Assignment 4:
Progressive photon mapping

EDAN30

April 11, 2011

In this assignment you will be asked to extend your ray tracer to support progressive photon mapping. In order to pass the assignment you need to complete all tasks. Make sure you can explain your solutions in detail.

1 Scene setup

We start with the Cornell box from the previous assignment, no reflection and refraction, and with a point light.

Task 1: *Render a reference image using your path tracer.* You need this in order to validate your result using progressive photon mapping.

2 Progressive photon mapping

Progressive photon mapping is a technique that handles many types of light paths in a robust manner. In particular, the difficult case where a scene is dominated by caustics from small light sources. This part of the assignment requires you to implement the algorithm, step by step.

2.1 Forward tracing pass

Progressive photon mapping is a two pass algorithm that first performs a single forward tracing pass, from the eye, and then multiple photon tracing passes, from light sources. Like a whitted tracer, the forward tracing pass

traces rays trough the pixels of the image. Sphere shaped sampling regions are registered at each intersection, called *hit points*. A hit point remembers which pixel it belongs to and is used to gather flux from photons interacting near the hit point.

To generate an image, we go through each hit point an accumulate radiance in the pixel the hit point belongs to. In order to support super sampling (and later reflection and refraction) the hit point stores a weight indicating how much of its radiance should contribute to the pixel. E.g., if 4 samples per pixel is used, the weight of each resulting hit point should be 0.25.

Task 2: *Implement the forward tracing pass and calculate direct illumination at each hit point. Accumulate the weighted radiance of each hit point to the corresponding pixel.* In this step, the true radiance is not yet known - use direct illumination. Set the initial radius of the hit points to 0.5.

A hit point should contain the following information:

```
struct Hitpoint {
    Intersection is;
    int pixelX, pixelY;
    float pixelWeight;
    float radius;
    Color directIllumination;
    float photonCount;
    int newPhotonCount;
    Color totalFlux;
};
```

Note that the intersection data structure contains information like BRDF and surface normal. The last three fields should be initialized to zero.

Task 3: *Construct a BVH around the hit points to accelerate point-in-sphere tests.* Augment your old BVH ray-accelerator to use *Hitpoint* structures instead of *Intersectable*. Change the BVH traversal to use a simple point-in-box test, instead of ray-box intersection.

2.2 Photon tracing pass

In each photon tracing pass we trace photons from the light source to the scene and let them bounce around. There is always a single photon; russian roulette is used to choose between reflection, refraction, diffuse bounce and absorption (termination). For each bounce, the photon's flux is accumulated in overlapping hit points. For now, we stick to diffuse bounces.

The power of a photon is described by its flux, Φ_p. When a photon is emitted from a point light, it has the flux:

$$\Phi_p = 4\pi I$$

where I is the intensity of the light source. Note that each photon carries the full power of the light source. We compensate for this later by dividing by the number of photons emitted.

In order to determine what happens when a photon hits a diffuse surface, we look at the rendering equation. Since we have a single incident photon, the rendering equation becomes:

$$L(x \to \Theta) = f_r(x, \Psi_p \leftrightarrow \Theta) L_p(x \leftarrow \Psi_p) cos(N_x, \Psi_p)$$

where L_p is the incoming illumination from the photon. When working with photons we need the flux instead of the radiance. Using:

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \mathrm{d}A \cdot cos(N, \Psi)}$$

we get:

$$\Phi(x \to \Theta) = f_r(x, \Psi_p \leftrightarrow \Theta) \Phi_p(x \leftarrow \Psi_p) cos(N_x, \Theta)$$

This equation tells us how much of the flux that spreads to a single direction Θ. When the photon scatters over the hemisphere, the total emitted flux becomes:

$$\int_\Omega \Phi(x \to \Theta)\, \mathrm{d}w_\Theta = \int_\Omega f_r(x, \Psi_p \leftrightarrow \Theta) \Phi_p(x \leftarrow \Psi_p) cos(N_x, \Theta)\, \mathrm{d}w_\Theta$$

This can be approximated using multiple discrete samples:

$$\int_\Omega \Phi(x \to \Theta)\, \mathrm{d}w_\Theta \approx \frac{1}{n} \sum_{i=1}^{n} \frac{f_r(x, \Psi_p \leftrightarrow \Theta_i) \Phi_p(x \leftarrow \Psi_p) cos(N_x, \Theta_i)}{p(\Theta_i)}$$

Since we only want to propagate the photon in a single direction, we use a single sample:

$$\int_{\Omega} \Phi(x \to \Theta)\, \mathrm{d}w_{\Theta} \approx \frac{f_r(x, \Psi_p \leftrightarrow \Theta)\Phi_p(x \leftarrow \Psi_p)cos(N_x, \Theta)}{p(\Theta)} \tag{1}$$

Because direct illumination at each hit point is known, the first bounce of a photon is ignored. For the following bounces, the photon should determine overlapping hit points. Use the BVH to quickly determine overlap candidates. You need to determine both that the photon is within the radius of the hit point, and that the surface normal of the hit point and the surface normal of the photon hit is not facing each other. At each overlap, the photon flux is accumulated into the hit point using the following pseudo code:

`hit.newPhotonCount +=` 1
`hit.totalFlux +=` $\Phi_p \cdot$ `hit.is.mMaterial->evalBRDF (` `hit.is` $, -\Psi_p$)

Once the total flux for a hit point is known, its radiance can be calculated using the following relation:

$$L(x \to \Theta) = \frac{\Phi_{total}}{n_{total}\pi r^2} + L_{direct}(x \to \Theta)$$

where n_{total} is the total number of photons emitted into the scene (all passes), Φ_{total} is the same as `hit.totalFlux`, r is the same as `hit.radius` and $L_{direct}(x \to \Theta)$ is the same as `hit.directIllumination`.

Task 4: *Implement the photon tracing pass.* Trace 100000 photons from the light source in random directions.

Use multiple diffuse bounces, terminated using russian roulette. The diffuse bounces should be implemented by continuing the photon path in a random direction on the hemisphere. Use a uniformly sampled direction, sampled using rejection sampling. Use equation (1) to calculate the photon flux after diffuse scatter.

Task 5: *Repeat the photon tracing pass and output the result of each pass to an image.*

2.3 Radius reduction

In order to guarantee convergence of the algorithm, the radius of each hit point needs to be reduced after each photon tracing pass. For this to work, both the total flux and the number of photons need to be compensated. The following pseudo code takes care of that:

$A =$ `hit.photonCount` + `hit.newPhotonCount`
$B =$ `hit.photonCount` $+ \alpha \cdot$ `hit.newPhotonCount`
`hit.radius` *= $\sqrt{\frac{B}{A}}$
`hit.totalFlux` *= $\frac{B}{A}$
`hit.photonCount` $= B$
`hit.newPhotonCount` $= 0$

Task 6: *Implement radius reduction and repeat the photon tracing passes.* Use $\alpha = 0.7$.

2.4 Cosine weighted sampling

Similar to the previous assignment, about path tracing, we have a geometric term in equation (1).

Task 7: *Use your knowledge to improve the efficiency by implementing cosine weighted sampling.*

2.5 Reflection and refraction

Task 8: *Implement reflection and refraction for the photons using russian roulette.* Use the reflectivity and transparency of the material as probabilities for these events. If none of these events occur, do diffuse scatter.

By letting the photons do reflection and refraction, you will see some beautiful caustics. However, in order for the viewer to actually see the reflections and refractions, the forward tracing pass needs to be extended as well. To this end, perform whitted style reflection and refraction and generate hit points at the intersections with appropriate weights.

Task 9: *Download surface.obj from the course web page and add it to your scene.* Use the following code:

```
Diffuse* water = new Diffuse(Color(0.6f, 0.6f, 0.7f),
                             0.2f, 0.7f, 1.33f);
Mesh* surface = new Mesh("data/surface.obj", water);
surface->setScale(120.0f);
surface->setTranslation(0.0f, 50.0f, 0.0f);
scene->add(surface);
```