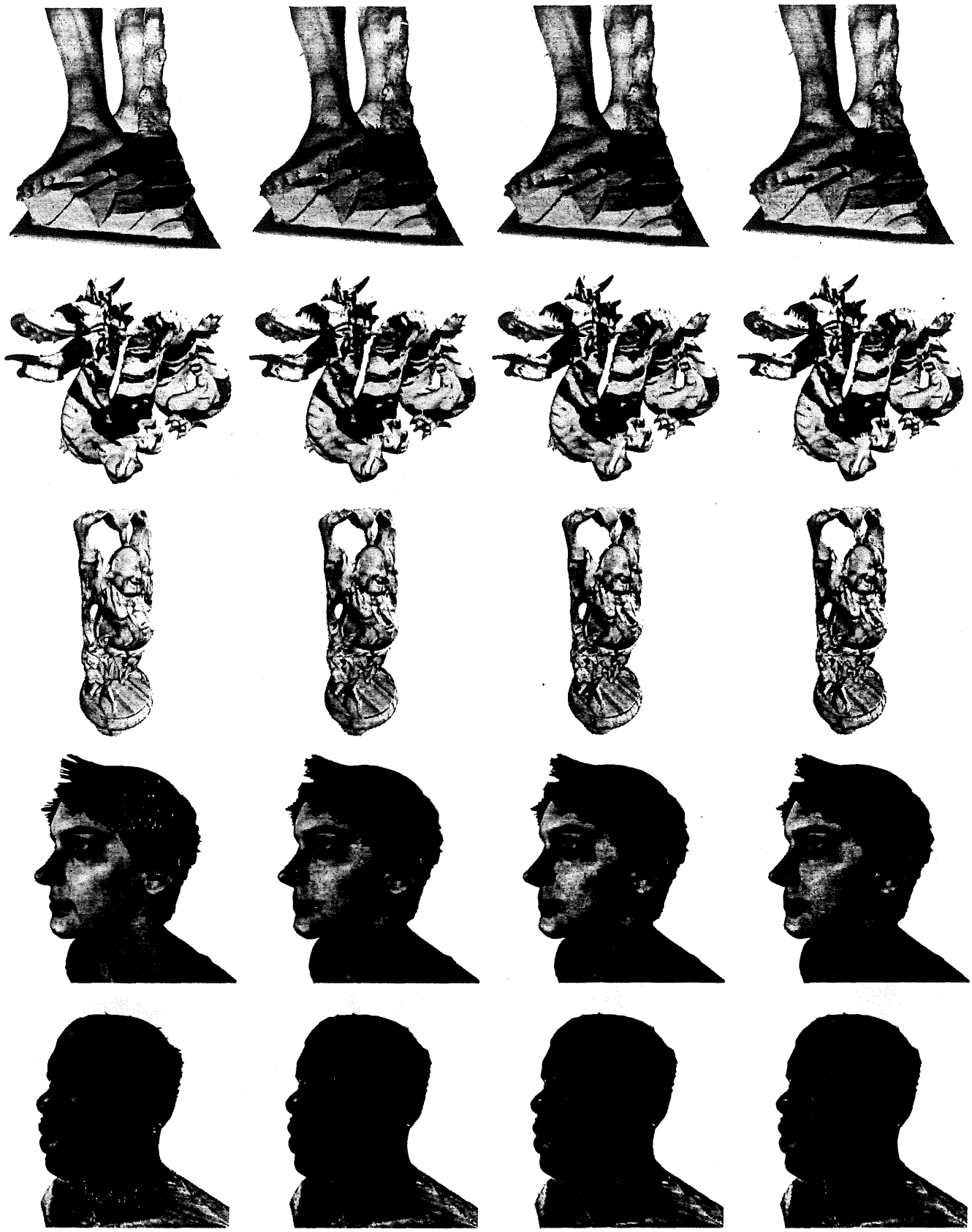

FIGURE 13. Renderings of complex polygonal objects from six reference depth-meshes, aligned with the bounding box, using our proposed positional weighted blending algorithm. The leftmost in any row shows the actual polygonal object, followed by a simple binary blending of depth-meshes, a positional weighted blending and our proposed per-pixel weighted blending algorithm.