PBRT core

Digital Image Synthesis

Yung-Yu Chuang

9/24/2008

with slides by Pat Hanrahan

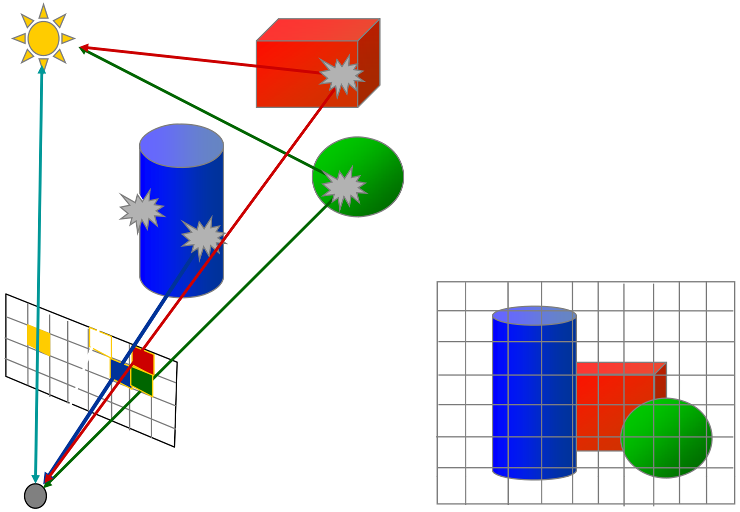
Announcements

- Please subscribe the mailing list.
- Doxygen (online, download or doxygen by yourself)
- HW#1 will be assigned next week (10/1) and due on 10/22.
- The class of 10/29 is cancelled because I will be attending ACM Multimedia 2008 in Vancouver.

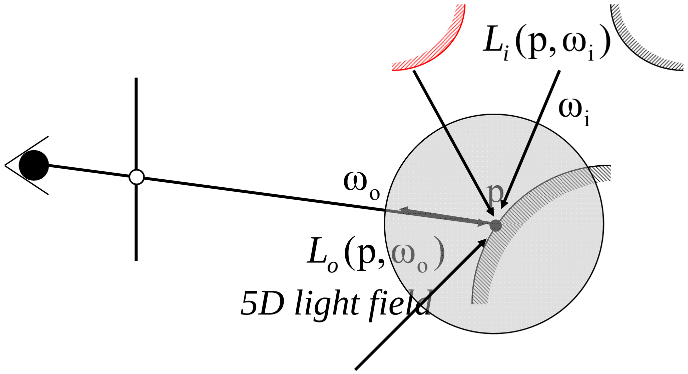
This course

- Study of how state-of-art ray tracers work

Ray casting



Rendering equation (Kajiya 1986)


$L_i(\mathrm{p},\omega_\mathrm{i})$
ω_i
ω_o
p
$L_o(\mathrm{p},\omega_\mathrm{o})$
5D light field

$$L_o(\mathrm{p},\omega_\mathrm{o}) = L_e(\mathrm{p},\omega_\mathrm{o}) + \int_{s^2} f(\mathrm{p},\omega_\mathrm{o},\omega_\mathrm{i}) L_i(\mathrm{p},\omega_\mathrm{i}) \left|\cos\theta_\mathrm{i}\right| \, d\omega_\mathrm{i}$$

pbrt

- pbrt (physically-based ray tracing) attempts to simulate physical interaction between light and matter based on ray tracing.
- A plug-in architecture: core code performs the main flow and defines the interfaces to plug-ins. Necessary modules are loaded at run time as DLLs, so it is easy to extend the system without modifying the core.

pbrt plug-ins (see source browser also)

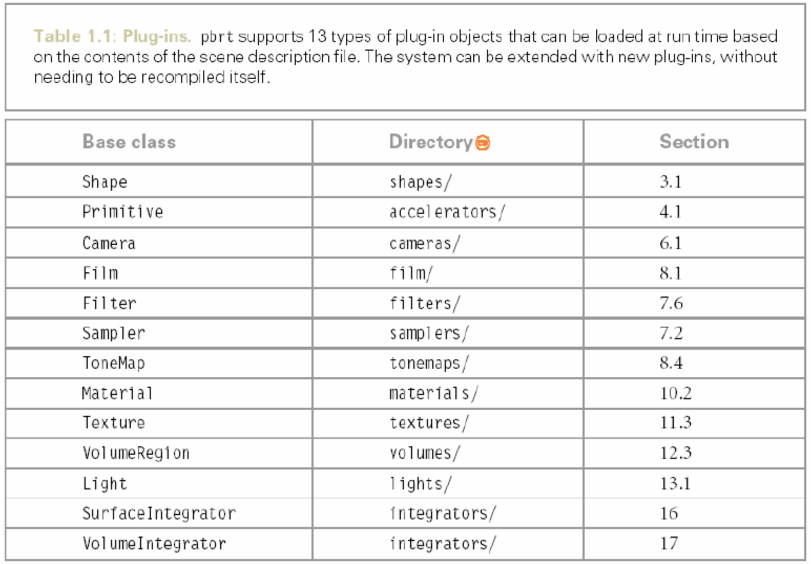


Table 1.1: Plug-ins. pbrt supports 13 types of plug-in objects that can be loaded at run time based on the contents of the scene description file. The system can be extended with new plug-ins, without needing to be recompiled itself.

Base class	Directory	Section
Shape	shapes/	3.1
Primitive	accelerators/	4.1
Camera	cameras/	6.1
Film	film/	8.1
Filter	filters/	7.6
Sampler	samplers/	7.2
ToneMap	tonemaps/	8.4
Material	materials/	10.2
Texture	textures/	11.3
VolumeRegion	volumes/	12.3
Light	lights/	13.1
SurfaceIntegrator	integrators/	16
VolumeIntegrator	integrators/	17

Phases of execution

- **`main()`** in renderer/pbrt.cpp

```
int main(int argc, char *argv[]) {
  <Print welcome banner>
  pbrtInit();
  // Process scene description
  if (argc == 1) {
    // Parse scene from standard input
    ParseFile("-");
  } else {
    // Parse scene from input files
    for (int i = 1; i < argc; i++)
      if (!ParseFile(argv[i]))
        Error("Couldn't open …\"%s\"\n", argv[i]);
  }
  pbrtCleanup();
  return 0;
}
```

Example scene

```
LookAt 0 10 100   0 -1 0 0 1 0
Camera "perspective" "float fov" [30]
PixelFilter "mitchell"
            "float xwidth" [2] "float ywidth" [2]
Sampler "bestcandidate"
Film "image" "string filename" ["test.exr"]
     "integer xresolution" [200]
     "integer yresolution" [200]
# this is a meaningless comment
WorldBegin

AttributeBegin
  CoordSysTransform "camera"
  LightSource "distant"
              "point from" [0 0 0] "point to" [0 0 1]
              "color L"    [3 3 3]
AttributeEnd
```

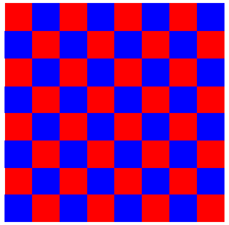
rendering options

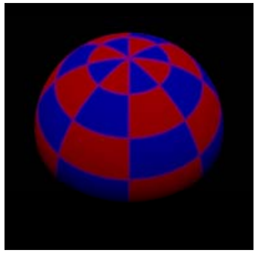
id “type” param-list

“type name” [value]

Example scene

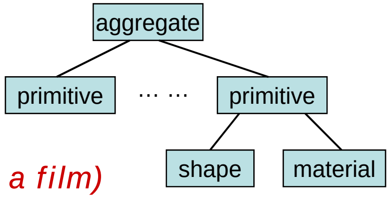
```
AttributeBegin
  Rotate 135 1 0 0

  Texture "checks" "color" "checkerboard"
          "float uscale" [8] "float vscale" [8]
          "color tex1" [1 0 0] "color tex2" [0 0 1]

  Material "matte"
           "texture Kd" "checks"
  Shape "sphere" "float radius" [20]
AttributeEnd
WorldEnd
```





Scene parsing (Appendix B)

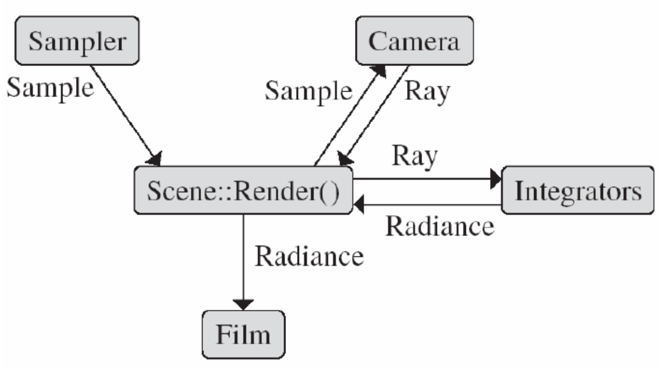
- core/pbrtlex.l and core/pbrtparse.y
- After parsing, a **scene** object is created (core/scene.*)



```
class scene {
  Primitive *aggregate;
  vector<Light *> lights;
  Camera *camera;  (contains a film)
  VolumeRegion *volumeRegion;
  SurfaceIntegrator *surfaceIntegrator;
  VolumeIntegrator *volumeIntegrator;
  Sampler *sampler; (generates sample positions
  BBox bound;        for eye rays and integrators)
};
```

Rendering

- **`Scene::Render()`** is invoked.

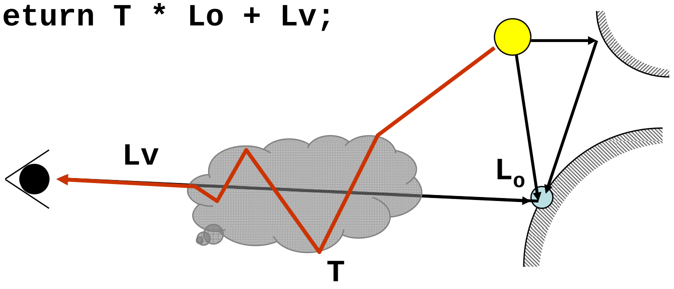


Scene::Render()

```
while (sampler->GetNextSample(sample)) {
  RayDifferential ray;
  float rayW=camera->GenerateRay(*sample,&ray);
   for effects such as vignetting
  <Generate ray differentials for camera ray>

  float alpha; opacity along the ray
  Spectrum Ls = 0.f;
  if (rayW > 0.f)
     Ls = rayW * Li(ray, sample, &alpha);
  ...
  camera->film->AddSample(*sample,ray,Ls,alpha);
  ...
}
```

Scene::Li

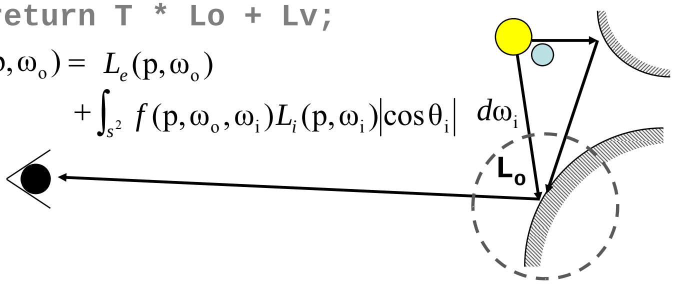
```
Spectrum Scene::Li(RayDifferential &ray,
                   Sample *sample, float *alpha)
{
  Spectrum Lo=surfaceIntegrator->Li(…);
  Spectrum T=volumeIntegrator->Transmittance(…);
  Spectrum Lv=volumeIntegrator->Li(…);
  return T * Lo + Lv;
}
```



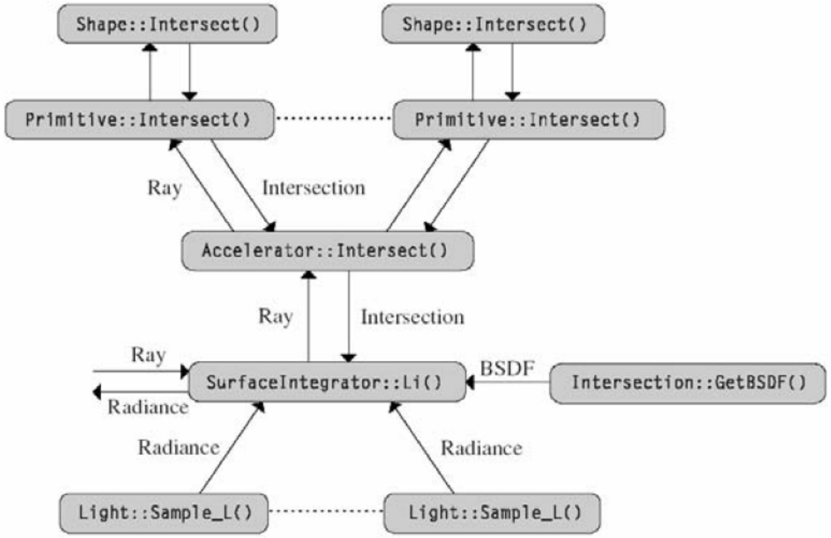
Surface integrator

```
Spectrum Scene::Li(RayDifferential &ray,
                   Sample *sample, float *alpha)
{
  Spectrum Lo=surfaceIntegrator->Li(…);
  Spectrum T=volumeIntegrator->Transmittance(…);
  Spectrum Lv=volumeIntegrator->Li(…);
  return T * Lo + Lv;
```

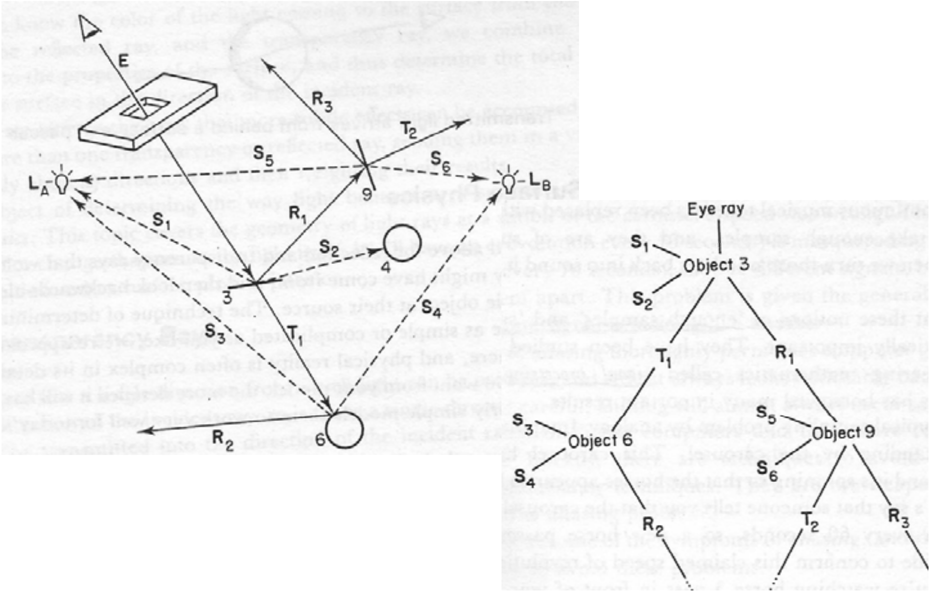
$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{s^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i)|\cos\theta_i| \, d\omega_i$$



Surface integrator



Whitted model

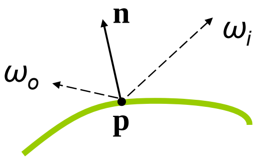


Whitted integrator

- in integrators/whitted.cpp

```
class WhittedIntegrator:public SurfaceIntegrator

Spectrum WhittedIntegrator::Li(Scene *scene,
    RayDifferential &ray, Sample *sample, float *alpha)
{
  ...
  bool hitSomething=scene->Intersect(ray,&isect);
  if (!hitSomething) {include effects of lights without geometry}
  else {
    ...
    <Computed emitted and reflect light at isect>
  }
}
```

Whitted integrator



```
 BSDF *bsdf=isect.GetBSDF(ray);
 ...
 Vector wo=-ray.d;
 L+=isect.Le(wo);
 Vector wi; direct lighting
 for (u_int i = 0; i < scene->lights.size(); ++i) {
   VisibilityTester visibility;
   Spectrum Li = scene->lights[i]->
                         Sample_L(p, &wi, &visibility);
   if (Li.Black()) continue;
   Spectrum f = bsdf->f(wo, wi);
   if (!f.Black() && visibility.Unoccluded(scene))
      L += f * Li * AbsDot(wi, n) *
                  visibility.Transmittance(scene);
}
```

Whitted integrator

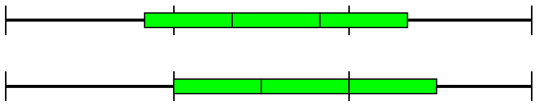
```
if (rayDepth++ < maxDepth) {
  Spectrum f = bsdf->Sample_f(wo, &wi,
         BxDFType(BSDF_REFLECTION | BSDF_SPECULAR));
  if (!f.Black()) {
    <compute rd for specular reflection>
    L += scene->Li(rd, sample) * f * AbsDot(wi, n);
  }
  f = bsdf->Sample_f(wo, &wi,
      BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR));
  if (!f.Black()) {
    <compute rd for specular transmission>
    L += scene->Li(rd, sample) * f * AbsDot(wi, n);
  }
}
```

Code optimization

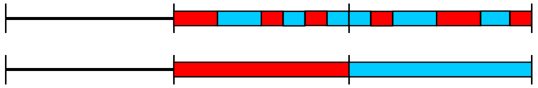
- Two commonly used tips
  - Divide, square root and trigonometric are among the slowest (10-50 times slower than +*). Multiplying 1/r for dividing r.
  - Being cache conscious

Cache-conscious programming

- **`alloca`**
- **`AllocAligned(), FreeAligned()`** make sure that memory is cache-aligned



- Use union and bitfields to reduce size and increase locality
- Split data into hot and cold



Cache-conscious programming

- Arena-based allocation allows faster allocation and better locality because of contiguous addresses.
- Blocked 2D array, used for film