Understanding GI With a Basic Shader

Plastic Shader with GI

I assume you have read the Writing Your First Shader page, and understand the basics of compilation and installation of shaders. Here we present an old-style "plastic" shader, and then show how this type of shader is properly written to use the features of global illumination.

First a refresher: the plastic shader in traditional scanline renderers uses a combination of 3 terms:

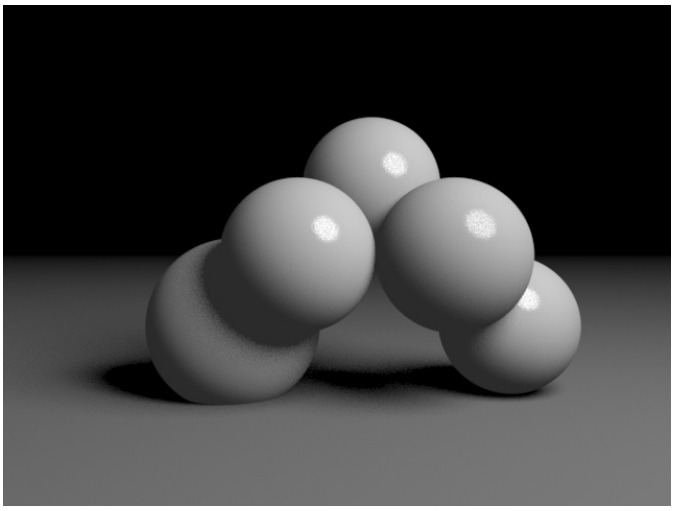
- Specular: the shiny reflected light that bounces off of shiny and glossy surfaces.
- Diffuse: the scattered surface color as lit directly by light sources.
- Ambient: the ambient light bouncing around the scene, not directly from a light source.

You have probably been using these terms for some time, they are probably very familiar. Ambient is simply the user-provided color, added evenly over the surface. Lambertian diffuse is the user-supplied color multiplied by the light color and then multiplied by the amount of light falling on the surface at whatever its angle is to the light (the cosine of the angle from the light to the normal, or L dot N): Wikipedia. And a Blinn-Phong specular is the dot product between the normal and the half-angle from viewer to the light source, raised to a power: Wikipedia.

The shader loops over every light source, accumulating the specular and diffuse components for each light, and then adds the ambient in at the end.

You can see bad speckly noise in the specular highlight from poor sampling; properly written BRDFs can take advantage of Arnold's multiple importance sampling (MIS) to fix this problem, see Implementing Multiple Importance Sampled BRDFs.

Here is an example render that uses these three terms:



And here is the shader code for Arnold that created this image:

```
#include <ai.h>
#include <cstring>

AI_SHADER_NODE_EXPORT_METHODS(SimpleMethods);

enum SimpleParams {
   p_Ka_color,
   p_Kd_color,
   p_Ks_color,
   p_roughness
};

node_parameters
{
   AiParameterRGB("Ka_color", 0.7f, 0.7f, 0.7f);
   AiParameterRGB("Kd_color", 0.7f, 0.7f, 0.7f);
   AiParameterRGB("Ks_color", 0.7f, 0.7f, 0.7f);
   AiParameterFLT("roughness", 0.2f);
}

node_initialize
{
}

node_update
{
}

node_finish
{
}

shader_evaluate
{
   // classic plastic controls are: ambient, diffuse, specular, and roughness.
   // Ka (ambient color), Kd (diffuse color), Ks (specular color), and roughness (scalar)
   AtColor Ka = AiShaderEvalParamRGB(p_Ka_color);
   AtColor Kd = AiShaderEvalParamRGB(p_Kd_color);
   AtColor Ks = AiShaderEvalParamRGB(p_Ks_color);
   float roughness = 10 / AiShaderEvalParamFlt(p_roughness);

   AiLightsPrepare(sg);
   AtColor La = AI_RGB_BLACK; // initialize light accumulator to = 0
   while (AiLightsGetSample(sg)) // loop over the lights
   {
      float LdotN = AiV3Dot(sg->Ld, sg->Nf);
      if (LdotN < 0) LdotN = 0;
      AtVector H = AiV3Normalize(-sg->Rd + sg->Ld);
      float spec = AiV3Dot(sg->Nf, H); // N dot H
      if (spec < 0) spec = 0;
      // Lambertian diffuse
      La += sg->Li * sg->we * LdotN * Kd;
      // Blinn-Phong specular
      La += sg->Li * sg->we * pow(spec, roughness) * Ks;
   }
   // color = accumulated light + ambient
   sg->out.RGB = La + Ka;
}

node_loader
{
   if (i > 0) return false;

   node->methods     = SimpleMethods;
   node->output_type = AI_TYPE_RGB;
   node->name        = "olde_plastic";
   node->node_type   = AI_NODE_SHADER;
   strcpy(node->version, AI_VERSION);
   return true;
}
```

With global illumination, we can improve the image quality, which needs a few changes in how this shader is written.

First of all, we get rid of the ambient term. Most lighting pipelines have avoided using an ambient term for many years now, it often only flattens the image. Global illumination correctly computes the bounced light in the scene, making this old hack unnecessary.

The next concept to grasp is that the energy balance becomes important. Scanline renderers are a bit like 2d-compositing. The three terms are simply added up and mixed together. A global illumination framework is more like photography, bright surfaces reflect light energy more than dark ones, and affect the other objects in the scene. This means that the total energy your shader returns must be less than or equal to the energy in unless it is meant to be a light source. In a global illumination system, light bounces off of multiple surfaces, and the shader is called each time. If your shader returns more energy than goes in, it can amplify the incoming light at every bounce. This feedback loop means even a tiny amount of extra energy can cause a scene to blow out past white very quickly. In this GI plastic shader, we account for this by adding a "specbalance" control. To get a brighter specular, you must take the energy from the diffuse, and vice-versa.

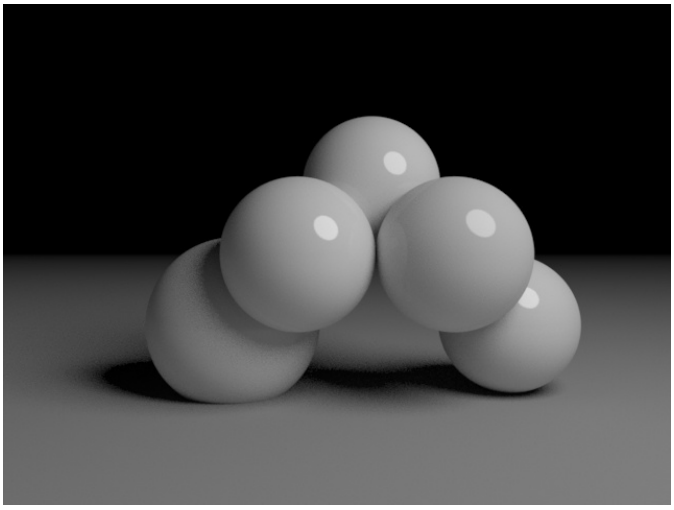
Getting good, energy-balanced specular and diffuse terms can be a bit tricky; fortunately, there are many good BRDFs that ship with Arnold's API, ready to be used. This shader takes advantage of Arnold's multiple importance sampling (MIS) methods, and splits the returned values into 4 terms:

- direct diffuse: old-style diffuse from direct light
- indirect diffuse: GI bounced light as it affects the diffuse surface computation
- direct specular: good energy-balanced specular from direct light
- indirect specular: GI bounced light as it affects the specular surface computation

To allow Arnold to take advantage of the multiple importance sampling technique, the BRDF must first be registered with an **MISCreateData**; the details of this are laid out in the Implementing multiple importance sampled BRDFs documentation.

In the light loop, the direct light and specular are accumulated, and outside the loop, the indirect functions are evaluated. This allows you to pass out the direct and indirect values to an AOV for compositing tweaks while keeping the proper energy balance in the main render.

Here is an example render that uses global illumination (GI) and multiple importance sampling (MIS):



Note that, thanks to MIS, the specular highlight of the big area light renders with much less noise than before. And here is the shader code that created this image:

```
#include <ai.h>
#include <cstring>

AI_SHADER_NODE_EXPORT_METHODS(SimpleMethods);

enum SimpleParams
{
   p_Kd_color,
   p_Ks_color,
   p_roughness,
   p_specbalance
};

node_parameters
{
   AiParameterRGB("Kd_color", 1.f, 0.7f, 0.7f);
   AiParameterRGB("Ks_color", 0.7f, 0.7f, 0.7f);
   AiParameterFLT("roughness", 0.2f);
   AiParameterFLT("specbalance", 0.1f);
}

node_initialize
{
}

node_update
{
}

node_finish
{
}

shader_evaluate
{
   // Kd (diffuse color), Ks (specular color), and roughness (scalar)
   AtColor Kd = AiShaderEvalParamRGB(p_Kd_color);
   AtColor Ks = AiShaderEvalParamRGB(p_Ks_color);
   float roughness = AiShaderEvalParamFlt(p_roughness);
   float specbalance = AiShaderEvalParamFlt(p_specbalance);

   // direct specular and diffuse accumulators,
   // and indirect diffuse and specular accumulators...
   AtColor Dsa,Dda,IDs,IDd;
   Dsa = Dda = IDs = IDd = AI_RGB_BLACK;
   void *spec_data = AiWardDuerMISCreateData(sg, NULL, NULL, roughness, roughness);
   void *diff_data = AiOrenNayarMISCreateData(sg, 0.0f);
   AiLightsPrepare(sg);
   while (AiLightsGetSample(sg)) // loop over the lights to compute direct effects
   {
      // direct specular
      if (AiLightGetAffectSpecular(sg->Lp))
         Dsa += AiEvaluateLightSample(sg, spec_data, AiWardDuerMISSample, AiWardDuerMISBRDF, AiWardDuerMISPDF) *
specbalance;
      // direct diffuse
      if (AiLightGetAffectDiffuse(sg->Lp))
         Dda += AiEvaluateLightSample(sg, diff_data, AiOrenNayarMISSample, AiOrenNayarMISBRDF, AiOrenNayarMISPDF) * (1-
specbalance);
   }
   // indirect specular
   IDs = AiWardDuerIntegrate(&sg->Nf, sg, &sg->dPdu, &sg->dPdv, roughness, roughness) * specbalance;
   // indirect diffuse
   IDd = AiOrenNayarIntegrate(&sg->Nf, sg, 0.0f) * (1-specbalance);

   // add up indirect and direct contributions
   sg->out.RGB = Kd * (Dda + IDd) + Ks * (Dsa + IDs);
}

node_loader
{
   if (i > 0) return false;

   node->methods     = SimpleMethods;
   node->output_type = AI_TYPE_RGB;
   node->name        = "GI_plastic";
   node->node_type   = AI_NODE_SHADER;
   strcpy(node->version, AI_VERSION);
   return true;

}
```

The code, JPEG file, and a Scons SConstruct file are in the attached gzipped tarball.