Design Document
PBRT Extensions

Ian Vollick and Christian Lessig

1 Motivation

The most costly part of precomputation for PRT is creating per-vertex visibilty maps. Hardware accelerated rendering can be employed for generating these maps. We chose to add the necessary capabilities to `pbrt`, a physically-based ray tracer. The initial rationale for this choice was two-fold. `pbrt` provides a great deal of useful functionality such as a scene loader and a file writer. It also provides a convenient framework within which to compare hemispherical sampling and sampling over the hemicube. After abandoning the comparison of hemicube and hemispherical rendering, the rational for continuing with `pbrt` was that this work could be used in other research. It was thought that research in this area would be likely, given that graphics hardware is developing rapidly, and combinations of hardware and software techniques are commonplace in industry.

The goal is that the extented `pbrt` will be a testbed for using the latest hardware in the rendering pipeline and to allow fair comparisons of hardware and software rendering. It should be noted, however, that the extensions described in this report constitute a significant redefinition of `pbrt`. The hardware accelerated plugins cannot be considered physically-based.

2 Design Goals

- **Reusability**. The extensions should be as flexible as possible and applicable to a wide variety of problems. It should also be simple to understand what the extensions are for and how to use them.
- **Performance**. The extentions must permit faster precomputation than could be acheived using ray tracing.

3 Design Issues

3.1 Integrating New Functionality

It is crucial that new plugins to `pbrt` work seamlessly with the old. Otherwise, the use of the new plugins will necessarily be very specific, contradicting the stated goal of reusability.

- new plugins must work seamlessly with the old plugins.

- compatability detection?

3.2 Sampling

It is difficult to meet the performance design goal without making large-scale changes to the way that `pbrt` handle sampling. This is because the ray tracer sequentially processes pixels, whereas hardware processes them in parallel.

The naive approach is to use hardware to render a scene, cache the results, and lookup colors from the cache instead performing scene queries. This seamlessly integrates with `pbrt`. It can be slow however, since che cache is sampled sequetially.

A second approach is to generalize the notion of a sample. Currently `pbrt` uses point sampling along a single direction. The samples could be generalized to hold information obtained over a range of directions, as in beam tracing [Heckbert 1984]. That is, samples could be made to hold all the information obtained from a hardware render. This approach has not yet been implemented since it would require significant changes to pbrt. Instead, visibility information obtained from hardware rendering is directly writted to disk. This avoids the difficulty of passing generalized samples through the pipeline.

3.3 Rendering Sequences

For PRT and many other applications it is necessary to render large sequences of images. The overhead of loading the renderer and scene information for each image is unacceptable if the system is to meet the performance design goal. Therefore it is important to be able to load the scene data only once along with a sequence of camera positions. Adding this functionality to `pbrt` necessitated the creation of a new type of plugin, the renderer. Two renderer plugins have been written, the `still_renderer` which implements the old functionality, and the `sequence_renderer` which permits the user to specify a sequence of camera frames from which to render.

4 Limitations and Future Work

The current version of the `pbrt` extensions have the following limitations, each of which could be addressed in the future.

- A `Spectrum` subclass for beam samples is missing.
- Hemicube visibility calculations are done using the depth buffer, and this requires reading back visibility information from hardware six times per hemicube. It would be faster, however, to use the stencil buffer since the visibility information for all six faces of a hemicube can be stored simultaneously and read back at once, as is done in Kontkanen et. al. [Kontkanen 2005].
- No integrator has been written which actually renders the scene using hardware.
- The extensions described in this report only allow the generation of visibility maps for triangle meshes. Other geometry types supported by `pbrt` such as NURBS could be render in hardware using ray casting

5 Conclusion

The `pbrt` extensions represent significant progress towards meeting the stated design goals. However, more work needs to be done to improve interoperability with the rest of `pbrt`, particularly the default plugins, and to ensure that the software is platform independent. The extensions only provide the functionality required for PRT, but much of this functionality, such as the sequence renderer, is highly reusable. This code will be a

good reference for extending `pbrt` to employ graphics hardware for other portions of the rendering pipeline. It remains to be seen if this work will be of use to future researchers, but we believe it will be.

References

[Fabio et. al. 1995] Fabio Pellacini, Aaron Lefohn, Mark Leone, Kiril Vidimče, Alex Mohr, John Warren. Pixar animation studios. *Lpics: a Hybrid Hardware-Accelerated Relighting Engine for Computer Cinematography*, SIGGRAPH 1995

[Heckbert 1984] Paul S. Heckbert, Pat Hanrahan *Beam Tracing Polygonal Objects*, SIGGRAPH 1984

[Kontkanen 2005] Janne Kontkanen and Samuli Laine, *Ambient Occlusion Fields*, Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games, pp 41–48, 2005

A Precomputation: Combining Hemispherical Sampling and Hemicube

Preprocessing consists of rendering a hemicube visibility map for each vertex in the scene. Hemispherical sampling has also been implemented, but using a hemicube readily allows for the use of OpenGL and therefore hardware acceleration. The position and orientation of each hemicube can be aligned with a local coordinate system of each vertex. However, because there is no efficient rotation for wavelets, it is not sensible to use a local coordinate frame per vertex. Instead, when using wavelets as basis functions all hemicubes should be aligned in a global coordinate frame. Plugins with the postfix `global` can be used for these computations.

Below a pseudo-code description of the sequence renderer.

```
for i = 1 to NUM_VERTS {
  orientCamera( positions[i], ups[i], views[i] );

  // May open a new file for writing (and close the old)
  integrator.preProcess()

  while !sampler.done() {
    sample = sample.getNextSample()
    ray = camera.getRay( sample )

    // Render face.  May write hemicube face data to file
    integrator.render( ray, sample )

    film.addSample( sample )
  }

  // If the integrator has been doing file writing, this should
  // be a noop.
  // That is, if one uses opengl_hemicube_visibility_fast, one
  // should also use film_noop.
  film.writeImage()
}
```

Clearly, the sampler, camera, integrator, and film are interdependant. Several sample `.pbrt` files have been listed below to demonstrate which plugins should be used together. Better interoperability of the plugins, especially with the default ones, could be achieved with some additional effort.

B pbrt Extension Documentation

The following section describes the `pbrt` extensions implemented for this project. Please refer to the doxygen documentation for additional implementation details. Example `.pbrt` files have also been listed in a subsequent section to give an example of these plugins in context.

B.1 Samplers

- **Hemicube Sampler** The hemicube sampler works in conjunction with the hemicube camera described below. It is responsible for generating as many samples as needed per face of the hemicube. The sampler modifies the `time`, `imageX`, and `imageY` attributes of each sample to indicate its corresponding hemicube face, and its x and y coordinates, respectively.

  The number of samples needed per face varies depending on the file and integrator plugins employed for generating the final cubemap. Using the default `pbrt` rendering pipeline, the film needs at least one sample for every pixel of the cube map. To speed up the computation, the `hemicube_sampler` permits one to generate the cubemap using a single sample per face, used in conjunction with the `opengl_integrator_fast` and `film_noop` plugins. Sidestepping the default pipeline in this manner increases performance by a factor of 100 or more.
  Usage:

```
Sampler "hemicube_sampler"
  ["integer xresolution" xres]
  ["integer yresolution" yres]
```

B.2 Cameras

- **Hemicube Camera** Given the sample information described above, this camera generates rays which sample the hemicube above a vertex.
  Usage: `Camera "hemicube_camera"`
- **Hemisphere Camera** The hemicube camera maps image space coordinates to samples over the hemisphere. Specifically, if x and y are the image-space coordinates of the sample, N_x and N_y are horizontal and vertical resolutions, and M is the camera to world space transformation, then the ray's direction d is $M[\sin\theta\cos\phi, \cos\theta, \sin\theta\sin\phi]^T$, where

$$\theta = \left(\frac{\pi}{2}\right)\left(\frac{y}{N_y}\right), \text{ and } \phi = 2\pi\left(\frac{x}{N_x}\right).$$

  The hemisphere camera is a slightly modified version of the `environment` camera from default `pbrt` plugins.
  Usage: `Camera "hemisphere_camera"`

B.3 Surface Integrators

Surface integrators compute the radiance for each ray. In our case, however, we are not concerned with radiance, but visibility.

- **Visibility** The `visibility` integrator determines visibility via occlusion testing using ray tracing. It makes use of a specialized scene query function which only determines visibility; no shading is performed.
  Usage: `SurfaceIntegrator "visibility"`
- **OpenGL Hemicube Visibility** This integrator is similar to the visibility integrator above except that it calculates visibility using OpenGL and the result is cached. The per pixel visibility is determined by sampling the cached faces. Using the hardware accelerated OpenGL pipeline speeds up the computations significantly, but for simple scenes the cost performing scene queries to deterine visibility is also quite modest.

  It should be noted that the faces of the hemicube are rendered lazily. If the hemicubes are generated in the local vertex frames, only 5 sides need to be sampled since the bottom face will never be used. However, if the hemicubes are aligned in world space, all six must be rendered.

  Usage: `SurfaceIntegrator "opengl_hemicube_visibility"`
- **Fast OpenGL Hemicube Visibility** This integrator inherits from `opengl_hemicube_visibility`, but sidesteps `pbrt`'s rendering pipeline and writes the hemicube data directly after rendering a face. The faces are still calculated lazily. The faces are sampled in a predetermined order and as they are rendered, they are written to disk to simplify the usage of this data in subsequent stages of the pipeline. To do this, set the x and y resolution of the hemicube sampler to 1. The user of this integrator should also be sure to use a film that does not cause any file to be written to disk, see `film_noop` below.
  Usage:

```
SurfaceIntegrator "opengl_hemicube_visibility_fast"
  "string filename" filename
  "integer xresolution" xres
  "integer yresolution" yres
```

B.4 Film

- **Vismap** Stores run length encoded visibility image.
  Usage:

```
Film "vismap"
  "string filename" filename
  "integer xresolution" xres
  "integer yresolution" yres
```

- **Noop** To avoid redundant file writing, this film should be used with `opengl_hemicube_visibility_fast`.
  Usage: `Film "film_noop"`
- **Hemicube Vismap** Similar to a vismap, but writes run length encoded visibility information for each face of the hemicube.
  Usage:

```
Film "hemicube_vismap'
  "string filename" filename
  "integer xresolution" xres
  "integer yresolution" yres
```

B.5 Renderers

The rendering system in `pbrt` has been fully revised so that the renderer can be specified as a plugin. This has been necessary to enable the rendering of sequences without re-

initializing the whole rendering system for each frame.

- **Still Renderer** This plugin provides the functionality of `pbrt`'s default renderer.
  Usage: `Renderer "still_renderer"`
- **Sequence Renderer** This plugin renders a sequence of frames. For each frame, a camera position, an up vectors and a view vector are required. If the filename argument to the film (or possibly the integrator), was `basename.ext`, then the rendered images are of the form `basename.fnum.ext` where `fnum` is the frame number.
  Usage:

```
Film "sequence_renderer"
  "point positions" [ x1 y1 z1 x2 y2 z2 ... ]
  "vector views"    [ x1 y1 z1 x2 y2 z2 ... ]
  "vector ups"      [ x1 y1 z1 x2 y2 z2 ... ]
```

B.6 Accelerators

- **Object list** This pluginc stores the scene geometry as a flat list. It should be used with the OpenGL integrators mentioned above.
  Usage: `Accelerator "objectlist"`

B.7 General

The core functionality has also been extended and modified to match the increased requirements. For example, the RLE encoding has been abstracted into a class and added to `pbrt`'s core functionality.

C Example Scene Files

C.1 hemicube.pbrt

```
Film "hemicube_vismap"
  "string filename" ["foo.dat"]
  "integer xresolution" 512
  "integer yresolution" 512
Sampler "hemicube_sampler"
LookAt 0.0 0.0 0.0 0.0 0.0 -1.0 0.0 1.0 0.0
Camera "hemicube_camera"
SurfaceIntegrator "opengl_hemicube_visibility"
Accelerator "objectlist"
Include "foo_camera.pbrt"
Include "foo_scene.pbrt"
```

C.2 fastHemicube.pbrt

```
Film "film_noop"
Sampler "hemicube_sampler"
  "integer xresolution" 1
  "integer yresolution" 1
LookAt 0.0 0.0 0.0 0.0 0.0 -1.0 0.0 1.0 0.0
Camera "hemicube_camera"
SurfaceIntegrator "opengl_hemicube_visibility_fast"
```

```
  "string filename" "foo_fast.dat"
  "integer xresolution" 512
  "integer yresolution" 512
Accelerator "objectlist"
Include "foo_camera.pbrt"
Include "foo_scene.pbrt"
```