Hybrid Rendering for Interactive Virtual Scenes

Dan Morris, Neel Joshi
dmorris@cs.stanford.edu, nsj@cs.stanford.edu
Robotics Laboratory, Department of Computer Science, Stanford University, Stanford, CA

Abstract

Interactive virtual environments used in conjunction with haptic displays are often static-viewpoint scenes that contain a mixture of static and dynamic virtual objects. The immersive realism of these environments is often limited by the graphical rendering system, typically OpenGL or Direct3D. In order to present more realistic scenes for haptic interaction without requiring additional modeling complexity, we have developed a technique for co-locating a pre-rendered, raytraced scene with objects rendered graphically and haptically in real-time. We describe the depth-buffering and perspective techniques that were necessary to achieve co-location among representations, and we demonstrate real-time haptic interaction with a scene rendered using photon-mapping.

1. Introduction

The development of high-degree-of-freedom haptic feedback devices has allowed increasingly realistic physical interactions with virtual objects. However, the computational complexity of haptic rendering currently limits many haptic environments to simple geometric primitives or low-polygon-count meshes (Ruspini et al, 1997). The realism of these simple environments and the sense of physical immersion is thus heavily dependent on the graphic rendering system, which is typically OpenGL.

Since most commercially available haptic devices are mounted to a non-mobile base, the virtual environment used with these devices is often a static representation of the device's workspace, typically rendered from a single or infrequently-changing point of view.

The need for convincing graphical representations of static-viewpoint scenes suggests raytracing. However, interactive raytracing – even for fairly simple scenes – is not widely available (Wald et al, 2003). Furthermore, the computational complexity of real-time raytracing would be prohibitively high when coupled with the computational complexity of haptic rendering.

This paper describes a compromise between the interactivity of z-buffering and the detailed visual effects provided by raytracing. We feed a RenderMan scene file (Upstill, 1990) to a modified ray tracer that produces depth information along with the final image. Our system them renders the raytraced image as a point-cloud in OpenGL, and uses the original RenderMan scene file to place haptic objects in the environment. Additional objects can be rendered graphically in real-time via OpenGL, using the original scene file and a custom projection matrix. The result is a visuo-haptic environment that leverages the realistic graphical effects available via raytracing.

2. Methods

Our system consists of two independent software modules. The first module is a modified raytracer that exports depth information for each pixel. The second module is a real-time rendering system that displays the output of the raytracer, renders additional objects in OpenGL, and provides a haptic representation of the raytraced scene.

FIGURE 1 provides an overview of the system's architecture.

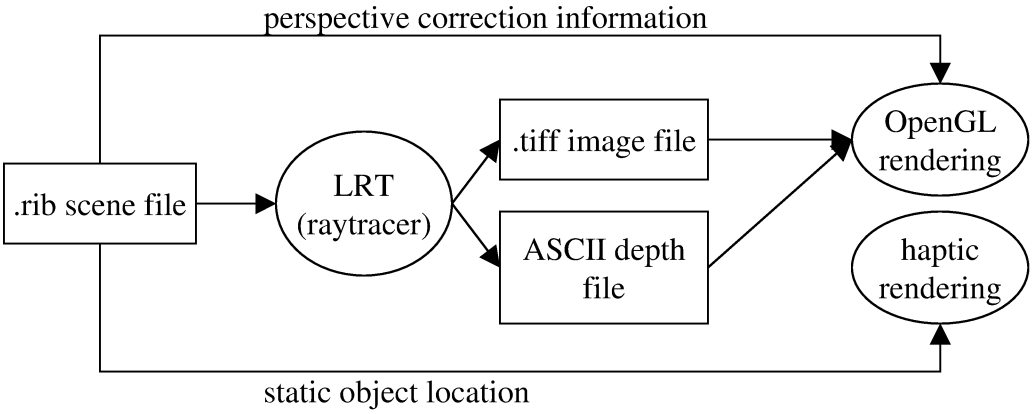

FIGURE 1: A schematic representation of the rendering system's architecture

2.1. Raytracing

In order to extract depth information with a raytraced image, we modified the LRT raytracing system, provided in source form with the preliminary manuscript of (Pharr and Humphreys, 2003). LRT reads RenderMan .rib files and generates standard image files in .tiff or .jpeg format.

We modified LRT to recognize a custom .rib file option that requests an additional output file in addition to the final image. This file is an ASCII table that provides a depth value for each pixel in the image. Depth values are obtained by casting a ray from the viewpoint through each pixel on the film plane and determining the distance along each ray at which the nearest object is intersected. These are the same intersections that are used for generating pixel color in the final image.

The exported depth values thus represent positive distance from the camera, in the same units used to describe object and camera locations in the original scene description.

2.2 Graphical rendering

We developed a software package that reads the output files from LRT and renders each pixel in the raytraced image as an OpenGL point. The system sets the OpenGL viewport to be the same

size as the input image, to ensure that each input pixel maps to exactly one pixel on the real-time display. Because perspective correction and lighting effects have already been applied by the raytracer, it is necessary to render the pixels orthographically, with OpenGL lighting disabled. Thus, when rendered with no other objects in the scene, the OpenGL representation of the raytraced points will be an exact replica of the .tiff file output by the raytracer.

The system also allows other objects to be rendered graphically in the scene, along with the raytraced point-cloud. The goal of the system is to provide the illusion that these objects – which are dynamically rendered in real-time – are part of the raytraced scene. Thus OpenGL objects must be rendered with appropriate perspective correction, and they should occlude or be occluded by appropriate points in the point cloud.

In order to achieve appropriate perspective correction, the system reads the original scene file to determine the field-of-view angle used to generate the raytraced image. Objects are then rendered in OpenGL using a standard perspective projection matrix, initialized with the OpenGL function `gluPerspective( )`.

The values written to the depth buffer after OpenGL vertices are transformed through this projection matrix do not represent distance from the eyepoint. Rather, the depth buffer typically represents a nonlinear function of eyepoint-distance, intended to maximize precision in the range of distances near the eyepoint. The precise mapping of eyepoint distance z_{eye} through the projection matrix defined by `gluPerspective( )` or similar functions is:

$$z_{transformed} = z_{eye} \frac{zFar + zNear}{zFar - zNear} - 2\frac{zNear * zFar}{zFar - zNear}$$

…where $z_{transformed}$ is the resulting depth value that is passed on to the viewport transformation, z*Far* is the location of the far clip plane, and *zNear* is the location of the near clip plane.

The orthographic projection matrix used to render the point cloud is initialized using the OpenGL function `gluOrtho2D( )`. However, this projection matrix produces depth buffer values in an entirely different range than those generated by the `gluPerspective( )` transformation. Therefore, in order to force objects rendered in real-time to properly occlude and be occluded by raytraced points, it is necessary to modify this orthographic projection to produce depth buffer values that line up with those generated by the perspective transformation.

Specifically, we "manually" perform the above transformation on each pixel's depth value as it is read from the ASCII depth table produced by the raytracer. Thus the depth coordinate associated with each pixel is exactly what should be fed to the viewport matrix to ensure a proper comparison against the polygons rendered in real-time. In other words, we have pre-transformed each point's z-coordinate, so we want our orthographic projection matrix to transform only x and y, and not z. Therefore, we explicitly modify the standard `gluOrtho2D( )` projection matrix as follows before rendering our point cloud.

The immediate output of `gluOrtho2D( )` :

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{zFar-zNear} & -\frac{zFar+zNear}{zFar-zNear} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

…is modified to…

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

…where *right* and *left* are locations of the horizontal clipping planes, *top* and *bottom* are the locations of vertical clipping planes, and *zFar* and *zNear* are the locations of the near and far clipping planes.

Note that the modified projection matrix does not operate on the z-coordinate of each input vertex at all, so the correct depth values – computed "manually" for each raytraced point when processing the input file – are passed directly to the viewport transformation. Thus the objects rendered dynamically in OpenGL with a standard perspective projection "line up" correctly with the points generated by the raytracer, which are rendered orthographically.

The system also reads the positions of light sources from the scene file, to place OpenGL lights in the corresponding positions. Therefore, although much more sophisticated lighting effects are available for raytraced objects, the lighting of dynamically-rendered objects appears approximately correct in the scene.

2.3 Haptic rendering

This project was motivated by environments that are common to haptic rendering environments, so it was critical to include co-located haptic rendering in our application. The system parses the .rib file that was used for raytracing and extracts the locations of all polygons. These polygons are tessellated and rendered haptically as rigid triangles.

A dedicated thread reads the position of a haptic input device – either a SensAble Phantom (Massie and Salisbury, 1994) or a Force Dimension Delta (Grange et al, 2001) – and transforms the input device position into a device-independent range that is consistent with the location of polygons in the rendered scene. The gain of this transformation – which controls the "physical size" of the scene – can be set arbitrarily. The haptic thread continuously tests for penetration of any of the polygons extracted from the scene file, and generates forces to oppose any penetration of scene file objects. This is a standard technique used for haptic rendering of rigid surfaces (Salisbury et al, 1995).

The position of the haptic device is also rendered as an OpenGL sphere in the scene using the techniques described above. The result is that forces are generated when the visual representation of the device intersects raytraced polygons, and the user has a sense of physical interaction with raytraced objects.

2.4 Development notes

LRT development was done in Linux using GNU development tools.

Our rendering application runs in Windows, and was developed using Visual C++, MFC, OpenGL, and the SensAble Ghost API for haptic rendering.

3. Results

As a demonstration of the system's capabilities, we raytraced the classic "Cornell Box" (Goral et al, 1984) using photon-mapping (Wann Jensen, 1996), an effect not achievable in real-time, and read the corresponding scene file into our system. FIGURE 2 contains a screenshot of our running application. Note that the blue sphere – representing the current position of the haptic device – is rendered in real-time, but appears to be located between raytraced objects. Also note that the specular highlight on the sphere appears to be the result of a light source that is at the location of the light source in the raytraced scene.

A more convincing demonstration of occlusion and perspective transformation requires video, and a video of the sphere moving around in the scene can be viewed at :

`http://robotics.stanford.edu/jks-folks/hybrid.rendering/ball.in.box.rm`

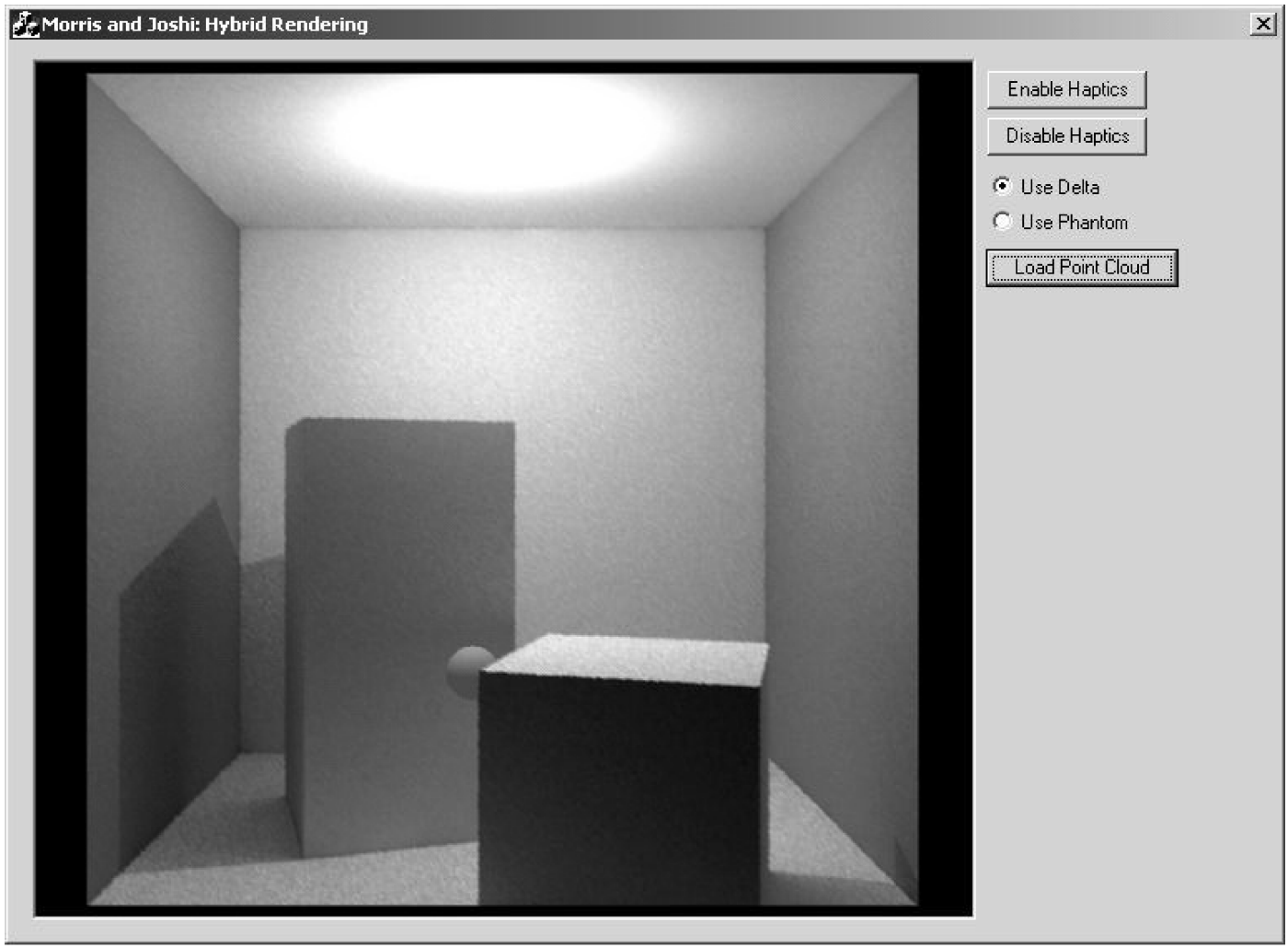

FIGURE 2: A screenshot demonstrating an OpenGL object rendered among raytraced points.

4. Conclusion

4.1 Applications

We present a novel technique for aligning pre-rendered scenes with objects rendered in real-time. This is significantly more powerful than simply texturing complex images onto polygons in an OpenGL scene, since it allows occlusion among objects rendered in OpenGL and objects rendered offline.

We also demonstrate a potential application of this technique by allowing a user to haptically and graphically interact with raytraced objects in real-time. We expect that this will be a valuable approach to demonstrating and exploring potential applications of haptic environments.

4.2 Future work

In order to increase the degree to which dynamically-rendered objects are truly interacting with raytraced objects, we would like to incorporate real-time shadow-casting. Since we have the

location of the light source and the original polygons, it would be possible to cast projective shadows onto those polygons and render translucent polygons that would partially occlude the point cloud. The effect would be real-time shadowcasting onto raytraced objects.

Additionally, we would like to extend this work to allow haptic annotations (roughness, rigidity, etc.) among the material properties specified in the original .rib file. This could be the basis for a scene file format that describes both haptic and visual properties of rendered objects.

References

Goral C.M., Torrance K.E., Greenberg D.P., and Battaile B. "Modeling the interaction of light between diffuse surfaces." Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, 1984.

Grange S, Conti F, Helmer P, Rouiller P, and Baur C. "`Overview of the Delta Haptic Device". Eurohaptics '01, Birmingham, England, July 2001.

Massie T.H. and Salisbury K. The Phantom Haptic Interface: A Device for Probing Virtual Objects. In Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interface for Virtual Environments and Teleoperator Systems (Chicago, IL), 1994.

Pharr M and Humphreys G. *Physically Based Rendering: Design and Implementation of a Rendering System*. 2003 (unpublished manuscript).

Ruspini D.C., Kolarov K, and Khatib O. "The Haptic Display of Complex Graphical Environments." Computer Graphics Proceedings, Annual Conference Series, 1997.

Salisbury K, Brock D, Massie M, Swarup N, and Zilles C. "Haptic Rendering: Programming Touch Interaction with Virtual Objects." Proceedings of the 1995 Symposium on Interactive 3D graphics.

Upstill S. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. 1990.

Wald I., Purcell T.J., Schmittler J, Benthin C, and Slusallek P. "Realtime Ray Tracing and its use for Interactive Global Illumination." Eurographics State of the Art Reports, 2003.

Wann Jensen H. "Global Illumination using Photon Maps." In "Rendering Techniques '96", 1996.