View-Dependent Layered Projective Texture Maps

Alex Reche-Martinez[1,2] and George Drettakis[1]

[1]REVES/INRIA Sophia-Antipolis,
http://www-sop.inria.fr/reves
{Alex.Reche,George.Drettakis}@sophia.inria.fr

[2]Centre Scientifique et Technique du Bâtiment,
http://www.cstb.fr

Abstract

Capturing and rendering of real scenes in an immersive virtual environment is still a challenging task. We present a novel workflow, using high-quality, view-dependent projective texturing at low texture memory cost, while reinforcing artist's control over image quality. Photographs of a real scene are first used to create a 3D model with standard tools. Our method automatically orders geometry into optimized visibility layers for each photograph. These layers are subsequently used to create standard 2D image-editing layers, enabling artists to fill in missing texture using standard techniques such as clone brushing. The result of this preprocess is used by our novel layered projective texture rendering algorithm, which has low texture memory consumption, high interactive image quality and avoids the need for subdividing geometry for visibility. We show results of our implementation on a real-world project.

1. Introduction

Recent developments in modeling from images and other real-world capture techniques (e.g., laser scanning, stereo-based methods), have resulted in an increasing interest in the creation and display of 3D models in realistic computer graphics and virtual environments. The visual quality of such environments can be very high, thanks to the richness of the textures extracted from high-resolution digital photography and the quality of the reconstructed 3D geometry.

The applications are numerous. Urban planning and environmental impact studies, archaeology and education, training, design but also film production or computer games are just a few cases where these highly-realistic virtual environments can be used.

Modeling-from-images approaches have been largely based on pioneering work in Computer Vision [15, 4]. Image-based modelling and rendering methods have also been developed (e.g., [9, 5]), which are an alternative method to capture and display real world scenes. Debevec et al's [2] Facade approach, and the follow-up interactive version [3], use a view-dependent rendering algorithm. The textures used in display are blended from multiple views. In the interactive case, projective textures are used, requiring a geometric visibility pre-process (subdivision of polygons).

In practice, several commercial modeling-from-images products have been since developed and are used for real world projects[1]. All of these products however have adopted standard, rather than projective texture mapping for display: each polygon has an associated texture, which is extracted from the input photographs by applying inverse camera projection. The output can thus be directly viewed with a traditional rendering system. This choice also allows artists to intervene at various stages of the process, using standard image-editing tools (such as Adobe Photoshop^{TM}(http://www.adobe.com), or GIMP (http://www.gimp.org)), to fill in missing textures or remove undesirable objects etc., or standard modeling tools to edit geometry.

In this paper, we adopt a view-dependent display model, by blending textures taken from different input photographs corresponding to different viewpoints, thus achieving high quality. We strive to stay compatible with standard rendering systems typically in the context of a scene-graph based approach. We develop a new projective texture rendering approach, which does not require subdivision of polygons. We automatically generate image-layers, reducing texture memory requirements compared to texture extraction approaches. The image-layers can be used in a standard image-editing workflow, permitting artists to tightly control final image quality.

We have implemented our system, and we show results from a reconstructed 3D model of a real-world project, which is the construction of the Nice Tramway in France.

A longer technical report describing the details and more results of this work can be found at **http://www.inria.fr/reves/publications/data/2003/RD03/**.

[1]E.g., www.realviz.com, www.canoma.com, www.photomodeler.com.

2. Related Previous Work

In their original paper [2], Debevec et al. presented the first modeling-from-images system Facade. The idea of view-dependent texture mapping (VDTM) was presented, but the rendering approach is clearly off-line and uses model-based stereo to obtain very impressive visual results. This work was transposed to an interactive rendering context [3] in which polygons partially visible in one of the input images are subdivided. This method has the advantage of being completely automatic; the flip side of this is the lack of quality control in a traditional content-creation workflow. In particular, for surfaces which do not receive a projective texture, simple color interpolation is used for hole-filling, producing visual errors. Subdivision of input geometry could also be problematic in some contexts, due to numerical imprecision. The method we will present can be seen as an extension of this approach, by addressing these two issues.

Image-based modeling and rendering solutions have been developed since the mid-nineties. The Lightfield[9] and Lumigraph[5] algorithms are based on relatively dense sampling of images, so that pure (Lightfield) or geometry-assisted (Lumigraph) image interpolation suffices to render an image. The size of the data sets required for these approaches (gigabytes of image data), makes them very hard to use for realistic projects.

View-based rendering [12] or surface light fields [16] improve the basic methods. The Unstructured Lumigraph [1], can be seen as a compromise between the lumigraph and VDTM. Blending criteria are more sophisticated than VDTM, and the geometric model can be almost arbitrarily coarse. Another IBMR approach are layered depth images (LDI)[13], which add depth to a multi-layer image, resulting in high-quality renderings. All of these methods require special-purpose rendering algorithms, and in some cases are hard to integrate in a traditional scene-graph like rendering systems. For LDI's the results are dependent on the quality and availability of depth information.

Our algorithm for layer construction is based on a hardware visibility ordering algorithm by Krishnan et al.[8]. A more complete, software solution has been presented by Snyder and Lengyel [14]. Visibility cycles are resolved in the algorithm by Newell et al. [10]. Our work is also related to the Tour Into the Picture[6], where layers are built manually, and a single view is used. The work of Oh et al.[11], is also related in what concerns the integration of 3D information and image editing.

3. Overview

We assume that we have constructed a 3D model from images with calibrated cameras (we use REALVIZ

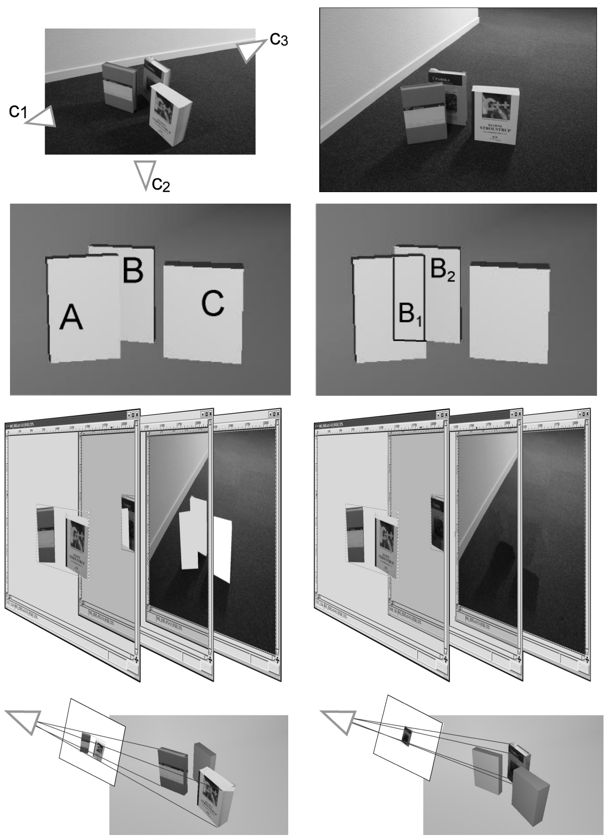

Figure 1. Row 1 left: The geometric configuration of the scenes and the cameras. Row 1 right: The input image corresponding to c_1. Row 2: the corresponding 3D geometry. Row 3 left: Resulting image-layers after geometry sorting. Row 3 right: Image-layers after editing (clone brushing, etc.). Row 4: Visualisation of the layers.

ImageModeler^{TM} in our examples).

Our goal is to achieve projective texture mapping, without the need to subdivide input geometry. In Fig. 1. the scene has three objects, A, B and C and we consider 3 views, cameras c_1, c_2 and c_3. When using the previous projective texture mapping algorithm[3], even for such a simple scene, objects A and B must be subdivided. For view c_1, B will be cut into the B_1, B_2 (see Fig. 1 second row). When moving from c_1 to c_2, object B_2 will have the image from c_1 as a projective texture, and then will blend the images from c_1 and c_2. Object B_1 however, will always be displayed with the image of c_2, since it is invisible from c_1.

Instead of doing this, we create layered images, or *image layers*, by first constructing *visibility layers*. For a given camera or view, a visibility layer is a set of surfaces for which no pair is mutually occluded. In the example of Fig. 1, and for camera c_1, there are three layers, the first containing A and C, the second containing B, which is partially hidden by A and the third the background. We then create an *image layer*, corresponding to each geometry layer (Fig. 1 third row). The first layer corresponds to the portions of the image from c_1 covered by the pixels corresponding to A and C, and the second image layer corresponds to object B.

We construct geometric visibility layers by adapting the hardware-based algorithm of [8], and then create standard image-editing (i.e., Adobe Photoshop^{TM} or GIMP) layers, which will be used as projective textures.

The artist then edits the image layer, e.g., to fill in missing texture (Fig. 1, 3rd row, right). The corresponding compact image layer are used as projective textures for the appropriate polygons of the reconstructed 3D model, avoiding the need for geometry subdivision. This is shown in the last row of Fig. 1. Because the layers and image-editing have resolved visibility the viewer can move around freely.

4. Creating Visibility and Image Layers

The first step is to create a set of images with calibrated cameras, and a corresponding coarse 3D model of the scene. We use REALVIZ ImageModeler^{TM} for this step. Our goal is to create a set of layers for each input image of the set, which will be subsequently used as projective textures for display. These layers have the following property: for any object in the first layer, no other object in the scene occludes it; for the second layer, once the objects of the first layer have been removed, the same property applies, and so on for each layer. Once these layers have been created, we can use projective texture mapping without the need to subdivide geometry, since we create a separate projective texture map for each layer.

As mentioned previously, we tightly integrate the creation of these layers with a standard image editing program, so that an artist can fill in the missing parts of a given layer use standard techniques such as clone brushing etc.

To create these image layers, we first perform a visibility sort in 3D, to create what we call *visibility layers*, which are groups of polygons. We then create what we call *image layers*, which are true image-editing-programme layers, usable in an image-editing programme such as Adobe Photoshop^{TM}or GIMP.

For the first step, we adapt an existing visibility sorting algorithm to create the visibility layers. The second step is achieved by projecting the visibility layers into the image space to create image-editing 2D layers, that are optimized to minimize the amount of texture wasted, and which then creates standard image-editing 2D layers, so that an artist can subsequently edit the result.

4.1. Visibility Layers

To compute the visibility layers we adapt the visibility algorithm of [8]. Initially, we render the entire scene into an item buffer, i.e., we use a unique identifier for each object in the scene. We then read this buffer to create an initial working set of potentially visible polygons, which thus contains all objects with at least one visible item buffer pixel containing its id.

The algorithm then iteratively constructs a set of visibility layers. For each iteration, the item-buffer of the working set is rendered into the color and the stencil buffer, and the z-test inverted (`GL_LESS`). For each pixel with a stencil value greater than 1, the object corresponding to the item-buffer value of the pixel is partially hidden, and thus the object is removed from the working set. The working set is then rendered iteratively, until the stencil buffer contains values less than or equal to 1. The remaining objects are inserted into the current visibility layer, and removed from the working set. The current layer is saved, then set to empty, and the iteration continues.

The output of this algorithm is a set of visibility layers. In each visibility layer we have a set of polygons, which are used to build layered images.

In our implementation we have optimized this process hierarchically thanks to the structure of the scene which is provided by the image modelling programme. In particular, the scene provided by ImageModeler^{TM} is organised into objects which contain a set of faces or polygons. Initially, identifiers are given to objects rather than faces, and the algorithm is performed on objects, rapidly eliminating large objects. When we can no longer proceed with objects, we split them into their constituent polygons. In our examples this is enough to resolve the visibility in the scene. Nonetheless, cycles could occur, even at the level of individual polygons/triangles, and in this case we would need to identify the cycles and cut the geometry using an algorithm such as that of Newell et al. [10].

4.2. Creating Image Layers

To create the image layers, we start with the set of visibility layers. For each visibility layer, we project each constituent polygon into image space, resulting in a 2D contour for each. We then perform a simple clustering algorithm for the set of contours. We use the fact that in `OpenGL` textures have resolution of a power of 2. For a pair of contours c_1 and c_2 we merge c_1 and c_2 into the same layer, if: $A_{12} < A_1 + A_2$, where A_i is the area of the minimal bounding box with resolution in x and y powers of two, with area greater than the area of the bounding box of the projected contour c_i. Thus each layer is potentially split into several layers.

This results in an increase of the number of layers, but is required to reduce the texture memory requirements of our projective textures. This is because very distant objects can belong to the same visibility layer, which would result in textures with large areas of empty space, wasting texture memory. The final set of layers is then ready for image editing in the appropriate program. These layers consist of a 2D bounding box in image space for the image being treated.

The layers are then sent to the image-editing program (GIMP in our case), which creates a 2D image layer corresponding to the input image.

Once layers have been created in the editing program, an artist can perform standard operations required to fill in missing texture, remove unwanted objects etc.

5. Display

After creating the visibility layers and manually editing the texture layers for the objects in the scene we use projective texture mapping for display.

For any viewpoint corresponding to the camera of a photograph, all we need to do is to render the geometry, and for each object assign as a texture the corresponding layer. To allow viewing from other viewpoints, we blend between textures with appropriate blending factors[3].

5.1. Data Structures

As mentioned previously we have designed our approach to fit into a typical scene-graph based rendering system. We use OpenGL Performer^{TM}[7] in our implementation.

For layered projective texture mapping, the graph is modified to contain a sub-image texture node. In addition to the information of a “standard node” with a projective texture material, this node contains the image coordinates of the sub-image (x_i, y_i, w_i, h_i) that will be used to compute the correct texture projection matrix.

To render the layers, we need to apply a sheared perspective transformation. This can be directly implemented using the OpenGL command `glFrustum` with parameters:

$$x'_{min} = x_{min} + \frac{x_i(x_{max} - x_{min})}{w} \qquad x'_{max} = x_{min} + \frac{(x_i + w_i)(x_{max} - x_{min})}{w}$$

$$y'_{min} = y_{min} + \frac{y_i(y_{max} - y_{min})}{h} \qquad y'_{max} = y_{min} + \frac{(y_i + h_i)(y_{max} - y_{min})}{h}$$

where $x_{min}, x_{max}, y_{min}$ and y_{max} are the values of the frustum for the projection of the original image, w and h are width and height of the original image and x_i, y_i, w_i and h_i correspond to the position and size of the layer in the original image.

5.2. Rendering Multiple Views

To render multiple views, we need to further modify our scene graph structure. We create a view-dependent projective texture node which contains a list of projective texture nodes, associated with each (unique) geometry.

When encountering such a node during graph traversal, the renderer chooses the blending factors for each appropriate texture[3], sets up the texture matrix for the nodes with non-zero blending factors, and renders the associated geometry in as many passes are required.

6. Results

We show an example which is a reconstructed model of Place Massena in Nice, France. This model has been created in the context of a virtual/augmented environment simulation of the Tramway project in Nice.

We use two input views, using input photos of resolution 2160x1440. In Fig. 2, we show intermediate synthetic views using our method. The algorithm creates a total of between 98 and 160 (optimized) layers for each image, many of which are very small and require no editing. The total texture memory required is 10-12 Megabytes, depending on the image. The algorithm to create the layers takes less than 3 minutes to complete on a Pentium IV, 1.2 Ghz. The entire process of editing and cloning the layers took about two days; using a texture extraction-based approach, with the same scene and the same views, required three weeks.

7. Discussion and Conclusions

Compared to the initial interactive projective texture mapping approach [3], we avoid the need to subdivide geometry for visibility, and we introduce more traditional artist intervention to manually fill in missing textures, rather than depending on interpolation. Thus the advantages of our approach are that we do not increase the polygon count of the scene, and that the approach fits well with traditional workflows, where artist’s control over image quality is paramount. The disadvantage is that hole-filling is no longer automatic.

Compared to texture-extraction methods, our approach significantly reduces texture memory requirements and results in better texture quality, since inverse perspective results in large textures and quality loss due to resampling. Informal observation of our system in use has shown that users are less sensitive to parallax errors due to the high quality of the projective textures.

In future work, we will be automating many aspects of the capture and texture editing phases and we plan to integrate this method in a VR setup.

8. Acknowledgements

This research was partially funded by the EU IST project CREATE, IST-2001-34231, http://www.cs.ucl.ac.uk/create. Im-

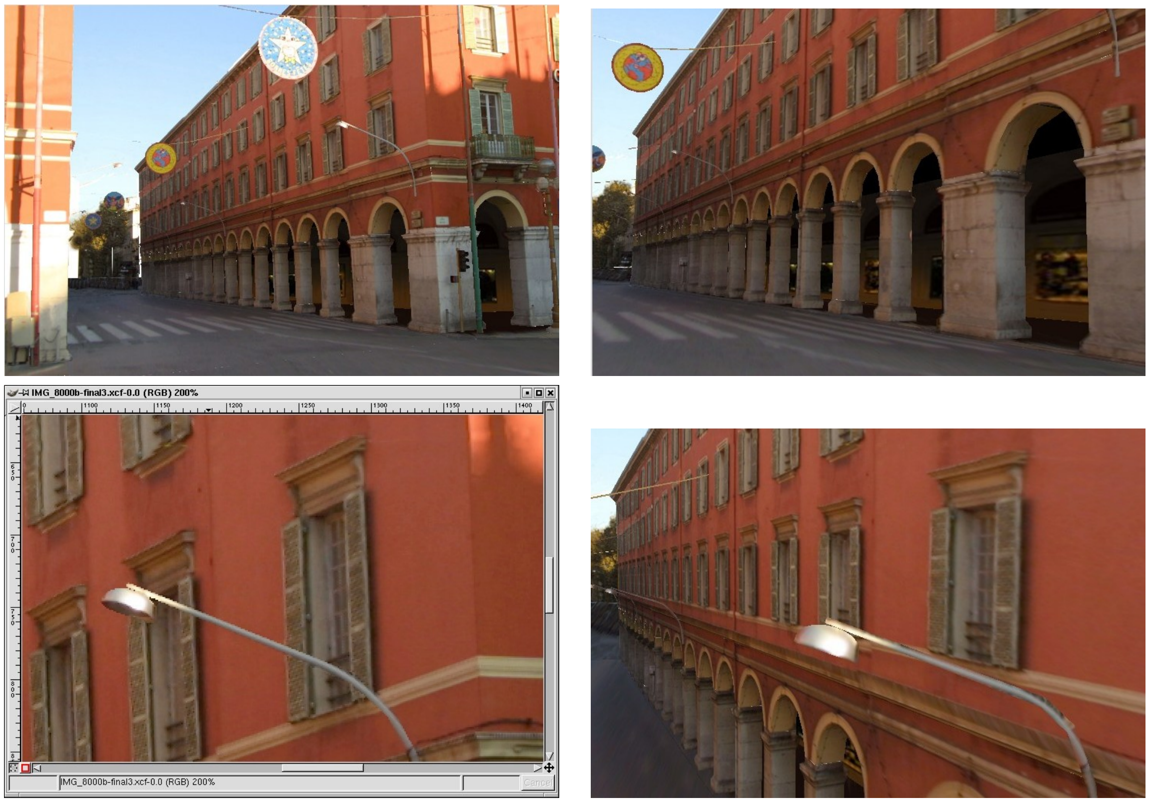
IMG_8000b-final3.xcf-0.0 (RGB) 200%
IMG_8000b-final3.xcf-0.0 (RGB) 200%

Figure 2. Top row: two intermediate (non-input) views, using our new, layered projective texture display. Lower row (right): Closeup of input image; (left) closeup of rendering of our system from a different viewpoint. The quality of the texture in the synthetic view is comparable to the input image.

ageModeler was provided by REALVIZ in the context of CREATE. Thanks to A. Olivier-Mangon for his insights and image-editing help and to F. Durand for his comments.

References

[1] C. Buehler et al. Unstructured lumigraph rendering. In *SIGGRAPH 2001, Computer Graphics Proc.*, Annual Conference Series, pages 425–432, 2001.

[2] P. Debevec et al. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proc. SIGGRAPH 96*, pages 11–20, August 1996.

[3] P. Debevec et al. Efficient view-dependent image-based rendering with projective texture-mapping. In *Rendering Techniques ’98*, 9th EG workshop on Rendering, Vienna, Austria, June 1998. Springer Verlag.

[4] O. Faugeras et al. 3-d reconstruction of urban scenes from image sequences. *CVGIP: Image Understanding*, 1997.

[5] S. J. Gortler et al. The lumigraph. In *SIGGRAPH 96 Conference Proc.*, Annual Conference Series, pages 43–54, Aug. 1996.

[6] Y. Horry et al. Tour into the picture. In *Computer Graphics Proceedings, SIGGRAPH’97*, Annual Conference Series, pages 225–232, Los Angeles, CA, August 1997. ACM.

[7] http://www.sgi.com/software/performer/.

[8] S. Krishnan et al. A Hardware-Assisted Visibility-Ordering algorithm with applications to volume rendering. In *Data Visualization 2001*, pages 233–242, 2001.

[9] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proc.*, Annual Conference Series, pages 31–42, Aug. 1996.

[10] M. E. Newell et al. A solution to the hidden surface problem. In *Proc. of the ACM Nat. Conf.*, pages 443–450, 1972.

[11] B. M. Oh et al. Image-based modeling and photo editing. In *SIGGRAPH 2001, Computer Graphics Proc.*, Annual Conference Series, 2001.

[12] K. Pulli et al. View-based rendering: Visualizing real objects from scanned range and color data. In *Rendering Techniques ’97 (Proc. of the 8th EG Workshop on Rendering) held in St. Etienne, France*, pages 23–34, 1997.

[13] J. W. Shade et al. Layered depth images. In *SIGGRAPH 98 Conference Proc.*, volume 32 of *Annual Conference Series*, pages 231–242, 1998.

[14] J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decompositions. In *SIGGRAPH 98 Conference Proc.*, 1998.

[15] C. J. Taylor and D. J. Kriegman. Structure and motion from line segments in multiple images. *IEEE Trans. on Pat. Analysis and Mach. Intelligence*, 17(11):1021–1032, 1995.

[16] D. N. Wood et al. Surface light fields for 3D photography. In *SIGGRAPH 2000, Annual Conference Proc.*, pages 287–296, 2000.