Implementing MIS-sampled BRDFs

This is a short guide on how to modify an existing BRDF so that it takes advantage of the multiple importance sampling routines in Arnold. The conversion process fundamentally consists of two parts. The first part is the implementation of BRDF-specific functions, and the second is a series of changes that are required in the light sampling loop. To help illustrate this process, code snippets from the stretched-Phong BRDF of the standard shader will be used as an example.

BRDF-specific functions

For the MIS sampling routines to work you need to implement 3 functions following the signatures defined by the AtBRDFEvalSampleFunc, AtBRDFEvalBrdfFunc and AtBRDFEvalPdfFunc types in the file ai_shader_brdf.h. Below I will describe each of these in turn.

- **AtBRDFEvalSampleFunc** -- The implementation of this function generates a sampling direction given a pair of random variables (rx and ry) with a distribution whose probability density is the one described by the AtBRDFEvalPdfFunc function. The implementation for the stretched-Phong BRDF used in the standard shader is as follows:

```c
static AtVector PhongEvalSample(const void *brdf, float rx, float ry)
{
    const PhongBrdf *phong = (const PhongBrdf*)brdf;
    AtVector omega;
    omega.z = pow(rx, 1 / (phong->pexp + 1));
    float r = sqrtf(1 - SQR(omega.z));
    omega.x = r * cos(ry * AI_PITIMES2);
    omega.y = r * sin(ry * AI_PITIMES2);
    AiV3RotateToFrame(omega, phong->U, phong->V, phong->R);
    if (AiV3Dot(omega, phong->Ng) < 0)
        return AI_V3_ZERO;
    else
        return omega;
}
```

- **AtBRDFEvalBrdfFunc** -- This function returns the result of your BRDF multiplied by the cosine of the angle between the surface normal and the incoming light direction (indir). The implementation in the standard shader is as follows:

```c
static AtColor PhongEvalBrdf(const void *brdf, const AtVector *indir)
{
    const PhongBrdf *phong = (const PhongBrdf*)brdf;
    float reflectance = AiStretchedPhongBRDF(indir, &(phong->View), &(phong->Nf), phong->pexp);
    reflectance *= AiV3Dot(phong->Nf, *indir); // Revise MIS technique. BRDF shouldn't have to do this.
    return AiColor(reflectance);
}
```

- **AtBRDFEvalPdfFunc** -- This function returns the probability density for a given direction. This function must match the density of samples generated by the AtBRDFEvalSampleFunc function, and it is required of every PDF that its integral over the sampling domain must be equal to 1. Its implementation for the stretched-Phong BRDF used in the standard shader is as follows:

```c
static float PhongEvalPdf(const void *brdf, const AtVector *indir)
{
    const PhongBrdf *phong = (const PhongBrdf*)brdf;
    if (AiV3Dot(phong->Ng, *indir) < 0)
        return 0;
    float RL = AiV3Dot(phong->R, *indir);
    RL = MAX(0, RL);
    return (phong->pexp + 1) * powf(RL, phong->pexp) * AI_ONEOVER2PI;
}
```
Note that all three of these functions receive as a parameter a void pointer. Think of it as the equivalent to the `this` pointer in a C++ class or to the `local_data` of an Arnold shader. You can pass any extra data required of your functor implementations through this pointer.

**Light sampling loop modifications**

The second modification that your shaders will require to take advantage of MIS is in the light sampling loop. Instead of directly evaluating your BRDF for each sample returned by `AiLightsGetSample()` you must instead use the `AiEvaluateLightSample()` function, passing it the shader globals, your newly defined functors and a pointer to any local data these functors may require. For example, the resulting stretched-Phong specular light loop in the standard shader looks like this:

```c
PhongBrdf phong_brdf;
PhongBrdfInit(&phong_brdf, sg, clamped_pexp);

AiLightsPrepare(sg);
while (AiLightsGetSample(sg))
{
    if (AiLightGetAffectSpecular(sg->Lp))
    {
        spec_dir += AiEvaluateLightSample(sg, &phong_brdf, PhongEvalSample, PhongEvalBrdf, PhongEvalPdf);
    }
}
```

The `PhongBrdfInit()` function is where the local data required by the functors, `PhongBrdf`, is initialized. As you can see the lights need to be prepared and samples have to be requested just as in a conventional light loop, but instead of making calls to the BRDF and directly weighting your samples, they get weighted through the `AiEvaluateLightSample()` function call using the newly implemented methods.

Make sure that your new sampling loop does not discard light samples that are below the surface normal. This can result in noise or even darkening of the lighting estimate. Even though the light sample is below the surface, part of the light may actually be above the surface in the case of area lights so `AiEvaluateLightSample()` should be called anyway.

**Final notes**

A few things to note is that our current MIS implementation takes an equal number of samples from the light towards the surface as from the surface towards the light and mixes them using the power heuristic. The total number of samples is controlled using the light's "samples" parameter. Also, please note that we are not entirely happy with the current MIS implementation and it may go through some refactoring at some point to simplify its adoption, improve its control and possibly to allow for other importance sampling techniques (resampling importance sampling, one-sample importance sampling, etc).