TEXTURED HIERARCHICAL PRECOMPUTED RADIANCE TRANSFER

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Harrison Lee McKenzie Chapter

June 2010

COMMITTEE MEMBERSHIP

TITLE:	Textured Hierarchical Precomputed Radiance Transfer
AUTHOR:	Harrison Lee McKenzie Chapter
DATE SUBMITTED:	June 2010
COMMITTEE CHAIR:	Zoë Wood, Ph.D.
COMMITTEE MEMBER:	Aaron Keen, Ph.D.
COMMITTEE MEMBER:	Chris Lupo, Ph.D.

Abstract

Textured Hierarchical Precomputed Radiance Transfer

Harrison Lee McKenzie Chapter

Computing complex lighting simulations such as global illumination is a computationally intensive task. Various real time solutions exist to approximate aspects of global illumination such as shadows, however, few of these methods offer single pass rendering solutions for soft shadows (self and other) and inter-reflections.

In contrast, Precomputed Radiance Transfer (PRT) is a real-time computer graphics technique which pre-calculates an object's response to potential incident light. At run time, the actual incident light can be used to quickly illuminate the surface, rendering effects such as soft self-shadows and inter-reflections.

In this thesis, we show that by calculating PRT lighting coefficients densely over a surface as texture data, additional surface detail can be encoded by integrating other computer graphics techniques, such as normal mapping. By calculating transfer coefficients densely over the surface of a mesh as texture data, greater fidelity can be achieved in lighting coarse meshes than simple interpolation can achieve. Furthermore, the lighting on low polygon objects can be enhanced by drawing surface normal and occlusion data from highly tessellated, detailed meshes. By applying such data to a decimated, simplified mesh, a more detailed and visually pleasing reconstruction can be displayed for a lower cost.

In addition, this thesis introduces Hierarchical PRT, which extends some surface effects, such as soft shadows, between objects. Previous approaches to PRT used a more complex neighborhood transfer scheme in order to extend these lighting effects. Hierarchical PRT attempts to capture scene information in a tree data structure which represents coarse lighting relationships between objects. Potential occlusions can be found at run time by utilizing the same spherical harmonic representation used to represent surface lighting to instead store light “filters” between scene tree nodes. Such “filters” can be combined over a set of nodes in the scene to obtain the net shadowing of an object with good performance.

We present both visually pleasing results on simplified meshes using normal mapping and textured PRT and initial results using Hierarchical PRT that captures low frequency lighting information for a small number of dynamic objects which shadow static scene objects with good results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Computer Graphics Foundation

Computer Graphics is the art of transmuting models and light into images with the quicksilver of mathematics. Since its foundation in the early 1960's, modern Computer Graphics has expanded to encompass areas ranging from the diverting fields of video games and cinema special effects to the elevated disciplines of computer aided design and medical visualization[13]. Though the types of images that are produced for these areas are understandably different, the overriding goal is always to produce the highest quality image possible from a set of data and resources.

In addition to the different subjects of the images produced for the different fields Computer Graphics serves, so too does the "budget" allocated to produce a given image vary with the discipline. The applications used in the arenas of CAD and film may acceptably run for hours or even days, utilizing complex simulation tools and extremely high quality models to produce very realistic results. Highly responsive applications on the other hand, such as those found in the fields of computer gaming and scientific imaging may be forced to operate at real-time speeds on resource constrained or mobile devices.

The disparity between these different application domains highlights an essential difference in the Computer Graphics techniques utilized to fulfill their needs: *Global Illumination* or *Real-time Graphics*. The former offers the realms of complex modeling and simulation of light at the price of space and time, while the latter first trades all for interactive frame rates, and then builds back what it can though clever rendering techniques.

1.2 Real-time Rendering

In order to achieve real-time rendering rates, Computer Graphics techniques traditionally rely upon a *local approximation* of light interacting with a surface, rather than modeling how light might affect a given scene holistically. Many shading models, such as that developed by Gouraud, adopt this simplified strategy for coloring components of an image after rasterization[1]. This method, and Gouraud shading, is in fact the nominal one utilized in the conventional *fixed function* graphics pipeline implemented on most graphics hardware (GPUs). Though useful and fast, it often produces images like the bland, flat results of Figure 1.1.

As Computer Graphics has become more advanced, however, additional elements have been added to real-time rendering methods to enhance their representation of an artist's vision. Things like texture mapping and bump mapping can apply pre-made, "continuous" images to geometry, greatly increasing the flexibility of simple, flat triangle in the quest to represent and recreate images of complex objects. Still, simple textures or normal perturbations, regardless of their size and detail, do not take the environment surrounding an element of the scene into account.

The key aspect which these simple techniques fail to take into account is the relationship *between* objects in the scene, beyond the simple adjacency of their triangles. Instead of simple images clumsily wallpapered on to triangles, real-time graphics greatly desires the effects from Global Illumination. Effects such as shadows, which convey the spatial quality of one element being in front of another, or reflections, which convey the way light interacts between two different objects are prevalent in Global Illumination. The human eye is very adept at interpreting the these effects, enunciated as color, to reconstruct where all of the objects in a scene are "actually" located.

While, as mentioned before, it is generally impractical to run Global Illumination strategies such as Ray-Tracing or Radiosity in real-time, for some scenes the final effects of such algorithms can be precalculated and utilized. For instance, static scenes may use textured "light map" images which represent how much light reaches a particular place in the scene. Though the assumptions necessary for such interactions are weighty – nothing can move – the visual effects granted, such as soft shadows, are very useful in improving the quality of the representation of a scene.

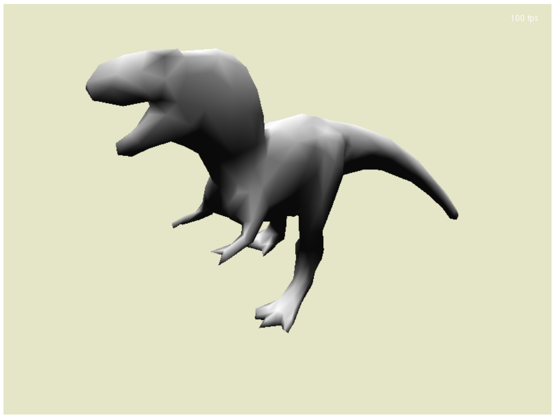


Figure 1.1: Tyrannosaurus with Gouraud shading

1.3 Precomputed Radiance Transfer

Though the techniques presented above provide many tools to real-time application developers to represent interesting effects over the surfaces of scene elements, the limited trade-off of local approximation and static-geometry is too limited for many domains. Many modern games, for instance, sell themselves on dynamic, mutable environments which are rendered in life-like detail and fantastic quality[3]. Precomputed Radiance Transfer is a technique which attempts to solve part of this issue by precomputing how an object is lit by *potential* light from different directions. This approach differs from the precomputation done with light maps because the *actual* lighting is not known ahead of time – rather, the precalculated potential effects are combined with the actual lighting at runtime.

Precomputed Radiance Transfer (PRT) can give many of the effects desired from global illumination, such as soft shadows and interreflections, affected dynamically by the changing lighting environment in the scene. Unfortunately, by itself the technique still falls short of providing effects between objects – PRT calculations are typically done only with respect to a given object, giving soft *self*-shadows, and *self*-interreflections. Further, if these calculations are only done on a per-vertex basis over an object, a dense, highly tessellated mesh will still be required to generate smooth lighting over the surface of the object. As additional geometric primitives engender more work to render, this is undesirable.

By creating PRT data densely over a surface as texture data, however, it is possible to continue using less geometrically intense representations of scene elements while maintaining the complex lighting effects, such as self-shadowing, which PRT allows. This thesis explores the generation of dense, Textured PRT data over meshes, as well as the storage, transfer, and rendering of that data. We demonstrate the flexibility which such sampling engenders over more standard per-vertex PRT methods, as well as incorporating other surface reconstruction effects, such as normal mapping.

The dense sampling of PRT data does not, unfortunately, allow interactions between objects. To facilitate this process and give scene objects the ability to interact with one another by transferring light between objects in scene "hierarchies," this thesis develops and explores the concept of rendering Hierarchical PRT scenes. We attempt to use this representation of a scene to quickly calculate the appropriate shadows for nodes in the scene while minimizing the necessary ray casting.

Chapter 2

Related Work

2.1 Polynomial Texture Maps

Polynomial Texture Maps, developed by Malzbender et al at Hewlett-Packard Labs[9], was a technique which attempted to isolate the aspects of color and illumination at a point on a surface to improve the relighting of objects with better storage efficiency. Their work developed sets of textures which were used in combination to shade a given point on a texture parameterized model by isolating the Luminance Model of the texture from the underlying surface properties at points on a surface.

Malzbender et al's work was based on the development of fast and correct color representation of actual objects under different lighting condition, captured with a camera rig. This, however, prototypes the isolation of a model's surface shading characteristics from the actual incident lighting, in many different directions. This isolation mirrors the approach later taken by Sloan et al with the Spherical Harmonic representations used in PRT[15].

2.2 Precomputed Radiance Transfer

2.2.1 PRT Basis

Sloan et al introduce Precomputed Radiance Transfer in “Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments[15].” In this work, the separation of how a mesh receives environmental light and the actual incident light are decoupled, and modeled as two separate Spherical Harmonic representations. The sampling of PRT coefficient vectors over a mesh and the subsequent storing of such data in textures is mentioned, but not extensively elaborated on. One component which is noted is the need for advanced, floating point texture profiles to successfully utilize the established PRT data. Beyond this, Sloan et al focus on establishing other components of PRT, rather than the specific flexibility provided by a textured representation.

2.2.2 Neighborhood Transfer

In [15], Sloan et al also develops a concept of “neighborhood transfer” which is used to light objects which are nearby other ones in the scene. The method presented samples the spatial volume *around* an object ahead of time, and uses that set of coefficients to light an object which is placed nearby. Unfortunately, due to the unknown properties of the recipient object, a transfer matrix, rather than a simple vector of SH coefficients, must be stored to light an object placed inside the resulting volume. Further, this method encounters problems in combining the shadowing from multiple volumes onto a single object.

2.3 Shadowing

2.3.1 Shadow Volumes

In addition to the self-shadowing aspects presented in [15], we leverage research done by Crow in [2] on inter-element shadowing as a model for conventional real-time shadowing. The "shadow volume" technique introduced by [2] is updated by Heidmann to be performed on the GPU[6], and further by Kilgard et al to utilize more modern graphics hardware[5]. Shadow Volumes utilize the stencil buffer of modern graphics hardware over multiple rendering passes to produce the hard, crisp shadows which we compare our hierarchical shadowing work against.

2.3.2 Shadow Mapping

Another technique used to render real-time shadows on modern graphics hardware is Shadow Mapping. Like Shadow Volumes, Shadow Mapping produces hard shadowing effects by executing multiple rendering passes of a scene from different perspectives[1]. We do not utilize results from this method explicitly, but it is indirectly useful as a comparison tool in terms of approach – our work focuses on a single pass shadowing solution, where both Shadow Volumes and Shadow Mapping utilize multiple rendering passes of the scene to produce a single frame.

Chapter 3

Background

In order to understand the migration of Precomputed Radiance Transfer data into textures and hierarchies, first a foundation in some core mathematical concepts is required. These systems form the basis for framing the problem PRT solves, as well as providing the tools to represent the solutions efficiently and compactly. In this chapter, we will first move through a generalized enunciation of Computer Graphics lighting and then to the tools which are used to make that representation computable.

3.1 The Rendering Equation

The Rendering Equation is an integral representing the light emitted at a point on a surface as a function of the total light received by the surrounding environment, and its impact on the surface from each direction[8]. As noted in Kajiya's seminal work, the Rendering Equation generalizes many different lighting approximations used in Computer Graphics. It is evaluated as an integral over the sphere surrounding a point, such that the illumination seen, $I(x, x')$, when viewing the surface at x' from a particular point x is:

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

This equation serves as a starting place for developing the surface and lighting representations used in Precomputed Radiance Transfer. Specifically, the presence of the integral over a sphere encourages the use of a spherical representation which can be evaluated quickly to represent all possible incoming directions of light.

3.2 BRDF

In addition to the spherical lighting representation encouraged by the Rendering Equation to gather incident light over a surrounding environment, the Rendering Equation includes corresponding spherical information about the surface being lit. The surface properties, such as color, diffuse reflectance, specular reflectance, and angular reflectance relationships are embodied in the BRDF of the surface – the $\rho(x,x',x'')$ in the equation above. The BRDF stands for the *Bidirectional Reflectance Distribution Function*.

The BRDF of a surface is a generalization of the *material properties* more often used in real-time graphics. While the real-time pipeline often uses simple BRDF models, such as a simple Lambertian cosine weighting coupled with diffuse and specular colors, the properties of real materials may be much more complex. In the $\rho(x,x',x'')$ equation, ρ is a function which maps a given input radiance at point x', from point x'', to an exiting radiance in the direction of x[8]. By utilizing precomputation, potentially complex BRDFs may be used for the surfaces being processed.

As described, some surface BRDFs may be much more complex than others. One useful distinction in surface characteristics is the difference between diffuse and specular light reflection. In diffuse lighting, the colorization seen on a surface is independent of the viewing direction – light is reflected uniformly off the surface, regardless of the light's incident direction. Specular lighting over materials, such as the lighting found on glossy, metallic objects, is reflected much more strongly in some directions based on the incident angle to the surface.

Though considerable successful work has been done to incorporate glossy surfaces into Precomputed Radiance Transfer, the additional "degree of freedom" required to represent the dependence of the color seen on the viewing angle necessitates an additional dimension in the PRT representation. This expansion increases the volume of data which is required to represent a surface considerably. Though specular contributions are quite valuable, our work focuses on representing diffuse materials densely over mesh surfaces, and excludes the additional dimension required to represent specular materials.

With the removal of the dependence on viewing direction, the BRDF of a surface's material can be thought of as a spherical function, which captures light from distinct sections of the surrounding environment, and adds them to the final color seen from any view direction. The spherical nature of the BRDF, the incident lighting, and the interaction between the two will play an important role in selecting the "encoding" of a surface's lighting characteristics.

3.3 Monte Carlo Integration

While the Rendering Equation is an elegant formulation of how the incident light at a point on a surface is related to its surrounding environment, the presence of the definite integral is problematic for the actual evaluation of the equation. In order to solve the integral over the sphere, or more generally, to represent a continuous spherical quantity, we use a Monte Carlo integration technique with stratified sampling [16].

To perform Monte Carlo integration, a spherical sampler is generated by reparameterizing a unit length, square grid. Each sample, however, is "jiggled" by a small, random amount. The i and j dimensions of this square grid are used to generate spherical coordinates on the unit sphere using the mapping:

$$\theta = 2cos^{-1}(\sqrt{1-i})$$

$$\phi = 2\pi j$$

To make these coordinates more spatially useful later in the procedure, we store the Cartesian coordinates representation of each sample by reversing the spherical coordinate mapping via:

$$x = sin(\theta)cos(\phi)$$

$$y = cos(\theta)$$

$$z = sin(\theta)sin(\phi)$$

As the density of this sampling increases, the samples gradually approximate the complete surface of a sphere. Using this set of sampling vectors, spherical functions (those parameterized by θ and ϕ) can be approximated using the value of the function in a particular sample direction. By summing the value of each sample ω, weighted by the number of samples n, we can approximate the integral of a function over a sphere:

$$\sum_{i=1}^{n} f(\omega)\left[\frac{4\pi}{n}\right] \approx \int_S f(\omega)d\omega$$

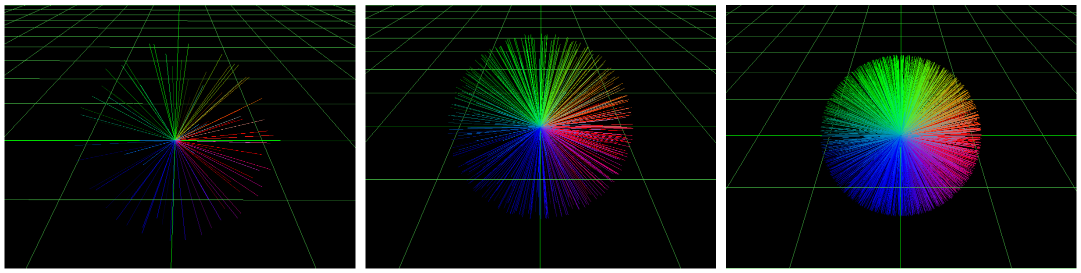


Figure 3.1: 100, 625, and 2500 component Sampler

Using a set of samples, the Monte Carlo integration method allows us to discretize the continuous problem of integrating the lighting over the sphere surrounding a point on an object, allowing us to solve the rendering equation when we have a spherical lighting function and spherical receiving function.

As described above, increasing the sampling density causes the resulting samples to more closely represent a sphere (as visualized in Figure 3.3). This also engenders the practical consideration, however, of speed – more samples means more calculations required to compute an integral. During the course of development, we've found that while sampling sizes of around 10,000 give very nice reconstructions, they typically encounter performance hardships when doing some run-time calculations. Between 625 samples and 2500 samples, however, still give fairly good results without being overly computationally taxing.

3.4 Spherical Harmonics

3.4.1 Introduction

Spherical Harmonics is a mathematical construct used to represent functions which are defined over the surface of a sphere. While the Monte Carlo method is useful for discretizing the integral presented earlier, attempting to store an intermediate spherical representation using samples alone is highly impractical for any useful number of sample directions. Instead, Precomputed Radiance Transfer typically *projects* functions into a spherical representation once they are sampled.

Spherical Harmonics, more formally, are the solution to Laplace's equation over the surface of a sphere – more simply, however, they can be thought of as the Fourier basis evaluated on a sphere, rather than a unit circle[15]. A spherical harmonic representation consists of a number of polynomial coefficients, representing the contribution of a particular basis function to the final spherical model. By adding up all the different components of the spherical representation, a relatively complex spherical function can be modeled.

To control the granularity of a spherical representation, Spherical Harmonics may be evaluated at a set number of "bands," which controls the fine-ness of the representation. Low band representations require less coefficients to construct, but can only adequately represent functions which have a low degree of variability. Higher band representations require more coefficients to be stored, but provide a more flexible reconstruction with greater fidelity due to the higher degree polynomial used [14].

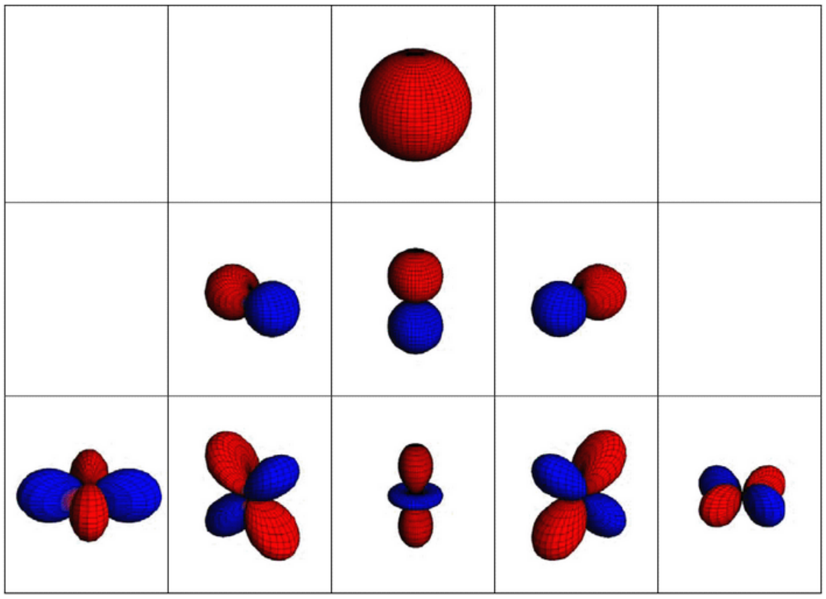


Figure 3.2: Visualization of Spherical Harmonic Bands [14]

3.4.2 Basis Functions

Meshing with the Monte Carlo integration method above, the Spherical Harmonic basis functions are parameterized over a sphere using (θ, ϕ). To project a function into spherical harmonics using the Monte Carlo style sampler described, we first evaluate the coefficients of the spherical harmonic basis functions for each sample direction. Since we have both the (x, y, z) and (θ, ϕ) values for each sample, we can evaluate the basis functions using:

$$y_l^m = \begin{cases} \sqrt(2) K_l^m cos(m\phi) P_l^m(cos\theta) & : x > 0 \\ \sqrt(2) K_l^m sin(-m\phi) P_l^{-m}(cos\theta) & : x < 0 \\ K_l^0 P_l^0(cos\theta) & : x = 0 \end{cases}$$

By examining the above equations, we can see that absent the constants $(K_l^m)$and recurrence relations (P_l^m), the samples are simply parameterized using the expected cosines and sines of θ and ϕ. The variation of bands l, and the individual basis functions in that band m, control the fine-ness of each (θ, ϕ) parameterization in a particular direction at a particular level.

3.4.3 Properties

SH Integration

In addition to being a convenient spherical encoding for the necessary spherical lighting and receiving functions, the Spherical Harmonic basis has other nice properties which make it amenable to our purpose. Most notable, the integral of the *product* of two Spherical Harmonic projections is simply the "n-dimensional" dot product of their respective coefficient vectors:

$$\sum_{i=1}^{l^2} f_i g_i \approx \int_S f(\omega) g(\omega) d\omega$$

Recall from the Rendering Equation that the shading seen at a point is proportional to the lighting coming in from a particular direction multiplied by the amount transferred in the viewer's direction from that point, given a particular input, over the entire surrounding area – a sphere. The quantity being integrated, then, in the Rendering Equation is exactly the product of two spherical functions. Thus, once we have projected a mesh's receiving function into the Spherical Harmonic basis, and calculated the incident radiance in terms of Spherical Harmonics as well, the integral of the two is a fast summation over the products of a small number of coefficients. The simplicity of this operation can be contrasted with the possible, but more "heavyweight" representation of summing the products of the samples used to construct both Spherical Harmonic representations.

SH Rotation

Another important property of Spherical Harmonics which makes them useful for Precomputed Radiance Transfer is invariance over rotations. Once the Spherical Harmonic projection of a function is established, the resulting representation can be "rotated" to an arbitrary orientation using a matrix, much like ordinary vector quantities are rotated in computer graphics. Though the matrix required is more complicated – a sparse NxN matrix[12], where N is the number of coefficients used in the SH representation – this property allows precomputed data to be transformed to move an object around the scene in different orientations, rather than requiring a recalculation of the SH projection.

Other potential basis functions, such as Haar Wavelets, do not exhibit this property. The resulting systems, such as Sun and Mukherjee's work with Precomputed Radiance Transfer for glossy objects, permit only transformations such as translation and uniform scaling, greatly restricting the versatility of the resulting application[17]. Though not exhibited specifically in our work, this valuable property is an important component which influenced the choice of basis functions used for our system and for the work done by Sloan et al[15].

3.5 Shadows

Shadows are an important tool in art and Computer Graphics as a way of expressing spatial data in a scene. While they are often computationally difficult to create using the conventional graphics pipeline, they highlight important relationships to the viewer with respect to brightness, shape, and position.

3.5.1 Soft Shadowing

When rendering shadowed areas, there is an important division in the quality of shadows which are produced by different techniques. *Hard* shadows are those produced such that the distinction between darkness and light is very abrupt – as if light were suddenly and totally cut off from reaching beyond the edge of some piece of occluding geometry. While these types of shadows are highly illustrative of the shape generating the shadow, they can also look false or unsettling[1]: light generally does not stop instantly, nor does it generally travel in the solely linear paths that such patterns suggest.

Soft shadows often appear more natural, and are created when the amount of light reaching around the edge of an object is eased more gradually. Physically, the difference can be seen in an examination of the light casting the shadow. Lights in the real world are generally not generated at a single point in space; rather, they are illuminations of a small area: even a light bulb has a surface with some extent in the world greater than an "infinitely small" point. The extent of the light which casts a shadow is an important feature: as the light's surface is traversed, the relationship between the light and the "silhouette edge" of a shadow casting object changes. This small but significant differential angle causes changes in the light which makes it past an occluding object near the edges.

The small changes in the amount of light which "reaches around" the silhouette edges of objects forms the *penumbra* – the place where only part of the energy from a light hits. The granularity between all-or-nothing lighting often makes soft shadows appear much more natural than their crisp cousins, and is one of the motivating factors for the work on our Textured PRT and Hierarchical PRT techniques.

Chapter 4

Textured PRT

4.1 Introduction

Textured Precomputed Radiance Transfer represents a progression of PRT Sampling from a more nominal per vertex basis to a much higher fidelity representation over the surface of a mesh. To frame our contributions to the projection and storage of Precomputed Radiance Transfer coefficient vectors into texture space, as well as the additional techniques this procedure facilitates, we will first present an overview of the general PRT algorithm when computed in texture space. For clarity, the precomputation and run-time steps will be divided into discrete sections.

4.1.1 Precomputation Overview

To begin, the preprocessor accepts a set of meshes with texture coordinates covering all the viewable triangles of the mesh, along with the size of the desired textures. For each triangle, the preprocessing system evaluates the triangle's extents in texture space, and iterates over each texel. For each texel, the object space location is computed by interpolating over the triangle's face in barycentric coordinates. Using an associated normal, a vector of SH transfer coefficients is computed, representing how that point is influenced by potential incident light. These coefficients are stored separately in an array of textures, with each coefficient "index" being represented by a discrete texture. Finally, post-processing is done to ensure correct evaluation of the images when texels are looked up with linear filtering routines.

In review:

1. Select the number of SH bands to be computed, and the density of samples to be used
2. Acquire mesh data with per-vertex texture coordinates for each face
3. Process extents of each face in texture space
4. Interpolate object space location over the face
5. Compute PRT transfer coefficient vector per texel
6. Write that texel's coefficient vector to the texture array
7. Store one coefficient per texture, for each texel on the original face
8. Apply post-processing to the textures to eliminate potential gaps

4.1.2 Rendering Overview

To render the precomputed surface data, a mesh is loaded into memory along with its associated coefficient textures. These textures are transferred to the

Graphics Processing Unit (GPU) hardware once, and stored as a two-dimensional array texture. For each frame, the incident light at a number of sample points on the mesh is calculated, and projected in to SH coefficients. These lighting coefficients are encoded in a temporary one-dimensional array texture, and sent to the GPU. The triangles of the mesh are then sent to the graphics hardware to be rendered, along with an associated set of "contribution" factors for each lighting sample. A fragment program is executed per pixel which is rasterized to the screen, which finds the color by computing the "n-dimensional" dot product of a fragment's model coefficients (looked up in the two-dimensional texture), and the weighted average of the lighting sample coefficients (looked up in the one dimensional texture).

To review once more:

1. Load mesh into memory
2. Load associated texture data onto the GPU
3. For each frame, sample incident radiance over the mesh
4. Encode lighting samples into a temporary texture, and send to GPU
5. Pass the triangles of the mesh to the GPU for rendering
6. For each resulting fragment pixel, look up the textured model coefficients
7. Find the lighting coefficients, based on the contribution of each lighting sample
8. Compute the dot product of the model transfer coefficients and lighting coefficients.

4.1.3 Contributions

In addition to elucidating the method of computing, storing, and utilizing textures to store a mesh's transfer coefficients (as this approach is given little coverage in other works), we present a method of coefficient storage which removes Sloan et al's requirement of more complex, floating point texture profiles for model coefficient storage[15].

Furthermore, we present a technique for utilizing high quality mesh data to compute the PRT representation over low polygon count, decimated meshes. This technique allows lower polygon count meshes to be used to increase performance without a substantial loss in visual quality.

Finally, we present an evolution in the storage of multiple lighting samples by encoding them in a texture, rather than passing a single sample for the entire mesh as a simple uniform array of floating point values.

These specific additions are found in the computation of the transfer vectors over the surface of the mesh, the post-processing of the texture data, the loading of a fragment's model coefficients from the two-dimensional array texture, and the transfer of lighting sample coefficients to the GPU.

4.2 Precomputation

4.2.1 Band and Sample selection

As described earlier (3.4.1), the number of bands used to project a function into Spherical Harmonics has a direct effect on the quality of the resulting reconstruction. At this stage of the procomputation pipeline, band selection influences how many textures will be allocated (as one texture is required for each SH coefficient), and how many basis functions will be evaluated for each sampling direction.

Also as described in the preceding section (3.3), a Monte Carlo sampling system is required to project mesh characteristics into Spherical Harmonics. At this stage, we construct a "sampler," to the *square* of a user specified density, using the stratified sampling technique presented earlier to generate both spherical and Cartesian coordinates per sample. For every sample constructed this way, the associated coefficients of there Spherical Harmonic basis functions are computed in that sample's direction. These coefficients are stored along with the sample, and used later to "add" that sample's value to the final SH representation at a particular texel.

4.2.2 Mesh Acquisition

Once all samples are prepared, meshes requested by the user are read in from the disk and stored in memory as part of a "scene." This scene representation allows multiple meshes to be processed relative to each other, rather than in isolation, if desired (the usefulness of this will be presented later in the work). For each object in the scene, a number of textures are allocated and zeroed, based on the number of bands requested earlier. The size of these textures is likewise a user defined parameter, though to integrate nicely into with conventional Computer Graphics hardware, they should be sized to powers of two. All meshes in the scene, if more than one mesh is being processed, are computed at the same number of bands and texture size.

At this stage, if a more geometrically complex set of surface normals are available in a normal map associated with any mesh being loaded, these are loaded as a texture and associated with the corresponding mesh. These normals, if available, are used later when computing the normal at a texel location on a triangle's face.

4.2.3 Face Processing

For each mesh, all its faces are iterated through. The texture space coordinates of each face's vertices are used to find the triangle's extents in texture space, given the resolution of the textures being produced. Once these locations are determined, the texels which represent that triangle are found using a barycentric method which finds if a texel position is inside or outside of the determined triangle extents. Once found, each texel is processed individually to find its object space position as well as its associated normal.

4.2.4 Texel Processing

As the texels under consideration are along the planar face of the triangle in object space, the position of *applying* that texel to the face of that triangle can be determined using again using barycentric coordinates. Barycentric coordinates represent a position over the face of the triangle as a linear combination of the three vertices which define the triangle[13]. By applying those same weights to the normals which are associated with each triangle vertex, the normal which closely represents that point on the triangle can also be found. This normal, however, will simply be a "smooth" progression of the three vertex normals over the surface of the face, and not represent any more complex geometric detail.

During this step, if a more accurate geometric representation of the mesh is available, we can use the surface normal associated with that representation instead. This approach closely mirrors that taken when normal mapping surfaces to represent more complex geometry that is actually being rendered [1]. Therefore, if we have a normal map available which contains more complex surface normals, use lookup inside that mapping to find the surface normal at a point on the face, rather than interpolating the normal. This approach gives a more detailed final rendering with simplified meshes than is possible by a simple interpolation of the vertex normals. Furthermore, because we are operating on a texel already parameterized with texture coordinates, the location to look up in the normal map are already computed.

4.2.5 Coefficient Computation

Now that the position, normal, and texel coordinates are known, we compute the Spherical Harmonic representation of the point's transfer function – how that point receives light from the surrounding environment. As touched on earlier in (3.1), the object's surface properties (the BRDF) and visibility need to be taken into account for each potential direction of the surrounding sphere. Utilizing the set of Monte Carlo samples, we perform the SH projection into a temporary vector of transfer coefficients.

For every sample direction in the sampler, we cast a ray out from the target position into the scene. If the ray does not strike any objects, we add that sample's SH basis function coefficients to our temporary vector of transfer coefficients. This ray cast takes into account the "geometry" term of the Rendering Equation which limits how certain lighting directions may or may not contact the surface. This "occlusion" of certain lighting contribution directions is what gives the effect of soft self-shadows over the surface of the mesh.

As of yet, however, we have not taken into account the BRDF of the surface relative to the direction being sampled. To do so, in addition to a binary "on-off" contribution change from the ray casting procedure, we weight each sample's basis functions by the cosine of the angle between the sample direction and the surface normal, clamped to zero. This simple but effective model for the BRDF, also known as the Lambert's Cosine Law, weights the effect of potential lighting contacting the surface "obliquely" lower than lighting directions which contact the surface "straight on."

4.2.6 Coefficient Storage

Once all directions in the Monte Carlo sample have been evaluated at the position corresponding to a given texel, the coefficients must be stored in the textures to await post-processing. Because the texture space coordinates are already available, we compute the texel location (given the established texture resolution), and store each coefficient of the temporary transfer vector into a the texture at its corresponding index.

4.2.7 Texture Post-Processing

Once the transfer coefficients have been calculated across all faces for a mesh in the scene and stored in their corresponding textures, a “push-pull” algorithm is applied to smooth the edges of triangle boundaries in texture space. This procedure is necessary to avoid sampling errors with linear texture filtering – four texels around a desired area may be sampled, and sharp divisions between areas with valid data and areas with no data can generate unsightly discontinuities in the final rendering.

To perform the “push-pull,” each texture is considered, and the texels which have been written to are marked. If a texel has no been written to, but one or more neighbors of the texel have, those surrounding texel values are averaged and used to color the unwritten texel. This simple algorithm eliminates discontinuities on the edges of triangle patches in texture space, without compromising the validity of data at those points – due to averaging, the texel should have approximately the same value when linearly filtered as it would otherwise receive. The case of linear filtering including “black texels,” or texels which have *no* valid value, in the final lookup process, is ameliorated.

After the “push-pull” procedure ensures that no errant texels influence the results via linear filtering, the range of the coefficients is compressed and shifted into the standard range of conventional textures: [0, 1]. For each texture being produced, the range of values which occurs in that texture is found by iterating

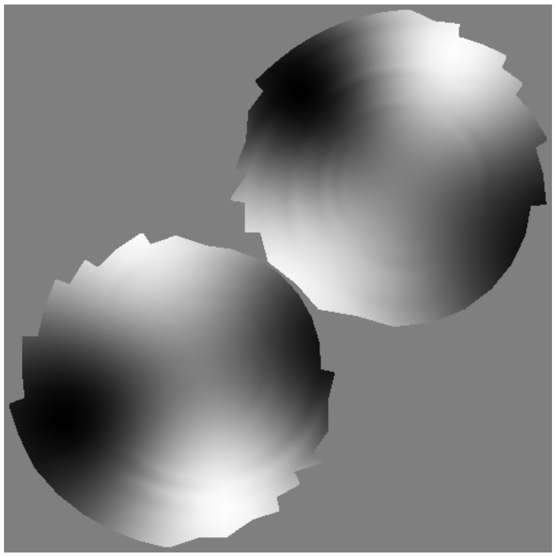


Figure 4.1: Textured coefficient storage over a sphere, at band four

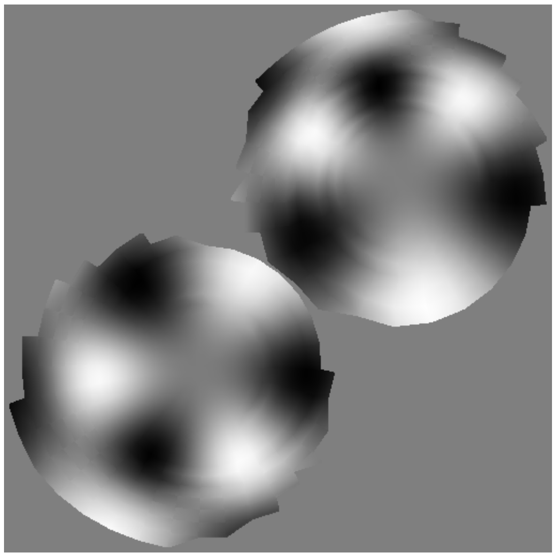


Figure 4.2: Textured coefficient storage over a sphere, at band twenty three

over all texels in the image. Once the minimum and maximum values are found, every value in the texture is transformed by the formula:

$$T = \frac{T_o - min}{max - min} = \frac{T_o - min}{range}$$

where T_o is the original value at a given texel and T is the final transformed value.

To facilitate the reconstruction of actual values during rendering (described below), the minimum and maximum values used to perform the range compression on each texture are stored in a separate file. These range compression scheme removes the dependence of performing textured PRT on advanced, floating point GPU profiles, which are required in Sloan et al's work[15], and minimizes difficulties in pushing the texture data in to the Graphics pipeline. Additionally, this technique permits the viewing of each separate coefficient texture as a simple image. This allowance has proved to be an invaluable tool for visualizing the data which is generated over a mesh with the naked eye, allowing a more intuitive view for debugging purposes. Additionally, this simple storage mechanism may open up more image based techniques, such as smoothing, sharpening, or filling, for future development.

A necessary note about the range compression technique used to confine the data to the [0, 1] range is the potential for resolution loss. Converting the coefficient data from normal, thirty-two bit floating point data into eight bit texture data in a small range represents a significant potential limitation on the values which can be reconstructed. In practice however, the floating point values displayed before compression were found during experimentation to rarely extend beyond the range of [-4, 4]. Furthermore, the discretization of the Monte Carlo sampling technique used, coupled with the relative smoothness of most surfaces, generated data without a tremendous number of steps. These two observed conditions serve to make the compression to the standard [0, 1] texture range more palatable.

As an additional anecdote about the suitability of the [0, 1] range compression, the use of thirty-two bit textures was considered and used briefly in testing. During this brief period, significant visual discrepancies were not observed compared to the range compressed data, the direct viewing of the textures was impossible due to the non-standard image format, and the texture data was four times as large due to the increased number of bits per texel. We therefore consider this scaling technique to be quite useful to the goals of visual and performance goals of textured PRT.

4.2.8 Further Preprocessing Considerations

The above procedure continues for all meshes, until all the faces in the scene have been processed, and their corresponding texels written. After the post-processing step, these textures are written out to disk, named corresponding to the mesh which they were generated to cover. The additional scaling data for the texture coefficients required to "inflate" them back into their normal range is also written out, named likewise based on the mesh the associated textures are covering.

An additional important consideration of this preprocessing procedure is the strict separability of all face processing tasks and texel processing tasks. Once the scene has been set up and the textures allocated, no components of this algorithm require data from any other part, which makes the system respond very well to extremely simple parallelization strategies. In our evaluations, we have seen a linear decrease in precomputation time with the number of processors allocated to the task.

4.3 Rendering

4.3.1 Basic Renderer Setup

In a process mirroring the procedures used in the preprocessing system, the number of Spherical Harmonic bands, sampling density, and possible meshes are selected when the application is initialized. The number of bands being used for rendering should match the number of bands utilized to preprocess the mesh, though the sampling density may vary to permit more granularity in balancing run-time performance with representation quality.

4.3.2 Texture Setup

During the loading of each mesh requested, the associated texture images are loaded into memory, for transfer to the GPU. To simplify this process, a two-dimensional array texture is pre-allocated without data and filled incrementally with each texture. The size requested from the graphics hardware is effectively the product of the images' width and height, multiplied by the number of bands being used. As each texture is loaded from disk, it is passed as a array of unsigned bytes to the graphics hardware, used to set a particular index of the texture array, and de-allocated. When all textures have been loaded, the texture handle received from the graphics hardware is stored with the mesh, and used to refer to the “stack” of texture images now resident on the GPU. This data is not transferred to the graphics hardware again over the lifetime of the application unless GPU texture memory becomes insufficient to store the data required by the application.

In addition to the array of images loaded on to the graphics hardware, the scaling factors used to “inflate” the range compressed data are loaded from a file corresponding to the mesh being prepared. A [min, max] pair is loaded for each texture, and stored in an array with the mesh – prior to rendering that mesh, those values will be uploaded to the GPU as a simple uniform. While this transfer is generally required every frame, the amount of data involved (two floating point values per texture) does not represent a significant amount of data, and causes no performance difficulties.

4.3.3 Incident Radiance Sample Points

Incident Radiance sample points are locations in the space surrounding a mesh where the Spherical Harmonic projection of *actual* incident lighting will be computed each frame. Recall from 3.1 that these are the second spherical representation (in addition to the surface receiving properties) needed to solve the Rendering Equation and find the shading at a given point on the surface. While a single incident lighting sample can adequately represent the lighting over a surface due to an infinitely distant source, more samples are needed to capture local variation in light over a mesh. More disperse samples near the mesh capture the differing angles between the mesh and the scene light sources in different regions of space, leading to more realistic variation. A single sample point can only capture one such angle, and thus does not adequately represent large meshes with relatively close lights.

As computing the SH projection of incident lighting at every surface point over every mesh in the scene would be impractical given even a modest number of vertices, a predetermined number of samples are established near each mesh to represent the actual illumination per frame over that surface. For simplicity, our system generally selects sample points at both extremes of a mesh's x, y, and z coordinates. While this sampling configuration is not ideal for all surfaces, it provides good results for many meshes which exhibit a generally convex, "balanced" structure.

4.3.4 Vertex Preparation

After locating lighting sample points near (or on) the surface of a mesh, each vertex is assigned a corresponding "contribution" from each lighting sample. These "contribution" factors are based on the minimal surface distance required to get from the vertex to the sample point. By inverting the ratio of the distance required to get to a sample (D_s) to the total distance to *all* samples (D_t), a variation scheme which maximizes the effect of nearby samples to vertices is created.

$$C_s = \frac{D_s}{D_t}$$

To normalize these contribution factors, each contribution set is summed up for a vertex, and each contribution is divided by the total of all contributions at that vertex. These contribution factors are then stored at the vertex, and passed as “attributes” of that point during rendering, in the same manner that the texture coordinates are passed to the GPU with the vertex’s position.

At run-time, these contribution factors are used to scale each lighting sample near the mesh, essentially interpolating the lighting representation over the surface to a particular point. This allows local lighting variation over the surface for light sources which have a perceivable angular difference between regions of the mesh – essentially lights which are sufficiently close to the mesh to strike the surface at different incident angles, depending on which region.

4.3.5 Light Sampling

Once the light samples have been placed and the per-vertex contributions of those samples has been calculated, the mesh is finally ready to be rendered. For each frame, it is positioned in the scene – either statically, or subject to an animation loop which continually repositions the mesh over time. Once the location is determined, the incident radiance can be sampled at the locations established earlier. For each sample, a coefficient vector is initialized corresponding to the number of bands being used in the current Spherical Harmonic representation. A single vector is then constructed from the sample point to each light and normalized to unit length. In a manner reminiscent of the construction of the Monte Carlo sampler in 3.3, the spherical coordinates (θ, ϕ) are found which correspond to the x, y, and z components of the new vector pointing towards the light.

Again akin to the Monte Carlo sampler developed earlier, the (θ, ϕ) coordinates are used to evaluate the corresponding Spherical Harmonic basis functions in the direction of the “light vector.” These coefficients are multiplied by π to represent an attenuated cosine lobe in the direction of the light. This operation is performed for each light in the scene, and the coefficient are summed up independently for each band and basis function to give the final Spherical light representation at that point in space, from all lights in the scene.

4.3.6 Lighting Coefficient Transfer

After calculating the lighting representations for each sample point near a particular mesh, the lighting coefficients need to be transferred to the GPU before they can be used to evaluate the surface shading along a fragment. While previous work passed a single lighting sample's coefficient vector as a uniform array (similar to how our work passes texture scaling factors)[15], this approach can become unwieldy when utilizing a larger number of lighting samples. Instead, we package our lighting samples into a small, temporary one-dimensional array texture. The texture is sized to be NxS units in size, where N is the number of Spherical Harmonic coefficients being used, and S is the number of lighting sample points being evaluated near the mesh. Similar to the assembly of the two dimensional array texture discussed previously, we allocate this texture once, and simply update sections of it each frame if the lighting may have changed at that sample point.

This method of transferring lighting sample data to the GPU is more computationally challenging than the more basic approach of passing a simple uniform array to the GPU per mesh, and may invoke small performance penalties as the GPU binds the texture into a texture unit. However, given the relatively small number of lighting samples being used in relation to the number of vertices and faces being rendered, the amortized cost of assembling and transferring the lighting samples once each frame has not become a performance bottleneck. As such, we consider the greater flexibility of our method in representing multiple lighting sample locations to the GPU, in contrast with a single sample, be worth the slight performance cost.

4.3.7 Rendering the Triangles

Vertex Data

Once the per-mesh data has been transferred (the lighting samples texture) and activated (the model coefficient texture array), each triangle of the mesh may be passed to the GPU for rendering. Transformations may be applied to the positions of the geometric primitives, which are faithfully carried out by a simple vertex shader. Additional data must be passed per vertex, however, to give the fragment program enough data to find the correct model and lighting coefficients.

For any vertex, the following data is required:

- Vertex Position
- Texture Coordinates
- The contribution of each light sample to that vertex

Notably absent as a requirement is the per-vertex *normal* which is usually a stable of shading calculations. Because of the effort put in during the precomputation step – particularly with respect to our work using models with more complex surface detail available – all of the usefulness of the normal has been "baked in" to the texture data, and is not required explicitly.

All vertex attributes are passed through to the fragment shader subject to interpolation over a triangle's face. The vertex position handles the additional task of transforming the position from object space to world space by utilizing the current Model-View-Projection matrix stack.

Fragment Shading

Once the vertex shader is done transforming the positions of the vertices, each triangle is rasterized to the screen and a fragment program is called to shade each pixel. This fragment shader is the element which brings together a point on the mesh's surface, the textured model coefficients, and the sampled lighting coefficients to create the color which is seen on the screen. To recap the origins of this data, texture coordinates and lighting contributions come from the vertex shader and are interpolated over a triangle's face, the lighting sample coefficients from the temporary one-dimensional lighting texture, and the model coefficients from the two-dimensional texture loaded for the mesh.

As described in 3.1, the color seen on the surface is the integral of the product of two spherical representations. And as elaborated in 3.4.3, the integral of the product of two functions projected into Spherical Harmonics is simply the sum of the product of their coefficient vectors. So, to bring these elements together, we need to look up the model coefficients from the two-dimensional texture defined over the surface of the mesh, and interpolate the lighting samples to get the lighting at our current point on the surface.

The first element, the model coefficients, are nearly trivial: we use the coefficient index to get a specific "depth" out of the two-dimensional texture array, and use the texture coordinates (u, v) to look up the value at that point on the mesh, subject to linear smoothing over the surrounding four texels. To "re-inflate" the resulting [0, 1] value acquired from the texture to the real range, we the [min, max] scaling values associated with that texture. Because the [0, 1] range can be thought of as a parametric variable ranging from the real [min, max] scale, we can use the standard library functions provided by most shading languages to interpolate between the minimum and maximum value.

$$Cg : C_m = lerp(min, max, compressed)$$

$$GLSL : C_m = mix(min, max, compressed)$$

These two functions quickly and easily linearly interpolate between the minimum and maximum values found in the texture before compression, and are generally implemented in hardware on modern GPUs.

The lighting coefficients are slightly more complicated, but not significantly so: each row of the lighting texture is processed as a separate lighting sample, and all the rows are weighted by the "contribution" scaling factor associated with that vertex for that lighting sample. To find the coefficient at a given index, a *column* of coefficients is examined – all these coefficients are of the same Spherical Harmonic basis function, just of each different sample location. The sum of these coefficients, each scaled by that light's contribution on the point in question, and normalized by the total contribution added, is used as the interpolated light value from all samples at that point on the mesh's surface, as:

$$L_n = \frac{\sum_{i=0}^{samples} contrib[i] \times tex(n, i)}{\sum_{i=0}^{samples} contrib[i]}$$

Now that we can acquire all the model and lighting coefficients necessary at a point, the only step left is to combine them: starting with a color initialized to black, we loop from zero to the number of Spherical Harmonic coefficients being used, take the product of the model coefficient and the lighting coefficient at that index, and add that to the color. After all the coefficients have been processed, we write that color, along with a full opacity alpha channel, to the framebuffer:

$$Color = \sum_{i=0}^{l^2} model_i \times light_i$$

4.3.8 Rendering Addendums

While the above description of the shading procedure is complete in all essential parts, there are two other elements of concern which have some influence on the final shading calculation.

Material Properties

While the diffuse color of a material may be included and "baked in" to the precomputation step, we have found it much more useful to calculate effects of potential lighting on the initial surface in the absence of the specific surface color, and apply this color later by multiplying it by the summation produced from model and lighting coefficients. While this does not capture the effects of some more interesting BRDFs, we have found it very useful for utilizing a single pre-processed mesh multiple times with arbitrary run-time coloration. If more complex BRDFs need to be represented, those properties should be "baked in" to the texture during precomputation as described earlier.

Gibbs' Phenomenon

A common signal processing problem known as "Gibbs' Phenomenon" occurs when functions with discontinuities (like our synthetic lights) are projected into discrete representations meant to represent continuous functions, like Spherical Harmonics[14]. As the number of bands increases, the resulting representation error can cause artifacts around areas of discontinuities, such as when the bright lights go to sudden black on the back surfaces of objects. In practice for Computer Graphics applications, this causes "Ringing" artifacts, in which areas on the shadowed side of a bright object exhibit rings of light. These artifacts are a large problem because they are often obvious – the human eye is good at spotting discontinuities, particularly when it comes to objects which should be in total shadow.

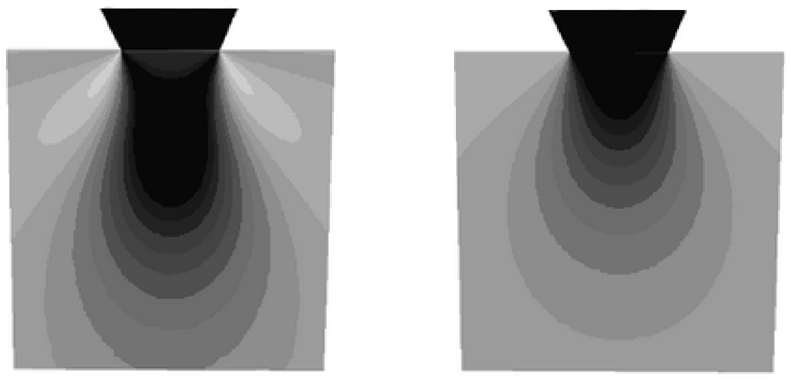


Figure 4.3: Unwindowed and windowed integration, showing Gibbs' Phenomenon

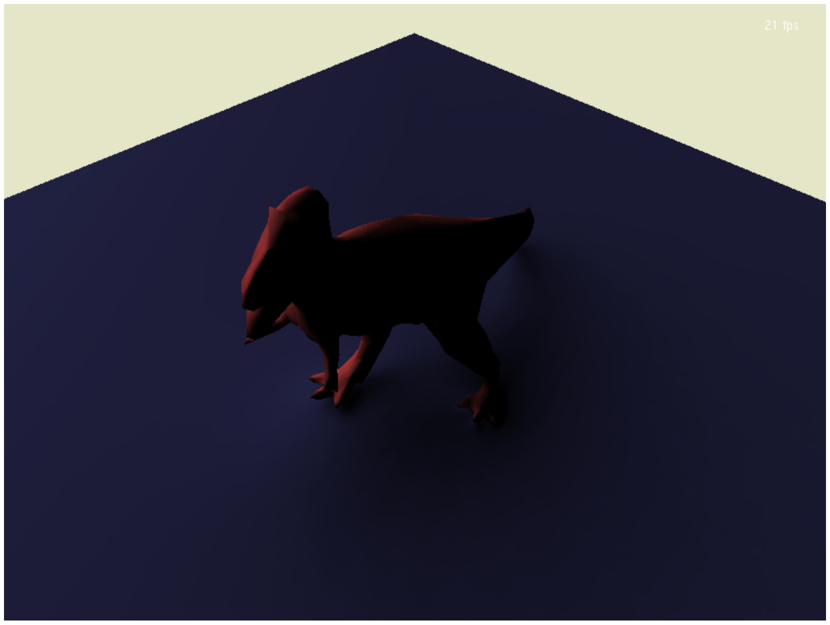
To combat Gibbs' Phenomenon, we utilize a technique called *windowing* which progressively reduces the impact of higher order Spherical Harmonic basis functions on the final results. In our work, we follow Sloan et al in utilizing a traditional signal processing window – the"Hann" window, or "Hanning Function." This window progressively reduces the weight given to higher order coefficients using a cosine curve, normalized to the number of total bands used in the representation. By incorporating this into the summation loop in the fragment shader, most of the distinctly effected of parts of the surface are calmed to a much more appropriate color.

4.4 Textured PRT Review

In this chapter, we have presented our techniques for generating and rendering densely sampled Precomputed Radiance Transfer data over the surface of a mesh. We have highlighted our specific contributions in the areas of:

1. Texture storage normalization, via range compression
2. Complex surface detail utilization, with normal mapped surface data
3. Textured transfer of lighting samples

These factors increase the utility of Precomputed Radiance Transfer techniques for real-time applications by decreasing the performance hardships though lower texture sizes (1), improving visual reconstruction by allowing highly detailed PRT transfer data over the surfaces of low polygon meshes (2) and permitting local light variation over mesh surfaces when using GPU textured rendering methods (3).



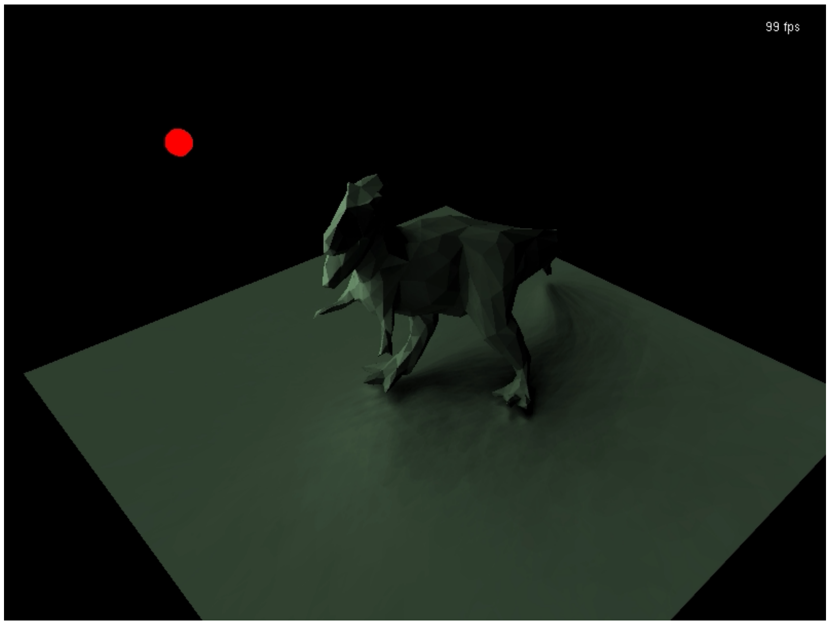


Figure 4.4: Tyra rendered with a ground plane, receiving shadows

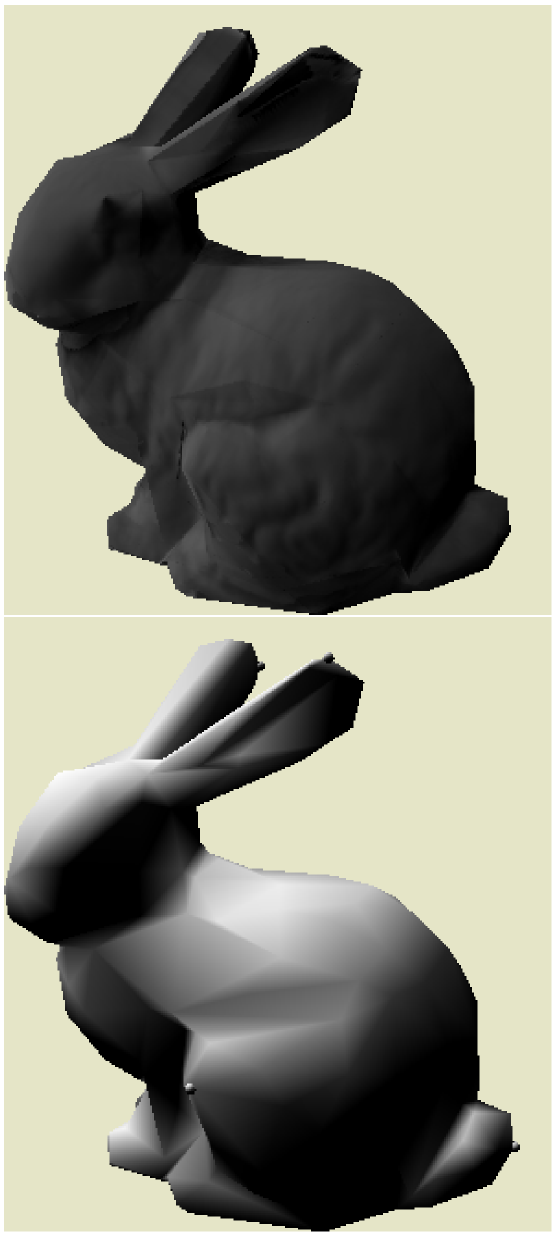


Figure 4.5: 500 face bunny rendered with PRT textures compared with standard smooth shading

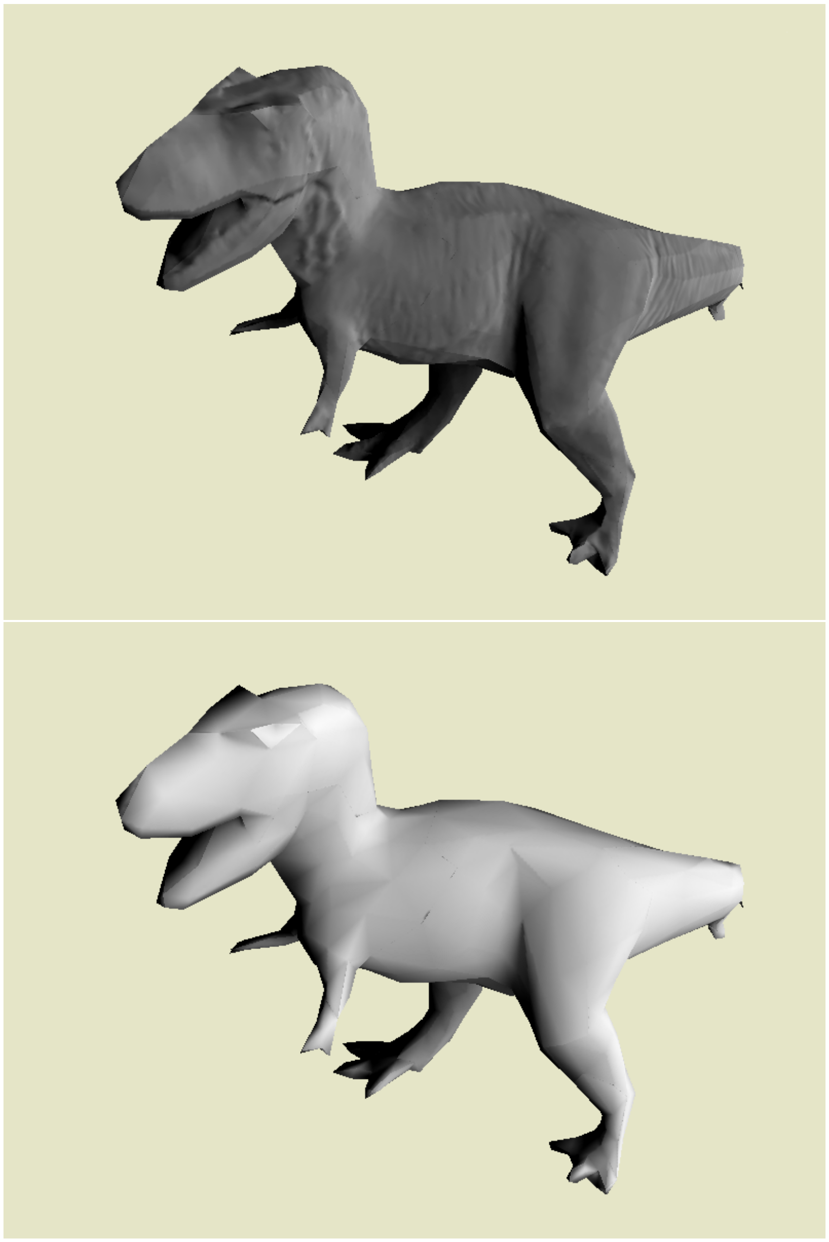


Figure 4.6: 1000 face Tyrannosaurus rendered with PRT textures with and without baked in normal maps

Chapter 5

Hierarchical PRT

5.1 Introduction

Having established a basis for rendering advanced effects – namely shadows – over the surfaces of simple geometry using textured mapped, densely sampled data, we turn our attention to migrating the local, self-shadows seen over such surfaces to shadows cast between objects. As described previously in Section 2.2.2, this process is often called *neighborhood transfer*, because it attempts to transfer radiance information between spatially related objects. To once more ground our contributions in an understanding of how Hierarchical PRT works, we will first provide a general overview of the algorithm.

5.1.1 Hierarchical PRT Overview

After loading the necessary run-time components of the previous Textured PRT system, we establish an "ordering" of the scene which divides it into pieces. First, we attempt identify scene elements which exhibit spatial coherence, and are likely candidates for further clustering. For each of these groups, a "top level" scene element is generated which will form the root node of a tree of scene elements. Once these general areas are established, we group geometry inside these nodes hierarchically on the basis that objects which are close to each other may be treated as a uniform object by more distant geometry. After these trees are established, for each frame, we attempt to find the "filtering" effect that each node has on its neighbors as light passes through the tree. Using these filter representations, the actual incident light at mesh sample points is masked off in directions corresponding to cast shadows. The resulting, shadowed incident lighting is used to light that surface.

Again, in review:

1. Load Textured PRT data normally
2. Identify "top level" node areas
3. Build "scene node" trees using the top level nodes as roots
4. Find the "filtering" effect that scene nodes at a particular tree level have on their neighbors
5. Mask off elements of the incident lighting found at a mesh's sample points using "filters"
6. Pass the incident lighting and render as in normal Textured PRT

5.1.2 Contributions

Rather than attempting to precompute models which represent the way that objects in a scene cast shadows onto their neighbors, our work develops the concept of a hierarchy of scene nodes which limits the extent of visibility testing. While the idea of a “scene graph” or spatial subdivision scheme is well established in Real-time Graphics[1], we believe that its applicability to shadowing PRT representations is a new way of harnessing tree-like scene structure.

With that in mind, we contribute the use of a tree-like hierarchy of objects and bounding volumes, the restriction of shadowing checks for establishing “filter” functions, and the application of windowing techniques to create a stable “masking” function, to the area of Precomputed Radiance Transfer.

5.1.3 Hierarchy Establishment

To begin our system, we require as input a set of meshes which have been preprocessed with Textured PRT techniques, a number of Spherical Harmonic bands to establish representation fineness, and a Monte Carlo sampler to project functions into Spherical Harmonics. As the rational and techniques underlying these elements have been presented elsewhere in this document, we will refrain from expounding upon them again here.

Once the meshes of a scene have been read in, the scene needs to be searched for spatial groupings between elements which are relatively close together. The rational for this decision is shadowing effects, particularly those generated from light sources that are relatively far away, are much more likely to result from an interaction between two elements close to either other than those which are far apart. Using this assumption, we create “top level” nodes for each grouping, which will form the basis of a tree at that location. The “top level” node forms the root of the new "shadowing hierarchy," and is used to cast shadows down though its branches.

The problem of locating which nodes are close to each other may be solved using methods from clustering or computational geometry. In our current implementation, however, the application programmer loading the models specifies the relationships between scene nodes, keeping in mind location, size, and expected animation motion. Updating the system to automate this step and include run-time checks for hierarchy elements which have become degenerate is an important area of future work. However, because these updates just attempt to keep the tree coherent, we see them as somewhat orthogonal to the original problem of passing shadowing data through the tree itself and do not attempt to solve it more completely here.

5.1.4 Node Types

In the Hierarchy, nodes in the tree may be of two types: a "Mesh Node" or a "Composite Node." As named, *mesh nodes* are scene elements which represent actual geometry. These leaf nodes possess transfer coefficient textures, geometry, lighting sample points, filtering coefficients, and boundary extents. *Composite nodes*, on the other hand, possess only lighting sample points, filter coefficients, children, and bounding extents. To arrange the tree-like structure, composite nodes may have as children either mesh nodes or more composite nodes. Further, both node types may exist at different levels of the tree simultaneously, permitting shadows cast from both composite nodes and mesh nodes to affect each other directly.

5.1.5 Establishing Node Filtering

To first establish a straight forward intuition about how the "filtering" of scene nodes should work, the reader might consider a natural example of lighting the floor of a dense rainforest. While the observer on the ground is clearly sheltered from the brilliant equatorial sun, no one tree removes all the light which would be seen in the trees absence, nor does any one tree remove light from all the surrounding directions it could reach the observer from. Rather, it is the interlocking combination of *all* the arboreal occlusion surrounding the observer which "filters" out light from different angles. Reverting back to Computer Science, we could say that we "mask off" certain representative areas of the lighting representation where we want to remove the values.

To shadow objects progressively down the node hierarchy, we perform an operation similar to the progressive assembly of a bitmask – we start with a completely "on" representation, which removes nothing when "and-ed" with a lighting representation, and then zero out components corresponding to regions which should receive no light from the world due to occlusions. When this mask is combined with the actual incident light computed at one of the sample points discussed Section 4.3.3, the mask "lets through" only lighting directions which are not shadowed.

To migrate this approach to the Spherical Harmonic representation we use for spherical functions, of which shadowing is now one, we first must construct a filter which has "all bits on," and filters out no light. Using our set of Monte Carlo sampling directions, we simply add up all the contributions present into a Spherical Harmonic coefficient vector. This is in contrast to its usage before, where we cast rays out along the sample directions to determine occlusions. While that approach will be used later, the initial unshadowed representation is of simple unit intensity over the entire sphere.

5.1.6 Moving Filtering Down the Tree

Once the unit basis for the "filter" is established, we traverse our way down the tree. At the next level down, we attempt to find out how much light is "removed" from the unit representation. We establish this by performing roughly the opposite of the calculation done earlier in preprocessing PRT meshes – for each object, we cast rays out along the set of Monte Carlo sampling directions. If a collision with a scene node at the same level of the tree is detected, we subtract that direction's corresponding Spherical Harmonic coefficients from the parent's filter, and store the resulting coefficient vector with that node in the tree. Progressively, these filters represent how much total shadow there is affecting a particular node and all of its children for a static scene.

Obviously, the tree-walking filtering goes beyond the first level in the tree. After all filters are computed among a node's children, the same procedure is invoked on all sub-nodes of those elements. This recursive darkening of the filtering function continues, as more and more shadowing is accumulated, until the process hits an actual mesh. As mesh nodes can only be leaves in the tree, the process stops.

This method has been utilized to shadow trees of relatively shallow depth successfully without performance penalties relative to the depth of the tree. Concern does exist over the stability of the Spherical Harmonic representation in excessively deep trees – extremely high values in some small areas may cause more artifacting in reconstruction than windowing techniques can reliably combat. The problems associated with sharp jumps in the projected representations is discussed in Section 4.3.8.

5.1.7 Combining Filtering with Lighting

After all tree-walking has been performed to establish the spherical shadowing function at each node in the tree, the actual incident lighting is calculated at each mesh node's sample points. Before these spherical lighting samples are transferred to the GPU and used to light the mesh as described in Section 4.3.6, the portions which are in shadow must be removed. We took a number of different approaches to subtracting lighting from the initial incident representation, and the ways in which they fail may be informative to the reader in understanding why we settled on the "product masking" operation which we contribute.

Subtraction

An obvious approach to removing lighting from the representation is to simply subtract the total filtering representation from the total lighting representation. This approach actually gives relatively smooth, stable results in simple cases, but tends to generate very unbelievable shadows. Because the filter representation is based in Spherical Harmonics, the areas which tend to get "blocked out" by other geometry tends to become nicely circular. When one of these representations is subtracted from another, the result is a large, black circle over the receiving object. Again, while this may sound like a desirable trait, the shadow tends to appear far in advance of the "casting" object, without any softness or smoothed edge. The *penumbra* of a casting object tends to remove even bright light, before the shadow casting object has actually interposed itself between the light and the shadowed recipient.

Spherical Harmonic Products

As an alternative to direct subtraction, another approach we took in developing the filtering procedure was to multiply the incident lighting sample coefficients with the filtering function. The filtering function, which only has darkened "low value" areas which should be in shadow, remains at a relatively unit scale over the rest of the sphere. Mathematical intuition, then, suggests that for non-shadowed areas the product of the two representations will be the analog of "multiplying by 1," preserving the original value. Concordantly, intuition suggests that the reverse should also be true – "multiplying by zero," will erase any value which was there.

Unfortunately, this approach fails as well. Spherical Harmonics, do not necessarily have "zeroed" coefficients in areas where they are black. Instead, the integration procedure explained earlier utilizes many different bands of coefficients to influence the final value at a particular point on the spherical representation. While some of areas may genuinely be zero or "one," much of the time values close to those are the result of a complex interaction between Spherical Harmonic bands. While this relationship is just fine for *integration* product utilization of the type discussed earlier, computing the product of the filtering function and the lighting function tends to "turn on" errant parts of the lighting representation on the back of the object. Regions which were black due to a complex balance of bands become lit when subjected to a representation which is generally quite bright, absent a few locations: the filtering function.

Additionally, the Spherical Harmonic product has more severe difficulties when more occluders are added to the tree; the combinations of relatively high frequency, discontinuous shadows on different areas of the sphere at the same time exacerbates the mutation of formerly dark parts of the incident lighting representation, leading to a highly "unstable" lighting function which tends to warp dramatically whenever something in the scene moves.

Spherical Harmonic Product Masking

Finally, we have the approach we developed to solve the problem of instability in the Spherical Harmonic products. Realizing that comparatively bright regions of the shadowing function were strongly effecting otherwise undetectable items in the lighting function, we employ a smoothing tool from earlier to lessen the impact of these bands on the final lighting coefficients which are generated – the Hanning function.

Ordinarily, the Spherical Harmonic product of functions f and g, when computed in the frequency (coefficient) domain, is generated using:

$$C_k = \sum_{i,j}^{l^2} \Gamma_{ijk} f_i g_j$$

to find the coefficient C_k, for each kth coefficient desired (Γ_{ijk} is a triple product tensor of three basis functions, and is constant). Observing this, we can see that the ith and jth coefficients of the respective functions will have just as much impact on the low band coefficients of the resulting product as the existing low band coefficients. As these low bands represent the most dramatic, course changes in the value of the function, the product rule in the frequency domain gives high bands potentially disproportionate power over the final representation, particularly if those high i and j coefficients represent dramatic changes in the function – like sudden, discontinuous, highly negative occlusion data.

By applying the Hanning function to this summation process, we can convert the equation to the much more stable form of:

$$C_k = \sum_{i,j}^{l^2} \Gamma_{ijk} \omega(i,l) f_i \omega(j,l) g_j$$

where $\omega(i,l)$ is the Hanning function evaluated at coefficient index i, over possible total bands l. This formulation is used to combine the filter representation and lighting representation in a stable manner. It should be noted, however, that this stabilization comes at a cost: it is more difficult to preserve some high frequency data over multiplication, because of the lower overall impact of the higher band Spherical Harmonic basis functions.

5.2 Transfer

Once the filter has been successfully applied to the incident lighting representation, the result can be encoded into the temporary textures presented in the previous section in exactly the same manner. Similarly, after the mask is applied, all the rendering functions for the textured data function exactly as they did before.

5.3 Filter Caching

In some cases, it is possible that some filtering data can be preserved across frames. This is a desirable property, as the ray casting procedure used to find if a tree node is occluded by its neighbors can be an expensive operation – even when placed in the hierarchy which dramatically limits the number of nodes in the scene is has to be checked against. While the idea of caching some scene data to avoid repeated calculations is certainly not new, we mention this approach's amenability to it as a merit of the method.

In order to provide simple caching services, the run-time system includes in the representations of composite and mesh nodes additional data about their "dirtiness." A mesh node propagates dirtiness when it is translated in the scene. Upon translation, the mesh node informs its parent about the move, and then waits to be recalculated. If the mesh occurs on a level in a composite node with any siblings, they will also have to be recalculated (as the translation may have affected their occlusion). Objects further in the tree, however, may not have to update their occlusion information. If the objects contained low in the tree are treated as a simple "point mass," or bounding volume, and the move does not cause that bounding volume to be modified, then the local variation will not cause any filtering changes further up the tree. This property preserves the filtering representations already established higher up, and allows them to be refined only at the smallest level necessary, rather than throughout the entire tree.

Though the assumption that sub nodes can be treated as simple bounding volumes may lead to some inaccuracies, it can also lead to magnificent speed increases. We leave it for other developers to decide if it is more important to have very precise occlusions contributed from sub nodes, or to have fast, "blob" like shadows in such cases – the hierarchical method presented here can support either.

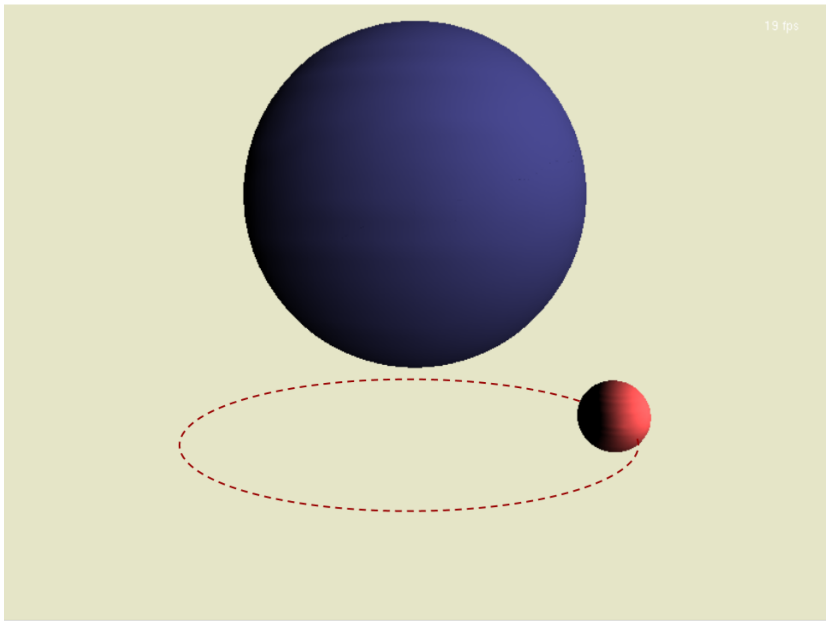
5.4 Hierarchical PRT Review

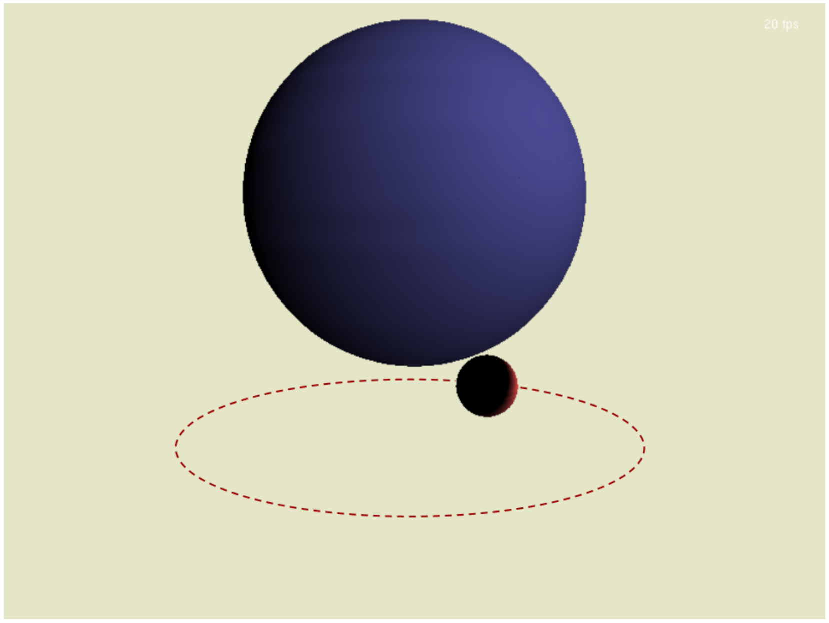
In this chapter, we have presented our techniques for representing spatially coherent elements of a scene with a hierarchical model. Specifically, our contributions are in utilizing such a hierarchy to:

1. Progressively refine “filters” representing shadow data down the hierarchy, using the Spherical Harmonic basis

2. Generate stable products of the resulting shadow representations and the sampled incident lighting at particular points in the tree

These factors increase the utility of Precomputed Radiance Transfer by providing a simple model for transferring shadow information across elements of a scene. Though the results are imperfect in some cases due to the trade-offs required to eliminate low “noise” in the combination of light and shadow, we believe it is a valuable step for assembling more spatially coherent methods utilizing PRT rendering.


19 fps


20 fps

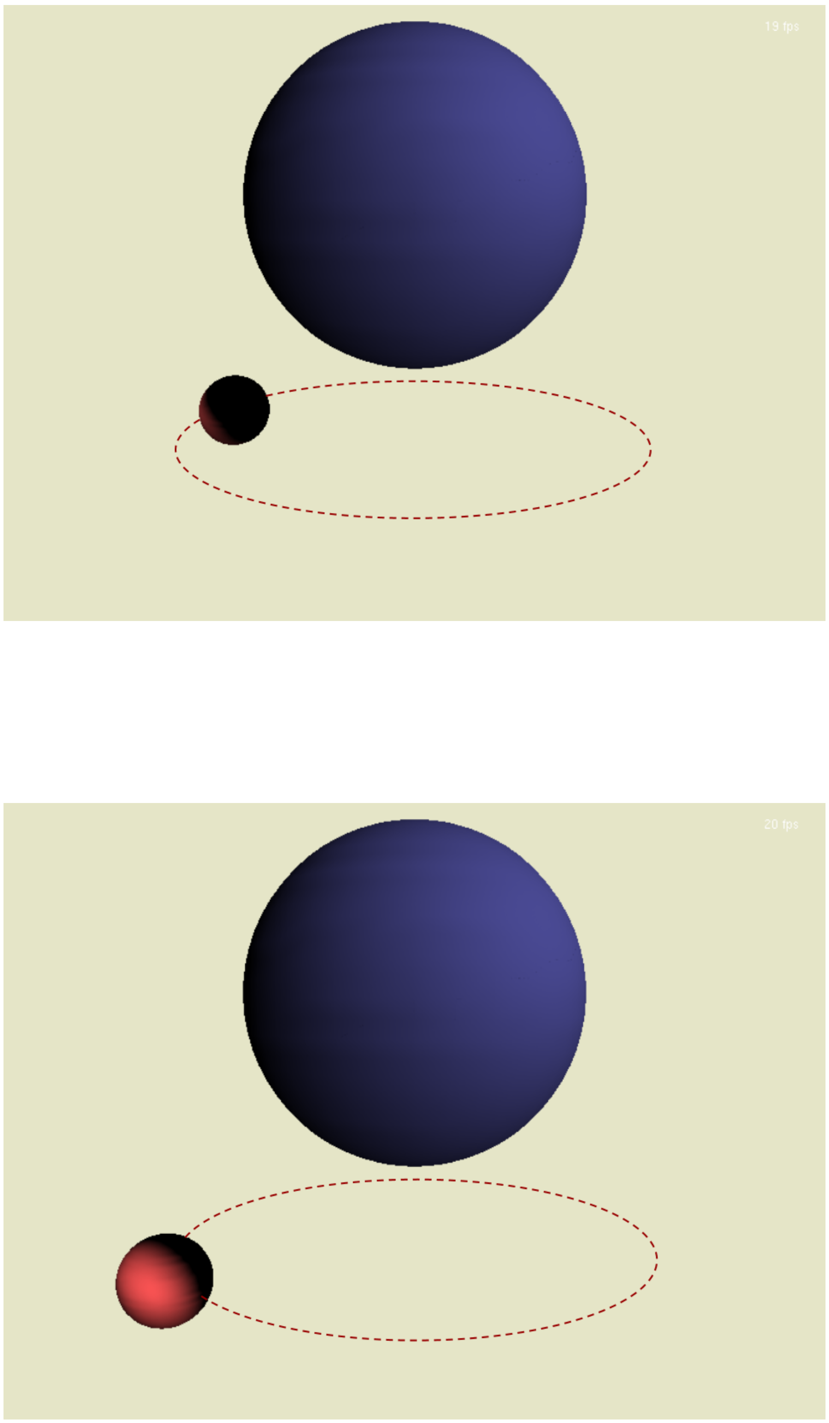
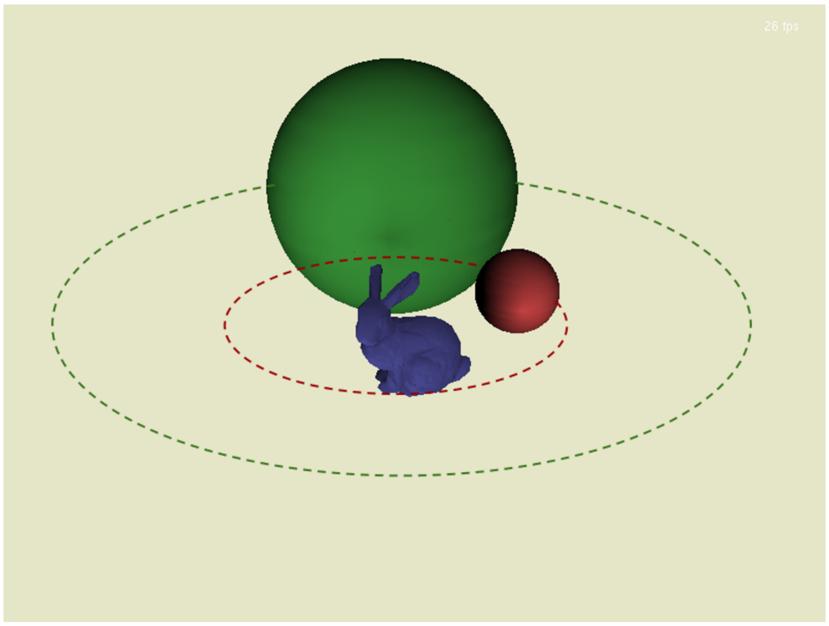
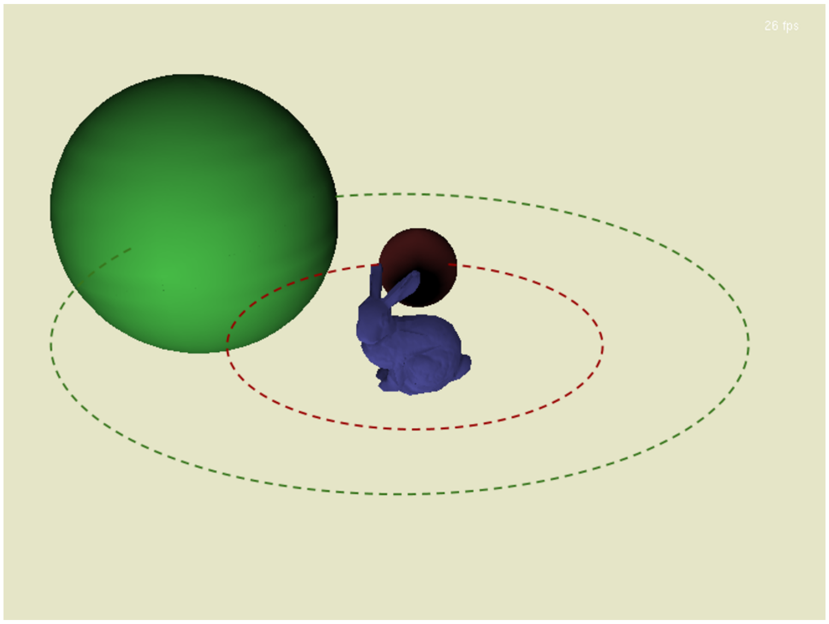
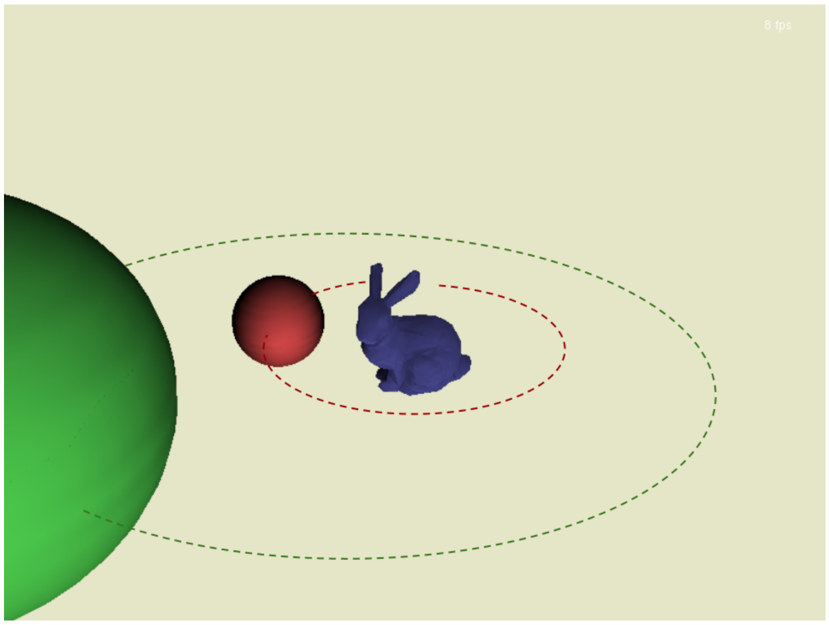
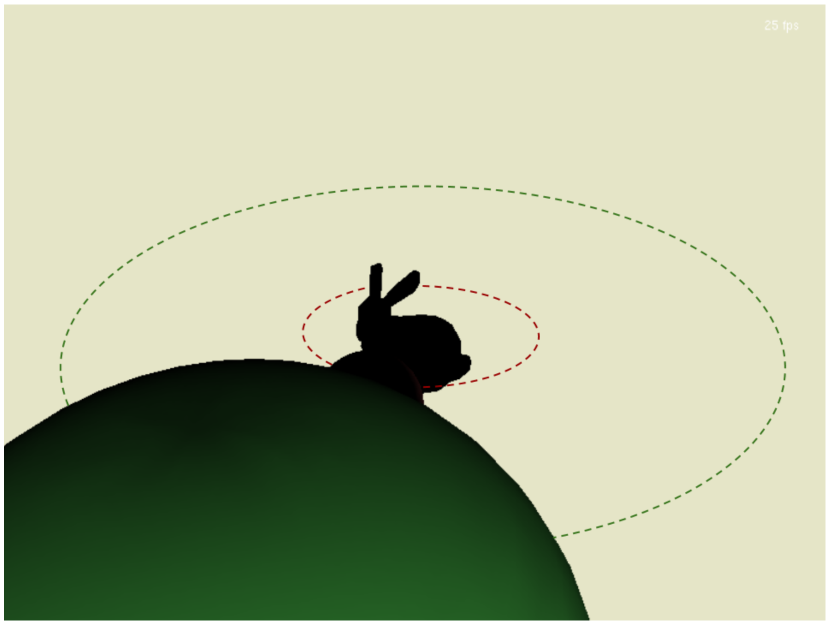


Figure 5.1: 2 Sphere Hierarchy, calculating dynamic shadows between objects


26 fps


26 fps


8 fps


25 fps

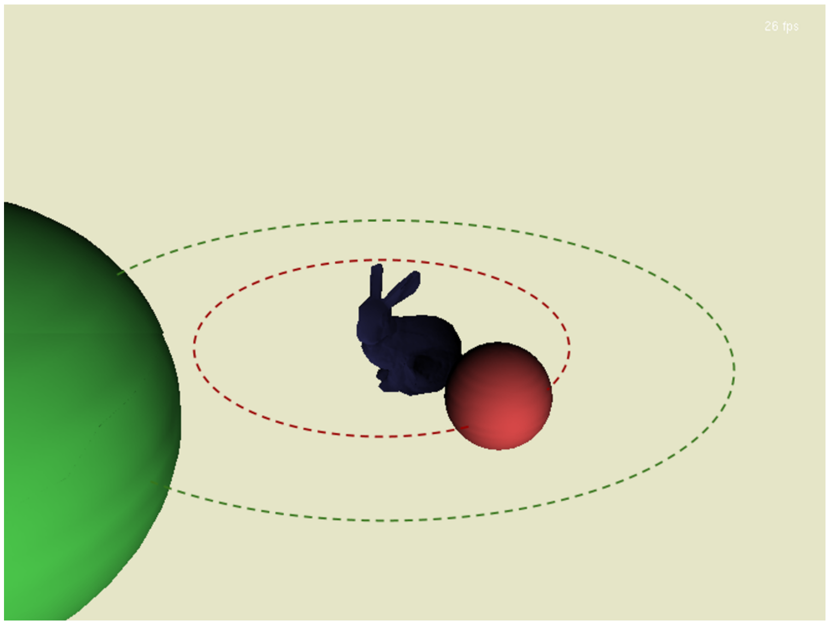


Figure 5.2: Bunny Hierarchy, calculating dynamic shadows between objects

Chapter 6

Results and Discussion

In this chapter, we will discuss the results which our implementations of the Textured PRT and Hierarchical PRT generate. In light of the properties of the Real-time domain, we will generally concentrate our analysis over three areas:

- Speed
- Storage Requirements
- Image Quality

Speed In our discussions of *speed*, we will focus primarily on nominal run-time speed, from the perspective of interactive, Real-time Graphics systems. While the thresholds for interactivity may vary with content being displayed, a minimum of fifteen frames per second (fps) is recognized from early literature[1]. More conventionally, however, a frame rate of sixty hertz (sixty frames per second) is the targeted rate of most Real-time Graphics applications[13]. We analyze the efficacy and scalability of our contributions with respect to this primary goal.

Storage While speed is the overriding performance concern in Real-time Graphics, the data sizes required to represent some elements may have a significant, though indirect, impact on both the frame rate and system scalability with respect to video memory, hard disk space, and distribution bandwidth. We analyze the storage issues associated with our contributions, attempting to separate performance difficulties due to storage from those due to other run-time factors.

Image Quality Finally, the quality of the final images produced is the key to a Computer Graphics system. While it is difficult to form objective criteria to quantify the quality of images, we use different rendering techniques as baselines to compare the resulting images subjectively using the following criteria:

- Perceived Surface Detail
- Fine Surface Shadowing Quality
- Inter-Object Shadow Quality

6.1 Testing Environment

Testing for our techniques was primarily performed on an Intel Core 2 E6420 running at 2.13 Ghz, with 2 GBs of system memory. The GPU utilized was an NVIDIA 8800 GTS with 320 MBs on-board memory and 96 stream processing units. To interact with the GPU, we use OpenGL 3.2 with GLSL 1.5.

We have also confirmed that these techniques run reasonably on lower-spec graphics hardware, such as a set of NVIDIA 8400 GSs (down to a mere 16 stream processors), but we do not present experimental data from these configurations.

6.2 Textured PRT

6.2.1 Comparison Algorithms

To analyze the efficiency of the presented Textured PRT algorithm, we decompose the run-time complexity of four different rendering techniques:

1. Phong Shading
2. Phong Shading with Normal Map
3. Per Vertex PRT
4. Textured PRT

Using these different cases we illustrate the speed, size, and quality trade-offs which our Textured PRT technique achieves relative to other, move conventional procedures.

To normalize the performance of these algorithms, we implement each using a vertex and fragment shader pair implemented in GLSL. This eliminates potential performance differences by creating a more uniform pipeline for testing content to flow though.

6.2.2 Test Case

To provide a simple test case which highlights lighting variations and occlusions over a surface, we use a decimated, five hundred face version of the Stanford Bunny. We draw the high-quality normal data required for normal mapping and empowered Textured PRT from the full resolution, sixty nine thousand face reconstruction of the Bunny. For the PRT examples, we use five Spherical Harmonic bands, for a total of twenty five coefficients. We preprocess such meshes using a ten thousand element sampler as described in 3.3. For textured data, we use 512×512 resolution images.

6.2.3 Speed

Technique	Frame Rate	GPU Operations	CPU Operations
Phong Shading	100	19	0
Normal Mapping	100	24	0
Vertex PRT	99	1	75
Textured PRT	66	23	0

Table 6.1: Speed statistics for Textured PRT comparison

As shown in Table 6.2.3, the utilization of Textured PRT in the fragment shader has an impact on the performance of the application, but still succeeds in maintaining interactive rates. Because of the similarity in operation count between the normal mapping shader and the Textured PRT shader, we believe that the performance penalty is due the streaming of our multisampled texture to the GPU, rather than the plurality to texture samples being used. Further anecdotal evidence has suggested that single sampling, such as that done in our Vertex PRT test, returns the frame rate back into the 90 fps region as well.

6.2.4 Storage

As shown in Table 6.2.4, these storage metrics reflect the amount of data required to be passed to the graphics hardware to render a mesh using the specified

Technique	**Per Mesh Data Size**	**Per Vertex Data Size**
Phong Shading	0	24
Normal Mapping	786 kb	56
Vertex PRT	0	24 (300)
Textured PRT	19.6 Mb	44

Table 6.2: Storage statistics for Textured PRT comparison

technique. The “Per Mesh” data size reflects quantities like texture data or lighting samples which must be utilized on a per-mesh basis. “Per Vertex” data, on the other hand, reflects the amount data which must be passed for each vertex of the mesh.

As shown, Textured PRT takes a large amount of memory on the GPU: we project that our testing hardware could only hold sixteen unique representations at a time without having to begin swapping texture data back to main memory. While GPU memory sizes are consistently increasing – the modern generation of GPUs have upwards of one gigabyte [11]– this level of memory consumption represents a hurdle for Textured PRT usability.

As a result of these findings, we have investigated the compressibility of our coefficient textures on the graphics hardware. Not enough research has been performed to present definitive conclusions, but anecdotal evidence suggests that a roughly 4:1 compression rate may be achievable without performance degradation through the use of the “S3TC” texturing extension to the OpenGL standard[4]. Unfortunately, some artifacting can occur around areas of high signal change when the texture is rendered – we leave this optimization of the algorithm as an element of future work.

Another interesting result shown by the above data is the comparatively large volume of data which must be transferred per vertex when using normal mapping. Intuitively, one might assume that Textured PRT would require yet more data to supply the "contributions" of each lighting sample, for each vertex. As our system uses six lighting samples per object, this translates into an additional twenty four bytes per vertex to interpolate the lighting representation alone. Surprisingly, the normal, bi-normal, and tangent vector data required to rotate scene lights into the "tangent space" utilized by normal mapping requires even more per vertex.

Unsurprisingly in the above results, the two interpolated methods – Phong shading and Vertex PRT – require the least data to be sent to the graphics hardware. In the former case, the interpolated position and normal data are simply used to evaluate the Phong lighting equation, while the latter does an even more straight forward interpolation of the position and color data sent per vertex over the entire triangle.

The low data volume for per-vertex PRT here is deceptive, however – in our implementation, the color calculated for each vertex is the result of a 25 component dot product performed before the vertex is passed to the graphics hardware. If the coefficients used to calculated the color passed to the graphics card are accounted for in our analysis, then the total jumps from twenty four bytes to three hundred bytes per vertex.

Another important consideration which reduces the effectiveness of per-vertex PRT rendering compared with Textured PRT rendering is the preclusion of storing the geometry on the graphics card. While we have not taken that step in this work, a standard technique in real-time rendering to utilize the GPU's memory is to set up "display lists," or "Vertex Buffer Objects"[7]. These constructs transfer geometry and per-vertex attribute data from the application's memory to the GPU once, and then simply ask the GPU to use that geometry over and over again. Because the color being interpolated between vertices in Vertex PRT rendering is established per-frame on the CPU side – and is dependent on a potentially changing light – it must be recalculated, resent, and re-interpolated every time the light or geometry moves.

Textured PRT circumvents this limitation by using a storage and transfer mechanism which, while large, defers the actual colorization (the dot product of spherical harmonic coefficients) until the fragment shading stage. This means that transforms can be applied to geometry, and the lighting changed, per-frame, without requiring the retransmission of *all* vertex data.

6.2.5 Quality

In terms of subjective quality, we've found the results of our Textured PRT technique to be the most satisfying of those in this set of methods. The following matrix describes the virtues and issues with each of the rendering methods, in terms of visual appearance.

Because of the nice properties which Textured PRT exhibits, in terms of surface effects like self-shadowing and normal reconstruction, we believe that the resulting image quality stands out from the other representations.

Technique	Virtues	Issues
Phong Shading	Nice per-pixel variation of lighting across faces	Does not add additional surface detail, leaving some areas feeling very faceted on simple meshes
Normal Mapping	Good approximation of complex surface geometry	Does not capture self-shadowing information over the model
Vertex PRT	Captures self-shadowing detail, interpolated over the mesh	Does not evaluate per-pixel, leading to the smoothing of discontinuities on simple meshes
Textured PRT	Recaptures surface detail and provides self-shadowing effects	Edges of lighting representation occasionally seem to extend a little far

Table 6.3: Textured PRT quality comparison matrix

6.3 Hierarchical PRT

We will now turn our attention to the results produced by our Hierarchical PRT lighting system. As this work utilizes much of the Textured PRT technique as a basis, we will avoid reproducing the more holistic analysis covered in the previous section, and focus on the additional constraints which utilizing Hierarchical PRT places on the system.

6.3.1 Test Cases

Test A Our most simple test case of using the Hierarchical PRT system to cast shadows from one node to another is using a one level tree with two leaf nodes. One mesh, a sphere, is located above $(+y)$ the origin, while another mesh rotates around it in the X-Z plane. A light cast from $< 8, 8, 8 >$ causes the moving object to occasionally pass into shadow behind the large sphere. We have used this test cast to confirm that shadows are cast from one node to another along an equal level of a single tree.

Test B A more advanced test case uses concentric rings of meshes in a "inverted planetary" configuration. A small mesh is placed in the center of the group, at the origin. A slightly larger mesh orbits around it. These two meshes are arranged into a "composite node" as presented in Section 5.1.4. This composite node is in turn orbited by a third mesh. The existing composite node and the third mesh are assembled into a second composite node, which forms the root of the tree. This test case demonstrates the transmission of filtering data down branches of the tree, rather than merely over a single level. To utilize the "orbital" model, the light is placed along the positive Z axis, and the meshes are animated to rotate about the origin at different speeds.

6.3.2 Speed

Test Case	625 samples	2500 samples	10,000 samples
Test A	50	26	8
Test B	57	38	15

Table 6.4: Hierarchical PRT frame rate variance

An analysis comparing of performance results from Table 6.3.2 yields some interesting results. Increased sampling density has the expected effect of slowing the application due to an increased number of ray casts being performed, but the variation between Test A and Test B is intriguing, given that Test B has more objects and more complicated animation patterns. We believe that this artifact is caused by the distances between the objects in each test case; the two meshes in Test A are closer together, and therefore they are more likely to generate potential collision hits. As our collision detection scheme for ray casts passes first through a bounding volume check before assessing contact with an actual face on a mesh's surface, we believe that the performance difference is explained by this "early culling" of rays happening more often in the Test B case than in the Test A case.

Another interesting note about the performance of Test B is the relative independence of the hierarchy's performance from the number of *total* nodes in the tree. Instead, as expected, the tree maintains similar performance at each step proportional to the number of nodes at a particular level.

Another important criteria in examining the performance of the hierarchy is the plurality of sampling locations utilized by the Textured PRT system. In Textured PRT, the addition of increased number of sample points does not pose a significant performance impediment, due to the simplicity of sampling incident radiance at a point in space. In contrast, finding the "filtering" representation for each point is much more time intensive, due to the ray casting procedure required to determine occlusion information. Removing multisampling from filtering calculations significantly increases performance, but forces shadowing to be "all or nothing" over an object. This has the no adverse effects for large objects occluding small ones, but precludes correct results for small objects casting small shadows on to large receivers.

6.3.3 Storage

The storage of the Hierarchical PRT system is very comparable to that of the Textured PRT system; While some additional data is necessary to maintain the tree representation, this is a very small amount of data relative to even a single simple mesh in the system – much less the texture data being used.

6.3.4 Quality

The quality and type of shadows which can be successfully cast using a Hierarchical PRT representation leaves something to be desired. We have had decent initial results casting shadows from large occluding objects on to small receivers, as well as in progressively darkening the filtering representation used to shadow leaf nodes. Unfortunately, we found that the limitations of Spherical Harmonic representability and the aggressiveness necessary in our stabilization function for the masking procedure causes the creation of small, crisp shadows to be very difficult.

As a basis for comparison on shadow quality, we will use the same test cases rendered with stencil Shadow Volumes instead. Shadow Volumes are a screen space, stencil buffer technique which detects shadowed areas of the framebuffer by extending the "silhouette" edges of objects when viewed from the light's location [10].

While the shadows generated using our Hierarchical PRT technique are reasonable, and softer, in the case where large occluding geometry is moving between the light and a smaller receiving object, our technique does not generate nearly as convincing results for small occluders. We believe these poor results occur for two reasons.

First, course incident lighting sampling, even with interpolation, cannot faithfully reproduce the shadowing effects granted using densely sampled representations. This becomes more apparent if we consider that in the initial Textured PRT calculation, we calculate occlusion information at *every texel.* This density allows

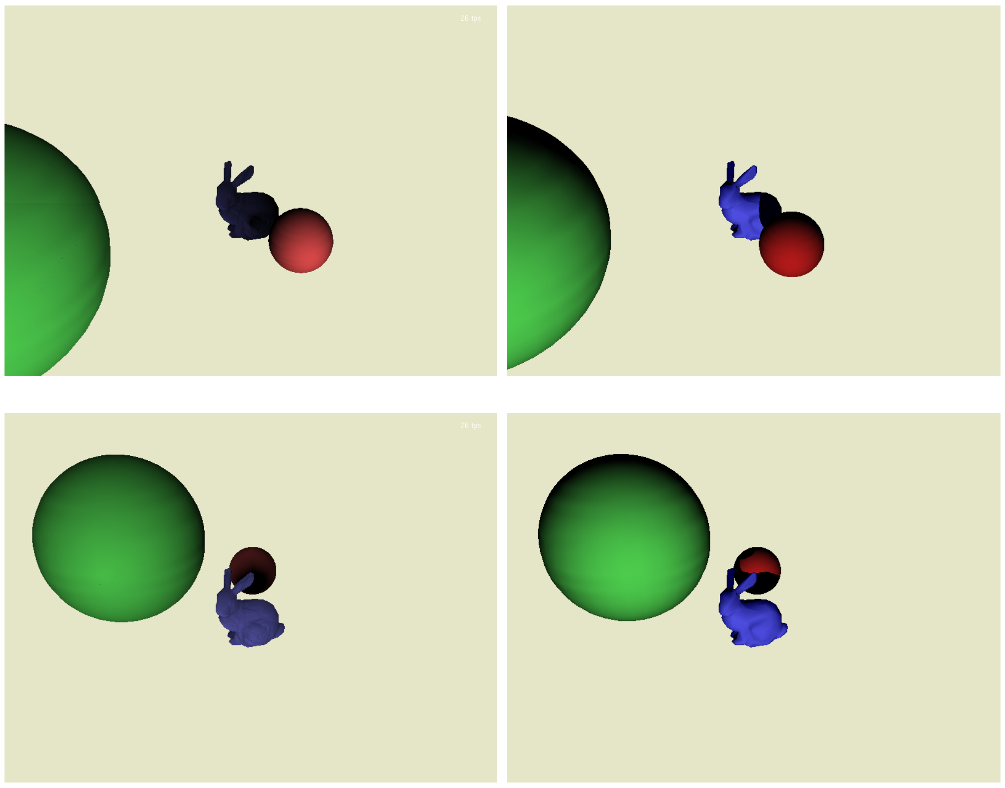


Figure 6.1: Bunny Hierarchy, Hierarchical PRT vs Shadow Volumes

for relatively quick changes in occlusion, because every texel's representation is independent. To recreate the visually pleasing soft shadows which occur over surfaces using our Textured PRT data, a very large number of sample points would be required. Calculating filter data through ray casting at all of those points would most likely result in computational intractability for most sampling densities.

Second, we believe that the Spherical Harmonic representation used is not amenable to representing the crisp, clear silhouettes which give effective shadowing from arbitrarily shaped shadow casters. The Spherical Harmonic representation set, though robust, is arranged in very circular patterns at each coarse-ness of basis function. Furthermore, Fourier style polynomial function reconstructions do not deal well with sharp discontinuities (as discussed before, in 4.3.8). Because we often want shadow to go from light to dark very quickly – a discontinuity – Spherical Harmonics turn out to not a great way to represent the necessary shadowing data. Combining these two issues has caused our resulting shadowing data, when made stable, to be relatively coarse in comparison with other real-time shadowing methods.

6.4 Conclusions

In this section we've discussed the results which we have found in our experiments implementing Textured PRT and Hierarchical PRT. We are relatively satisfied with the results we've gained through applying PRT techniques densely over meshes, though some potential subjective analysis remains in determining "optimal" textures sizes for different types of meshes.

Unfortunately, our exploration of Hierarchical PRT was less fruitful – while we have accomplished our goals in developing a system which can carry filter data downwards through a tree to provide occlusion information, our experiments have encountered unforeseen weaknesses in the standard representation. Though our system works well when the necessary "filter" function for the scene matches nicely with the Spherical Harmonic basis, our work has discovered cases which limits the viability of this approach.

Nevertheless, our exploratory work in Hierarchical PRT has generated soft, general shadows over dynamic scenes in a single pass. By comparison, conventional real-time shadowing algorithms like shadow volumes or shadow mapping often require a rendering pass for each light source, engendering concerns in complexity and scalability. Though conventional shadowing approaches utilizing hardware support, like shadow volumes, surpass the performance of our Hierarchical PRT tests, future optimizations possible utilizing our tree-like representation may narrow the gap.

6.5 Future Work

6.5.1 Textured PRT

To extend our work on textured PRT, we would be interested in seeing techniques which attempt to do more with high quality meshes being used to generate texture data. While we accomplish some feature preservation and reconstruction in this area, additional work could develop adaptive simplification techniques which correctly match “crevices” in the model with the self-shadowing shading variations which are possible using PRT.

6.5.2 Hierarchical PRT

We would be interested to see extensions to the Hierarchical PRT work which either utilize a different basis function which permits a more fine grained, stable product reprojection that we have been able to accomplish with the “masking product” method we have presented here. Such basis functions, however, will have to provided for the same arbitrary set of motions which Spherical Harmonics does (ie, rotations), which other basis function sets referenced in current literature do not.

Further, as mentioned in 5.1.3, we do not currently have a system which automatically groups elements into trees – that process is managed by hand. Work utilizing some adaptive clustering techniques might be able to do a better job, particularly over the course of an application's life, than a user can set up ahead of time.

Bibliography

[1] T. Akenine-Moller, T. Moller, and E. Haines. *Real-Time Rendering*. AK Peters, Ltd. Natick, MA, USA, 2002.

[2] F. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):242–248, 1977.

[3] CryTek. *CryEngine 3 Specifications*, May 2010.

[4] S. Domine. *Using Texture Compression in OpenGL*. NVIDIA.

[5] C. Everitt and M. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *NVIDIA White paper*, 6, 2002.

[6] T. Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991.

[7] C. E. John Spitzer. *Using Vertex Array Range and Fences*. NVIDIA, August 2000.

[8] J. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, page 150. ACM, 1986.

[9] T. Malzbender, D. Gelb, and H. Wolters. Polynomial texture maps. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer*

graphics and interactive techniques, pages 519–528, New York, NY, USA, 2001. ACM.

[10] A. K. Martin Stich, Carsten Wachter. *GPU Gems 3 - Efficient and Robust Shadow Volumes*, chapter 11. Addison-Wesley Professional, 2007.

[11] Newegg. Newegg desktop gpu memory amounts, June 2010.

[12] V. Schø̈nefeld. Spherical harmonics, July 2005.

[13] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics.* AK Peters, Ltd., 2009.

[14] P.-P. Sloan. Stupid spherical harmonics (sh) tricks. GDC 2008 Lecture, 2008.

[15] P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA, 2002. ACM.

[16] M. Slomp, M. Oliveira, and D. Patrício. A gentle introduction to precomputed radiance transfer. *RITA*, 13(2):131–160, 2006.

[17] W. Sun and A. Mukherjee. Generalized wavelet product integral for rendering dynamic glossy objects. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 955–966, New York, NY, USA, 2006. ACM.