PBRT core

Digital Image Synthesis

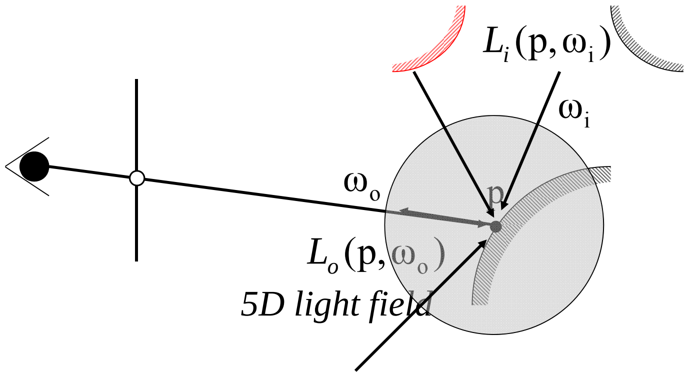
Yung-Yu Chuang

with slides by Pat Hanrahan

This course

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

Rendering equation (Kajiya 1986)



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

Basic components

- Cameras
- Ray-object intersection
- Light distribution
- Visibility
- Surface scattering
- Recursive ray tracing
- Ray propagation (in volume)

pbrt

- pbrt (physically-based ray tracing) attempts to simulate physical interaction between light and matter based on ray tracing.
- Structured using object-oriented paradigm: abstract base classes are defined for important entities and their interfaces
- It is easy to extend the system. New types inherit from the appropriate base class, are compiled and linked into the system.

pbrt abstract classes (see source browser)

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.

Base class	Directory	Section
Shape	shapes/	3.1
Aggregate	accelerators/	4.2
Camera	cameras/	6.1
Sampler	samplers/	7.2
Filter	filters/	7.7
Film	film/	7.8
Material	materials/	9.2
Texture	textures/	10.3
VolumeRegion	volumes/	11.3
Light	lights/	12.1
Renderer	renderers/	1.3.3
SurfaceIntegrator	integrators/	Ch. 15 intro
VolumeIntegrator	integrators/	16.2

Phases of execution

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

```
int main(int argc, char *argv[]) {
  Options options;
  vector<string> filenames;
  <Process command-line arguments>
  pbrtInit(options);
  if (filename.size() == 0) {
    // Parse scene from standard input
    ParseFile("-");
  } else {
    // Parse scene from input files
    for (int i = 0; i < filenames.size(); i++)
      if (!ParseFile(filenames[i]))
        Error("Couldn't open …, filenames[i].c_str());
  }
  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
```

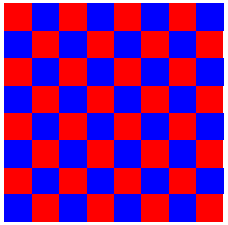
rendering options

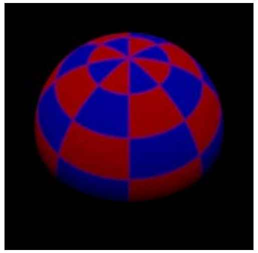
id "type" param-list

"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
```

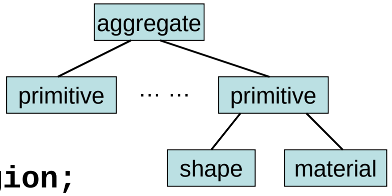




Scene parsing (Appendix B)

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

```
class scene {
  Primitive *aggregate;
  vector<Light *> lights;
  VolumeRegion *volumeRegion;
  BBox bound;
};
```



Rendering

- Rendering is handled by **Renderer** class.

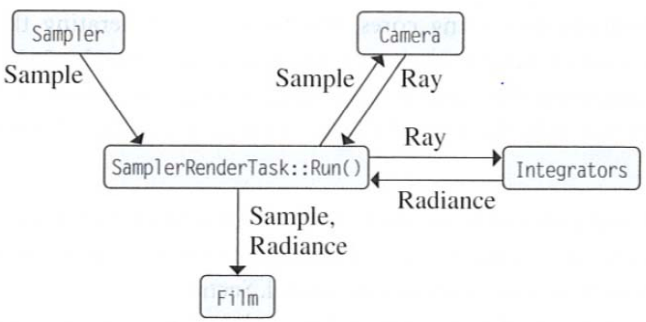
```
class Renderer {
  …   given a scene, render an image or a set of measurements
  virtual void Render(Scene *scene) = 0;

        computer radiance along a ray
  virtual Spectrum Li(Scene *scn, RayDifferential &r,
for MC sampling Sample *sample, RNG &rng,
            MemoryArena &arena, Intersection *isect,
 transmittance Spectrum *T) const = 0;
       return transmittance along a ray
  virtual Spectrum Transmittance(Scene *scene,
            RayDifferential &ray, Sample *sample,
            RNG &rng, MemoryArena &arena) const = 0;
};   The later two are usually relayed to Integrator
```

SamplerRenderer

```
class SamplerRenderer : public Renderer {
  …
  private:
    // SamplerRenderer Private Data
    Sampler *sampler;   choose samples on image plane
                        and for integration

    Camera *camera;   determine lens parameters (position,
                      orientation, focus, field of view)
                      with a film

    SurfaceIntegrator *surfaceIntegrator;
    VolumeIntegrator *volumeIntegrator;
};             calculate the rendering equation
```

The main rendering loop

- After **scene** and **Renderer** are constructed, **Renderer:Render()** is invoked.



Renderer:Render()

```
void SamplerRenderer::Render(const Scene *scene) {
  …
```

scene-dependent initialization such photon map

```
  surfaceIntegrator->Preprocess(scene,camera,this);
  volumeIntegrator->Preprocess(scene,camera,this);
```

sample structure depends on types of integrators

```
  Sample *sample = new Sample(sampler,
        surfaceIntegrator, volumeIntegrator, scene);
```

We want many tasks to fill in the core (see histogram next page). If there are too few, some core will be idle. But, threads have overheads. So, we do not want too many either.

```
  int nPixels = camera->film->xResolution
                 * camera->film->yResolution;
  int nTasks = max(32 * NumSystemCores(),
                 nPixels / (16*16));
  nTasks = RoundUpPow2(nTasks);
```

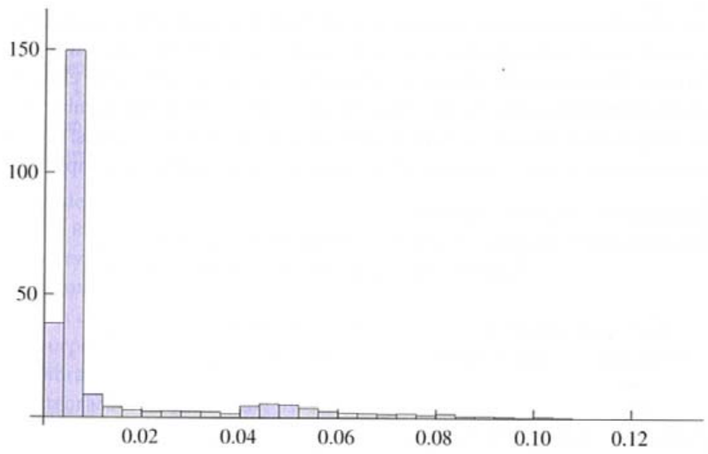
at least 32 tasks for a core

a task is about 16x16

power2 easier to divide

Histogram of running time

- The longest-running task took 151 times longer than the slowest.



Renderer:Render()

```
  vector<Task *> renderTasks;
  for (int i = 0; i < nTasks; ++i)
    renderTasks.push_back(new
    SamplerRendererTask(scene,this,camera,reporter,
      sampler, sample, nTasks-1-i, nTasks));

  EnqueueTasks(renderTasks);
  WaitForAllTasks();
  for (int i = 0; i < renderTasks.size(); ++i)
    delete renderTasks[i];

  delete sample;
  camera->film->WriteImage();
}
```

all information about renderer must be passed in

task id

total tasks

SamplerRenderTask::Run

- When the task system decided to run a task on a particular processor, **SamplerRenderTask::Run()** will be called.

```
void SamplerRendererTask::Run() {
  // decided which part it is responsible for
  …
  int sampleCount;
  while ((sampleCount=sampler ->
    GetMoreSamples(samples, rng)) > 0) {
    // Generate camera rays and compute radiance
```

SamplerRenderTask::Run

```
  for (int i = 0; i < sampleCount; ++i) {

  float rayWeight = camera->
      GenerateRayDifferential(samples[i], &rays[i]);
  rays[i].ScaleDifferentials(
             1.f / sqrtf(sampler->samplesPerPixel));

  if (rayWeight > 0.f)
    Ls[i] = rayWeight * renderer->Li(scene, rays[i],
      &samples[i], rng, arena, &isects[i], &Ts[i]);
  else { Ls[i] = 0.f; Ts[i] = 1.f; }

  for (int i = 0; i < sampleCount; ++i)
    camera->film->AddSample(samples[i], Ls[i]);
}
```

for vignetting

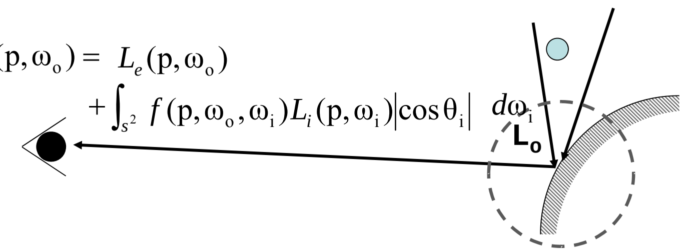
ray differential for antialiasing

SamplerRender::Li

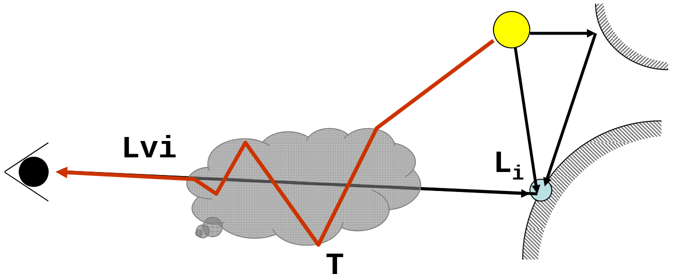
```
Spectrum SamplerRenderTask::Li(Scene *scene,
  RayDifferential &ray, Sample *sample,
  …, Intersection *isect, Spectrum *T)
{ Spectrum Li = 0.f;
  if (scene->Intersect(ray, isect))
    Li = surfaceIntegrator->Li(scene,this,
                ray, *isect, sample,rng, arena);
  else { // ray that doesn't hit any geometry
    for (i=0; i<scene->lights.size(); ++i)
      Li += scene->lights[i]->Le(ray);
  }
  Spectrum Lvi = volumeIntegrator->Li(scene, this,
                 ray, sample, rng, T, arena);
  return *T * Li + Lvi;
}
```

Surface integrator's Li

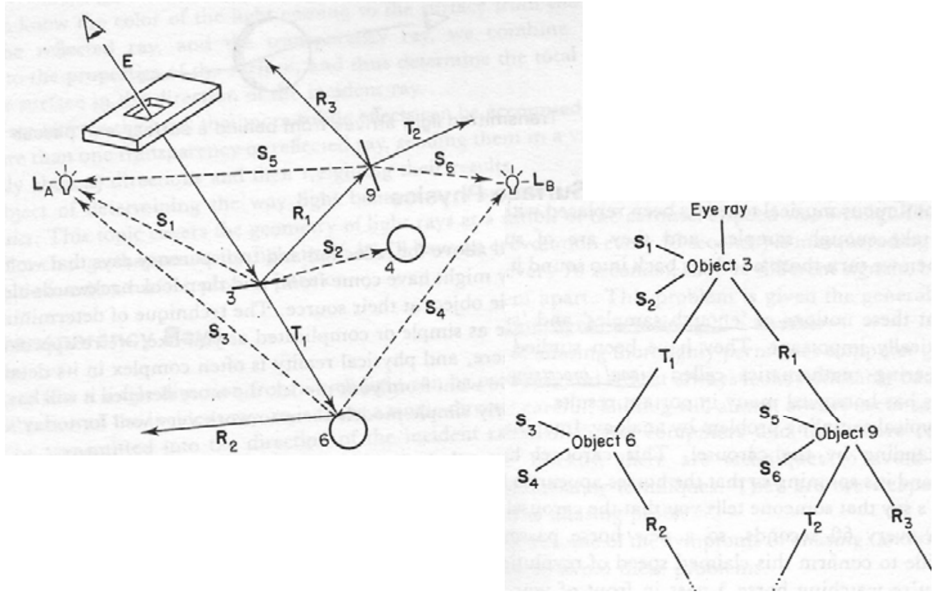
$$L_o(\mathrm{p},\omega_o) = L_e(\mathrm{p},\omega_o) + \int_{s^2} f(\mathrm{p},\omega_o,\omega_i) L_i(\mathrm{p},\omega_i) |\cos\theta_i| \, d\omega_i$$



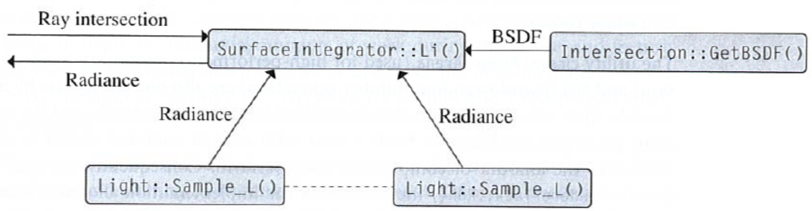
SamplerRender::Li



Whitted model



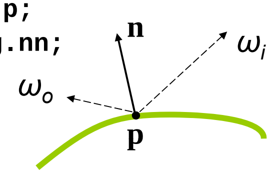
Whitted integrator



Whitted integrator

- in integrators/whitted.cpp

```
class WhittedIntegrator:public SurfaceIntegrator
Spectrum WhittedIntegrator::Li(Scene *scene,
        Renderer *renderer, RayDifferential &ray,
        Intersection &isect, Sample *sample, RNG &rng,
        MemoryArena &arena) const
{
  BSDF *bsdf = isect.GetBSDF(ray, arena);

  //Initialize common variables for Whitted integrator
  const Point &p = bsdf->dgShading.p;
  const Normal &n = bsdf->dgShading.nn;
  Vector wo = -ray.d;
```



Whitted integrator

```
// Compute emitted light if ray hits light source
L += isect.Le(wo);
// Add contribution of each light source direct lighting
for (i = 0; i < scene->lights.size(); ++i) {
  Vector wi;
  float pdf;
  VisibilityTester visibility;
  Spectrum Li = scene->lights[i]->Sample_L(p,
    isect.rayEpsilon, LightSample(rng), ray.time,
    &wi, &pdf, &visibility);
  if (Li.IsBlack() || pdf == 0.f) continue;
  Spectrum f = bsdf->f(wo, wi);
  if (!f.IsBlack() && visibility.Unoccluded(scene))
    L += f * Li * AbsDot(wi, n) *
       visibility.Transmittance(scene, renderer,
       sample, rng, arena) / pdf;
}
```

Whitted integrator

```
  if (ray.depth + 1 < maxDepth) {
  // Trace for specular reflection and refraction
  L += SpecularReflect(ray, bsdf, rng, isect,
                       renderer, scene, sample, arena);
  L += SpecularTransmit(ray, bsdf, rng, isect,
                       renderer, scene, sample, arena);
  }
  return L;
}
```

SpecularReflect

In core/integrator.cpp utility functions

```
Spectrum SpecularReflect(RayDifferential &ray,
  BSDF *bsdf, RNG &rng, const Intersection &isect,
  Renderer *renderer, Scene *scene, Sample *sample,
  MemoryArena &arena)
{
    Vector wo = -ray.d, wi;
    float pdf;
    const Point &p = bsdf->dgShading.p;
    const Normal &n = bsdf->dgShading.nn;

    Spectrum f = bsdf->Sample_f(wo, &wi,
      BSDFSample(rng), &pdf,
      BxDFType(BSDF_REFLECTION | BSDF_SPECULAR));
```

SpecularReflect

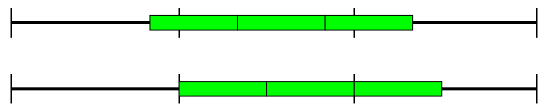
```
  Spectrum L = 0.f;
  if (pdf > 0.f && !f.IsBlack()
       && AbsDot(wi, n) != 0.f) {
    <Compute ray differential _rd_ for specular
     reflection>

    Spectrum Li = renderer->Li(scene, rd, sample,
                               rng, arena);
    L = f * Li * AbsDot(wi, n) / pdf;
  }
  return L;
}
```

Code optimization

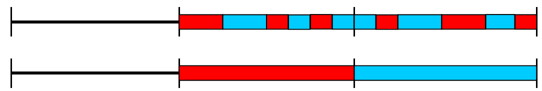
- Two commonly used tips
  - Divide, square root and trigonometric are among the slowest (10-50 times slower than +*). Multiplying 1/r for dividing r.
  - Being cache conscious

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