Introduction to PBRT

Phaedon Sinis & Alexis Haraux

CS348b • Spring 2012
STANFORD UNIVERSITY
Image credit:
"The State of the Art in Interactive Global Illumination",
Ritschel, Dachsbacher, Grosch, Kautz
FROM OPENGL TO PBRT

• You are now OpenGL Masters
• Triangles are easy
• You can render anything!
• What about chrome balls?

Photo Credit: http://www.flickr.com/photos/bunnyfrogs/galleries/72157623369298282/
<table>
<thead>
<tr>
<th>Effect</th>
<th>Rasterization</th>
<th>Ray Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadows</td>
<td>Shadow mapping</td>
<td>Trace a ray</td>
</tr>
<tr>
<td>Reflections</td>
<td>Environment mapping</td>
<td>Trace a ray</td>
</tr>
<tr>
<td>Refraction</td>
<td>Environment mapping</td>
<td>Trace a ray</td>
</tr>
<tr>
<td>Depth of Field</td>
<td>Post process blur</td>
<td>Trace more rays</td>
</tr>
<tr>
<td>Motion Blur</td>
<td>Post process blur</td>
<td>Trace more rays</td>
</tr>
<tr>
<td>Arbitrary Primitive</td>
<td>Triangulate ?</td>
<td>Trace a ray</td>
</tr>
<tr>
<td>Global Illumination</td>
<td>Many ways</td>
<td>Trace more rays</td>
</tr>
</tbody>
</table>
EXAMPLE: SHADOWS

"Area where direct light from a light source cannot reach due to obstruction by an object"
Wikipedia

OpenGL
• Render scene from point of view of the light
• Store depth into a texture (FBO)
• Transform fragment into light space
• Test depth with shadow map value

Ray Tracing
• Trace a ray to the light
EXAMPLE: SPECULAR REFLECTION

"the direction of incoming light, and the direction of outgoing light reflected make the same angle with respect to the surface normal" -Wikipedia

OpenGL

• Place the camera at the center of the object

• Render the scene for each face of a cube into a texture (FBO)

• Reflect the view vector by the normal

• Lookup into the cube map

Ray Tracing

• Trace a ray in the reflected direction
PIPELINES SIDE BY SIDE

OpenGL

Application

Vertex

Assembly

Rasterization

Fragment

FB Ops
Display

PBRT

Scene File Parsing

Intersection

Integration

Image

Data + Parameters

Transforms
Projection

Sampling

Shading
**FORGET ABOUT OPENGL ?**

<table>
<thead>
<tr>
<th>OpenGL</th>
<th>PBRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Triangle meshes</td>
<td>• Material: BRDF, BTDF, BSDF</td>
</tr>
<tr>
<td>• Textures</td>
<td>• Always debug in low res</td>
</tr>
<tr>
<td>• Phong Shading</td>
<td>• Write your own tools</td>
</tr>
<tr>
<td>• Debugging on screen (normals)</td>
<td>• Use OpenGL for visualization!</td>
</tr>
<tr>
<td>• Fast iteration</td>
<td></td>
</tr>
<tr>
<td>• Interactivity</td>
<td></td>
</tr>
</tbody>
</table>
Table 1.1: Main Interface Types. Most of pbrt is implemented in terms of 13 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

<table>
<thead>
<tr>
<th>Base class</th>
<th>Directory</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td>shapes/</td>
<td>3.1</td>
</tr>
<tr>
<td>Aggregate</td>
<td>accelerators/</td>
<td>4.2</td>
</tr>
<tr>
<td>Camera</td>
<td>cameras/</td>
<td>6.1</td>
</tr>
<tr>
<td>Sampler</td>
<td>samplers/</td>
<td>7.2</td>
</tr>
<tr>
<td>Filter</td>
<td>filters/</td>
<td>7.7</td>
</tr>
<tr>
<td>Film</td>
<td>film/</td>
<td>7.8</td>
</tr>
<tr>
<td>Material</td>
<td>materials/</td>
<td>9.2</td>
</tr>
<tr>
<td>Texture</td>
<td>textures/</td>
<td>10.3</td>
</tr>
<tr>
<td>VolumeRegion</td>
<td>volumes/</td>
<td>11.3</td>
</tr>
<tr>
<td>Light</td>
<td>lights/</td>
<td>12.1</td>
</tr>
<tr>
<td>Renderer</td>
<td>renderers/</td>
<td>1.3.3</td>
</tr>
<tr>
<td>SurfaceIntegrator</td>
<td>integrators/</td>
<td>Ch. 15 intro</td>
</tr>
<tr>
<td>VolumeIntegrator</td>
<td>integrators/</td>
<td>16.2</td>
</tr>
</tbody>
</table>
Let's begin by exploring PBRT's classes in more detail
Film class is used by renderer to output an image.

*You specify:*
- Image dimensions
- Crop window == Great for debugging!
- Output file: EXR, TGA, PFM

The "image" is required as it's the only subclass in PBRT
• Not all Renderers actually generate output images!

• Other uses: testing, preprocessing for subsurface scattering

• Default: **SamplerRenderer**
Integrator

• Implements light transport algorithm

• Default: `DirectLightingIntegrator`

• other surface integrators:
  - photon mapping, ambient occlusion, Whitted, etc

• volume integrators for single-scattering
Sampler

Sampler "lowdiscrepancy" "integer pixelsamples" [2]

- Default: low-discrepancy (**LDSampler**)
  - chooses a bunch of "random" samples with nice properties (points are somewhat evenly distributed across domain)

- others: stratified, adaptive, random...

- details to be covered in a few weeks
Camera

Camera "perspective" "float fov" [45]

• You can define:
  - shutter speed => motion blur
  - lens radius & focal distance => depth of field

• Default: PerspectiveCamera

• others: environment, orthographic

• EnvironmentCamera traces rays in *all* directions around a point.
Materials

• BSDFs to describe surface scattering of light

Provided in PBRT:
- glass
- matte
- metal
- mirror
- plastic
- translucent
- etc etc!
# floor

Texture "hardwood" "spectrum" "imagemap"
"string filename" ".../textures/floor.tga"

Material "matte" "texture Kd" "hardwood"

Shape "trianglemesh"
"integer indices" [0 1 2 1 3 2 ]
"point P" [ -4 -1.9 -5  4 -1.9 -5
       -4 -1.9  3  4 -1.9  3 ]
"float uv" [ 0 0 1 0 0 1 1 1 1 ]
Textures

• Also: bump-mapping is built in!
Film "image" "string filename" ["cs248scene.exr"]
   "integer xresolution" [400] "integer yresolution" [400]

Sampler "lowdiscrepancy" "integer pixelsamples" [2]

LookAt -40 0 4 0 0 5 0 0 1
Camera "perspective" "float fov" [45]

WorldBegin

LightSource "point" "rgb I" [50 50 50] "point from" [0 0 5]

AttributeBegin
   Include "cathedral.pbrt"
AttributeEnd

WorldEnd
Scene file output

(0.5 seconds on 8 cores)
Lights, Camera, Armadillo

(1554 seconds on 8 cores)
[A look at a real scene file from last year]
What happens when you render a PBRT scene file?

Let's trace the code.
PBRT source code in one slide

```c
int main(...) {
  // Parse command line
  // Do some stuff
  ParseFile(filenames[i]);
}

bool ParseFile(...) {
  // Do more stuff
  yyparse();
}
```

Data structures:
- ImageFilm
- PerspectiveCamera
- SpotLight
- GlassMaterial
- TriangleMesh

```c
void SamplerRendererTask::Run() {
  for each sample in this task {
    // shoot some rays
    // eval radiance:
    Ls[i] = rayWeight * renderer->Li(scene, rays[i], ...);
  }
  // Then pass each sample's radiance to camera:
  camera->film->AddSample(samples[i], Ls[i]);
}
```

```c
void SamplerRenderer::Render(const Scene *scene) {
  // preprocess the scene using appropriate integrators
  // compute dimensions and number of threads/tasks
  // put a bunch of SamplerRendererTasks in a vector
  for each SamplerRendererTask
    tasks[i]->Run();
}
```

```c
void pbrtWorldEnd() {
  // Setup
  // Create scene and render
  Renderer *renderer = renderOptions->MakeRenderer();
  Scene *scene = renderOptions->MakeScene();
  if (scene && renderer) renderer->Render(scene);
  // Cleanup
}
```
PBRT Architecture
Let's extend PBRT
(Live example)

In the review session, we saw how easy it is to extend PBRT with a new class. Here are the steps:

• Create new classes:
  cp src/cameras/perspective.h src/cameras/crazycam.h
  cp src/cameras/perspective.cpp src/cameras/crazycam.cpp

• Change all the class names in the new files from PerspectiveCamera to CrazyCamera

• Add the following two lines to the end of the CrazyCamera::GenerateRayDifferential() method in crazycam.cpp:
  // CRAZINESS!!!!
  ray->d.x *= 0.5*sin(8.0*ray->d.x) + 1;
  ray->d.y *= 0.5*sin(ray->d.y) + 1;

• Go to core/api.cpp and add the following condition to the MakeCamera() function:
  else if (name == "crazycamera")
    camera = CreateCrazyCamera(paramSet, animatedCam2World, film);

  and also add the header to the includes at the top.

• The makefile will automatically compile your new classes!!!
Let's extend PBRT
(Live example)

PerspectiveCamera

CrazyCamera
Final thoughts

• Meshes can be a pain to import.
  • MeshLab is a great (and free) tool:
    - http://meshlab.sourceforge.net/

• Always think about simple debugging approaches
  - especially on final project
  - may need to write scripts
  - visualize elements in OpenGL!
    (recycle your CS248 code)

• Enjoy the photons!

*Image credit: Ólafur Eliasson, One-way colour tunnel, 2007, SF MOMA*