RENDER SYSTEM

SunFlow

Quickstart
Version 0.07.2

Fpsunflower + Render System
Eugene (etr9j) + Content
Kristoffer Berg + Corporate Design
Tungee + Pdf-Composition

Glossary

INTRODUCTION

Sunflow is an open source rendering system for photo-realistic image synthesis. It is written in Java and built around a flexible ray tracing core and an extensible object-oriented design. It was created as a framework for experimenting with global illumination algorithms and new surface shading models.

Current Features :

* Fully extensible architecture. Almost every aspect of the renderer can be customized.
* User extensible features are exposed as interfaces, making it easy to integrate into existing applications
* Simple API for procedural scene creation
* Direct Illumination with soft shadows from area lights
* Adaptive sampling of area light sources
* Quasi-Monte Carlo sampling architecture: all sampling operations are fully deterministic
* Depth of field
* Camera motion blur
* Multi-threading
* Photon Mapping (for global illumination and caustics)
* Run-time compilation of shaders (via Janino)
* Lazy processing of tesselation, acceleration structure building and texture loading
* Lightmap generation (render to texture)
* Instancing (many copies of the same object take up very little additional memory)
* Bucket renderer
    o Adaptive anti-aliasing (over and under sampling)
    o Bucket based rendering (programmable ordering)
    o Multi-pixel image filtering
* Progressive refinement renderer
* Texture mapping (JPG, PNG, TGA, HDR loaders)
* Bump mapping
* Normal mapping
* Custom scene format: basic translators exist for 3ds files and Blender
* Output formats: HDR, PNG, TGA, OpenEXR (tiled only)
* Written 100% in Java
* Simple Swing GUI with progressive rendering display and console.
* Simple Swing display driver that can be embedded in other applications.
* Primitives
    o Triangle mesh
    o Hair curves
    o Disk
    o Sphere
    o Torus
    o Banchoff surface
    o Bezier patches (built-in teapot and gumbo models) - tesselated on demand
    o Infinite plane
    o Cube-grid (eg: Menger Sponge)
    o Cornell Box
    o Julia Quaternion Fractal

    o Particle Surface (large sets of spheres)
    o Programmable (surfaces can be tesseleated by Java code compiled and executed on demand)
* Cameras lenses
    o Pinhole
    o Spherical (produces a longitude/lattitude environment map
    o Thinlens (produces depth of field effects, including bokeh)
    o Fisheye
* Ray intersection accelerators
    o SAH KD-Tree
    o Bounding Interval Hierarchy (paper)
    o Uniform grid
    o Bounding volume hierarchy
    o Null (for simple scenes)
* Surface shaders
    o Diffuse
    o Mirror
    o Glass (with absorbtion)
    o Phong (with glossy reflections)
    o Ambient occlusion
    o Shiny diffuse
    o Anisotropic ward
    o Wireframe
    o Programmable (in Java - compiled during scene loading)
* Surface modifiers
    o Bump mapping
    o Normal mapping
* Light sources
    o Point light
    o Directional spotlight
    o Triangle area light (paper)
    o Image-based infinite area light
    o Physically based sun/sky system (paper)
* Photon mapping engines
    o KD-Tree
    o Grid based
* Image filters
    o Box
    o Triangle
    o Gaussian
    o Blackman Harris
    o Catmull-Rom
    o Mitchell
    o Lanczos
    o Sinc
* Bucket orderings
    o Hilbert

        o Spiral
        o Row
        o Column
        o Diagonal
        o Random
    * Global illumination engines
        o Irradiance caching
        o “Instant GI” - based on “Illumination in the Presence of Weak Singularities” (Thomas Kollig, Alexander Keller)
        o Path tracing
        o Fake ambient term
        o Ambient occlusion

Contributors:
    * Robert Lindsay
        o Blender exporter
    * Rami Jarakivi
        o TGA input/output
        o OpenEXR display driver
        o Swing GUI Drag and drop support
        o Various bug fixes and optimizations
    * Kristoffer Berg
        o Website design
        o Logo design
    * Artur Brinkmann
        o Documentation

Planned Features
    * Documentation (!)
    * Infinite progressive rendering
    * Shader graphs
    * Improved shader API
    * Dataflow formulation of the rendering pipeline
    * Advanced Swing GUI with support for all rendering options
    * Volume shaders for participating media
    * Geometric and light source motion blur
    * Spectral rendering

INSTALLING SUNFLOW

Requirements:

Java

Sunflow is a Java program, so you need to have Java installed in order to run it. You need Java 1.5 (sometimes called 5.0) or newer - older versions will not work. Versions of Java other than those from Sun are not supported and may not work.

Recommendations

You will get the best performance with Java 6.

Server VM

Sunflow will run considerably faster with the so called server VM (Virtual Machine). This server VM is part of Java, but it's not included in the standard (JRE) installation of Java. The server VM starts up more slowly and uses more memory than the standard VM, but it runs a lot faster in many cases.
Sunflow's built in realtime raytracing benchmark (-rtbench) for example runs more than twice as fast with the server VM (exact speedup depends on your hardware).
The easiest way to get the server VM is by installing the Java Development Kit (JDK). You can get it here: http://java.sun.com/javase/downloads/index.jsp. Download the plain JDK 6, you won't need the version with Java EE or NetBeans.
Note: The 64-bit version of Java already includes the server VM. OS X also ships with the server JVM by default.

Installation:

After downloading the binary version of Sunflow, just unzip the file into a directory of your choice.

Running Sunflow

When running sunflow, you need to pass some parameters to Java, in order to increase performance. For this purpose, it's most convenient to create a batch or script file.

Windows

Creating a batch file for starting Sunflow
Make sure you know where your JDK is installed. By default, this will be something like c:\program files\java\jdk1.6.0.
Now open notepad or the text editor of your choice and paste the following text there:

```
@set javadir="c:\program files\java\jdk1.6.0"
@set mem=1G
@%javadir%\bin\java -Xmx%mem% -server -jar sunflow.jar %*
@if %errorlevel% neq 0 pause
```

ILLUMINATION

Sunflow supports 5 main global illumination (gi) types: Instant GI, Irradiance Caching (aka Final Gathering), Path Tracing, Ambient Occlusion, and Fake Ambient Term. Remember that only one gi type at a time can be used in the scene.

Instant GI

This type of gi is based on this paper (pdf) (page 6 really explains everything in a nice way with pictures). And for me, this is a better way to go than path tracing (below). Hopefully, my understanding of the method is okay enough that I can adequately interpret the settings. If itís not correct, please post so I can correct myself here. What happens is that random points are added to the scene then a ray is traced from a position in the scene to those points to determine the radiance of that position. If rays coming from that point donít meet a particular criteria (set by the bias), a ray is scattered until it does meet the criteria.

Code:
```
gi {
  type igi
  samples 64
  sets 1
  b 0.01
  bias-samples 0
}
```

The samples values is the number of samples (virtual photons) used per set. Increasing the samples gives smoother results, but increases render time.

The number of sets seems to be the number of sets of virtual photons emitted. The more sets, the more points, and the more points added to the calculation, the more noisy less illuminated areas become.

The % bias, or b, is used for the estimate of direct illumination. The short of it is that if this value is too high, youíll see spots of light where the illumination is pooling, but if itís too low youíll lose indirect illumination contributions. This value should be above zero, but usually in the low numbers due the trick. Try starting with 1 and work your way down in big steps.

If the % bias is too high, youíll see spots of light in your scene, most often in the corners or other nooks and crannies. So, you naturally decrease the bias to get rid of them. This then can begin to reduce the amount of light in those corners since you are effectively reducing the amount of rays finding their way to the corners. With bias-samples, you can increase the amount of samples of those bias-diverted rays to get some of that light back into the corners. A value of 0 will not increase the sampling so youíll keep the slightly darker corners, but increasing the value of 1 or greater will give unbiased sampling results. To me, this is one of the key features of the paper, so definitely experiment with this value. Be forewarned, the larger you make this number the slower your scene will render.

Irradiance Caching (Final Gathering)

In a final gather, a hemisphere above a point is sampled by shooting rays at it. These rays bounce off the hemisphere. Secondary bounces are then calculated with either path tracing rays or global photon maps. Here is what the syntax for irradiance caching looks like when using path tracing for secondary bounces:

Code:
```
gi {
   type irr-cache
   samples 512
   tolerance 0.01
   spacing 0.05 5.0
}
```

The samples values is the number of samples (virtual rays) used to calculate the irradiance.

The tolerance option indicates how much error you will allow in the calculation. If the tolerance is set high, the calculation will be faster, but the result will not be as good. If the tolerance is set low, the calculation will be slower, but the result will be better.

The spacing option is used to determine a minimum and maximum distance from the sampled point that will be used to look for other initial ray bounces. The points found within this range will be used to determine what the irradiance intensity should be at the initial point.

Using the above syntax, secondary bounces from the point will be calculated using path tracing, which can be pretty slow. To speed it up, you can instead use a global photon map using the following syntax:

Code:
```
gi {
   type irr-cache
   samples 512
   tolerance 0.01
   spacing 0.05 5.0
 global 1000000 grid 100 0.75
}
```

The only difference is the addition of the global line. The increase in speed from using global photons is based on pre-computing local irradiance values at the many photon positions. In Sunflow, you set the number of global photons you want to use and how you want the photons mapped, either in a grid or a kd tree. The grid is a better way to go, but at least you know the kd option is available. Then you need to define the global estimate and radius (in the example above the estimate is 100 and the radius is 0.75). These values are used at a single secondary bounced photon (of the many photons used). At that point, a sphere with a radius (global radius) expands outward to encompass a certain number of other photons (global estimate) to be used to determine the irradiance at that point. For global estimates, typically 30 to 200 photons are used.

Path Tracing

Probably the most recognized and classic way of doing true global illumination. Thereís not much to say about it except that is probably the most accurate, but slowest, method. It usually gives out a noisy image unless your samples are really high, but if they're high, the image takes forever. Its implementation is very straight forward:

Code:
```
gi {
  type path
  samples 32
}
```

Ambient Occlusion

What more can I say than it's ambient occlusion. The settings are pretty straight forward:

Code:
```
gi {
  type ambocc
  bright { "sRGB nonlinear" 1 1 1 }
  dark { "sRGB nonlinear" 0 0 0 }
  samples 32
  maxdist 3.0
}
```

Fake Ambient Term

Get some quick ambient light in your scene using this one. You can find the reference for this here.

Code:
```
gi {
  type fake
  up 0 1 0
  sky { "sRGB nonlinear" 0 0 0 }
  ground { "sRGB nonlinear" 1 1 1 }
}
```

Overriding

Like shader overrides, you can also override the global photons and global illumination to render only these featureís contribution to the scene (so you can fine tune your settings). To view global photons you would use:

Code:
```
shader {
  name debug_globals
  type view-global
}

override debug_globals false
```

To view the gi in the scene, you would use:

Code:
```
shader {
  name debug_gi
  type view-irradiance
}

override debug_gi false
```

To render the scene normally all you would need to do is comment out the override line:

Code:
```
% override debug_gi false
```

or

Code:
```
% override debug_globals false
```

____

LIGHTNING

Point Light

Code:
```
light {
  type point
  color { "sRGB nonlinear" 1.000 1.000 1.000 }
  power 100.0
  p 1.0 3.0 6.0
}
```

For the point light, power is measured in watts.

Meshlight/Area Light

Code:
```
light {
  type meshlight
  name meshLamp
  emit { "sRGB nonlinear" 1.000 1.000 1.000 }
  radiance 100.0
  samples 16
  points 4
    0.6 0.1 6.0
    0.3 1.0 6.0
    1.0 1.0 5.5
    1.0 0.3 5.5
  triangles 2
    0 1 2
    0 2 3
}
```

For mesh lights, any mesh can become a light. For mesh lights you specify the radiance (watts/sr/m^2), the total power is radiance*area*pi (in watts then if your area was m^2). The example above is a simple square. The important thing to understand is that the light samples are per face (per triangle), so the more complicated the mesh the more it will take to render. See this thread for more information.

The point and area lights are attenuated, meaning that the farther away you go away from the light, the less power/influence on the scene it has. Also remember that you can use the power in negative numbers to remove light from the scene.

NON-ATTENUATED LIGHTS

Spherical Lights

Code:
```
light {
   type spherical
   color { “sRGB nonlinear” 1.000 1.000 1.000 }
   radiance 100.0
   center 5 -1.5 6
   radius 30
   samples 16
}
```

Directional Lights

Code:
```
light {
   type directional
   source 4 1 6
   target 3.5 0.8 5
   radius 23
   emit { “sRGB nonlinear” 1.000 1.000 1.000 }
}
```

INFINITELY FAR AWAY LIGHTS

For IBL/Sunsky power, the exact power would need is measured from the content of the map or the sun position. For IBL it will also depend on a correctly calibrated image.

Image Based Light

Code:
```
light {
   type ibl
   image “C:\mypath\myimage.hdr”
   center 0 -1 0
   up 0 0 1
   lock true
   samples 200
}
```

Sunsky

Code:

```
light {
   type sunsky
   up 0 0 1
   east 0 1 0
   sundir 0.5 0.2 0.8
   turbidity 6.0
   samples 16
}
```

The Sunsky light has a set horizon where the sky stops and the blackness of the world shows up. Normally an infinite plane is the work-around. The next version of Sunflow (0.07.3) will have a control to extend the sky, but you can also modify the source and compile Sunflow yourself so the sky extends on its own. In trunk/src/org/sunflow/core/light/SunSkyLight.java go to the line that says “groundExtendSky = false;” which is line 88 at the time of this writing, and change it to “= true”. Compile Sunflow and the sky will not longer terminate at the horizon.

CAMERA MODELS

Sunflow has four camera types: Pinhole, thinlens, spherical, and fisheye. I'll go into the syntax used for each and a few points that need to be made. As usual I'll update the post if more information is necessary to add.

A main aspects of all the cameras is the eye, target, and up values. Though obvious, I'll go over what they are. Eye is where in the world space the eye of the camera is, the target is where is the worldspace the camera is looking at, and the up value dictates in what direction (in terms of the worldspace) the camera sees as up (basically how the camera is rotated).

Pinhole

Probably what people would consider the "standard" perspective camera.

Code:
```
camera {
  type   pinhole
  eye    7.0 -6.0 5.0
  target 6.0 -5.0 4.0
  up     -0.30 0.30 0.90
  fov    49.134
  aspect 1.333
}
```

The SVN (0.07.3) has a neat feature added to the pinhole, which is camera shifting. It basically takes the perspective shot you have and shifts the view in the x or y without ruining the perspective. I would start with small values for shift. It looks like this:

Code:
```
camera {
  type   pinhole
  eye    7.0 -6.0 5.0
  target 6.0 -5.0 4.0
  up     -0.30 0.30 0.90
  fov    49.134
  aspect 1.333
  shift 0.1 0.2
}
```

Thinlens

The thinlens camera is our depth of field camera, which is also capable of doing bokeh effects. For a thinlens without bokeh you use:

Code:
```
camera {
   type   thinlens
   eye    7.0 -6.0 5.0
   target 6.0 -5.0 4.0
   up     -0.30 0.30 0.90
   fov    49.134
   aspect 1.333
   fdist 30.0
   lensr 1.0
}
```

If you want to use bokeh, you would add two attributes to the end (the number of sides and the rotation of the effect):

Code:
```
camera {
   type   thinlens
   eye    7.0 -6.0 5.0
   target 6.0 -5.0 4.0
   up     -0.30 0.30 0.90
   fov    49.134
   aspect 1.333
   fdist 30.0
   lensr 1.0
   sides 6
   rotation 36.0
}
```

As with the pinhole camera, shifting has been added as an option. It looks like this:

Code:
```
camera {
   type   thinlens
   eye    7.0 -6.0 5.0
   target 6.0 -5.0 4.0
   up     -0.30 0.30 0.90
   fov    49.134
   aspect 1.333
   shift 0.2 0.2
   fdist 30.0
   lensr 1.0
   sides 6
   rotation 36.0
}
```

Depth of field is also one of the two things (the other being motion blur) that are directly affected by samples in the image block. For example:

Code:
```
image {
   resolution 400 225
   aa 0 2
   samples 3
   filter gaussian
}
```

Spherical

The spherical camera produces a longitude/lattitude environment map.

Code:
```
camera {
   type   spherical
   eye    7.0 -6.0 5.0
   target 6.0 -5.0 4.0
   up     -0.30 0.30 0.90
}
```

Fisheye

A classic lens.

Code:

```
camera {
   type   fisheye
   eye    7.0 -6.0 5.0
   target 6.0 -5.0 4.0
   up     -0.30 0.30 0.90
}
```

MATERIALS

NON-TEXTURED SHADERS:

Constant Shader

Code:
```
shader {
  name sfdif.shader
  type diffuse
  diff { “sRGB nonlinear” 0.800 0.800 0.800 }
}
```

Aside from being used as a flat shader, the constant shader can also be used to fake lighting by increasing the value of the color beyond 1.0. It can also be used in conjunction with global illumination to get some interesting results.

Diffuse Shader

Code:
```
shader {
  name sfdif.shader
  type diffuse
  diff { “sRGB nonlinear” 0.800 0.800 0.800 }
}
```

Phong Shader

Code:
```
shader {
  name sfpho.shader
  type phong
  diff { “sRGB nonlinear” 0.800 0.800 0.800 }
  spec { “sRGB nonlinear” 1.0 1.0 1.0 } 50
  samples 4
}
```

The number after the spec color values is the “power” or hardness of the specularity. You can crank it pretty high (e.g. 50000), so start with low values like 50 and work you’re way up or down from there.

Shiny Shader

Code:
```
shader {
  name sfshi.shader
  type shiny
  diff { “sRGB nonlinear” 0.800 0.800 0.800 }
  refl 0.5
}
```

Remember that you can go beyond 1.0 for the reflection value to really kick it up a notch - bam.

Glass Shader

Code:
```
shader {
  name sfgla.shader
  type glass
  eta 1.0
  color { “sRGB nonlinear” 0.800 0.800 0.800 }
  absorbtion.distance 5.0
  absorbtion.color { “sRGB nonlinear” 1.0 1.0 1.0 }
}
```

The “eta” is more commonly known as “ior” or index of refraction. It’s important to note that for the glass shader, the absorption lines are optional. Kirk did some tests using the absorption lines on this thread. And for those savvy readers, I’ll mention that the syntax’s spelling of absorbtion isn’t a typo. This spelling error has been fixed in the SVN so that Sunflow will recognize this spelling and the correct one.

You can also manipulate the trace depths for glass by adding:

Code:
```
trace-depths {
  diff 1
  refl 4
  refr 4
}
```

If you don’t add this, the defaults of diff 1, refl 4, and refr 4 will be used. For a look at various trace-depths in action, see this post by Sielan.

To enable caustics for glass you would add this to the scene:
Code:
```
photons {
  caustics 1000000 kd 100 0.5
}
```

For casutics you set the number of photons you want to use. Currently only kd mapping is allowed. Then you need to define the estimate and radius (in the example above the estimate is 100 and the radius is 0.5). These values are used at a single secondary bounced photon (of the many photons used). At that point, sphere with a radius expands outward to encompass a certain number of other photons (estimate) to be used to determine the caustics at that point. For estimates, typically 30 to 200 photons are used.

Mirror Shader

Code:
```
shader {
  name sfmir.shader
  type mirror
  refl { "sRGB nonlinear" 0.800 0.800 0.800 }
}
```

Ward Shader:

Code:
```
shader {
  name sfwar.shader
  type ward
  diff { "sRGB nonlinear" .80 1 .80 }
  spec { "sRGB nonlinear" 1 1 1 }
  rough .07 .1
  samples 4
}
```

The x and y rough values correspond to u and v tangent directions, so for the ward shader, you'll need to have uv coordinates defined on the object.

Some users of the ward shader are surprised that the result doesn't look like a similar shader in their other favorite application. That look having a metallic surface texture. This shader is a true ward shader, whereas other apps may add other effects to get the metal bump.

Ambient Occlussion Shader

Code:
```
shader {
   name sfamb.shader
   type amb-occ
   bright { “sRGB nonlinear” 0 0 0 }
   dark { “sRGB nonlinear” 1 1 1 }
   samples 32
   dist 3.0
}
```

Wireframe Shader/Normal Shader/UV Shader/ID Shader/Gray Shader/Prims Shader/Quick AO Shader
These shaders are actually accessed via the command line. I go over how to use these shader flags here.

TEXTURED SHADERS:

Not all shaders can use textures, and those that can only use textures for their diffuse channel with the exception of the uber shader. The shaders that can handle textures are diffuse, phong, shiny, ward, ambocc, and uber. The image types that are recognized are tga, png, jpg, bmp, hdr, and igi (in the SVN). For textures, the objects must have UVs mapped. Also note that textures can also be in relative paths:

Code:
```
texture texturepath/mybump.jpg
```

Textured Diffuse Shader

Code:
```
shader {
   name sfdif.shader
   type diffuse
   texture “C:\mypath\image.png”
}
```

Textured Phong Shader

Code:
```
shader {
   name sfpho.shader
   type phong
   texture “C:\mypath\image.png”
   spec { “sRGB nonlinear” 1.0 1.0 1.0 } 50
   samples 4
}
```

The number after the spec color values is the “power” or hardness of the specularity. You can crank it pretty high (e.g. 50000), so start with low values like 50 and work you’re way up or down from there.

Textured Shiny Shader

Code:
```
shader {
  name sfshi.shader
  type shiny
  texture “C:\mypath\image.png”
  refl 0.5
}
```

Remember that you can go beyond 1.0 for the reflection value.

Textured Ward Shader

Code:
```
shader {
  name sfwar.shader
  type ward
  texture “C:\mypath\image.png”
  spec { “sRGB nonlinear” 1 1 1 }
  rough .07 .1
  samples 4
}
```

The x and y rough values correspond to u and v tangent directions.
Some users of the ward shader are surprised that the result doesn’t look like a similar shader in their other favorite application. That look having a metallic surface texture. This shader is a true ward shader, whereas other apps may add other effects to get the metal bump.

Textured Ambient Occlussion Shader:

Code:
```
shader {
  name sfamb.shader
  type amb-occ
  texture “C:\mypath\image.png”
  dark { “sRGB nonlinear” 1.0 1.0 1.0 }
  samples 32
  dist 3.0
}
```

The amb-occ textured shader has been tricky for me in that I’ve never had much success with it. Maybe someone will post to this thread how they’ve used it.

The Uber Shader

Code:
```
shader {
   name sfuber.shader
   type uber
   diff 0.800 0.800 0.800
   diff.texture “C:\mypath\diffimage.png”
   diff.blend 1.0
   spec 1.0 1.0 1.0
   spec.texture “C:\mypath\specimage.png”
   spec.blend 0.1
   glossy .1
   samples 4
}
```

The glossy setting, which gives you the shiny shader aspect, uses very large step values to achieve the result. So a value of zero is shiny, and a value of 1 has no glossyness. When playing with the settings try the following values: 0, 0.001, 0.01, 0.1, and 1. In the uber shader, setting the spec color to 0 and spec.blend to 0 0 0 will allow transparency, though this won’t translate to transparent shadows at the moment.

JANINO SHADERS - A class of their own:

You might ask the question “What is Janino?” Luckily this question was asked. Think of Janino as a Java compiler that only compiles when a janino shader is encountered, so if you write the shader in Java using Sunflow classes, you can come up with creative shaders without having to edit the source code directly. The syntax for this shader looks like this:

Code:
```
shader {
    name myJaninoShader
    type janino
    <code>
    Your code goes here
    </code>
}
```

There are currently three Janino shaders written by forum members at the time of this writing:
Mix Shader
Fresnel Shader
Stained Glass Shader

There is also an old wire frame janino shader running around but Sunflow now has wireframe built in so it's not often used.

Keep in mind that for colors the syntax for sRGB nonlinear color space is used in these examples. These values are what most of us are used to. If you prefer, you can use just values like 1.0 1.0 1.0 instead of { "sRGB nonlinear" 1.0 1.0 1.0 }. You can increase either color space values beyond 1.0 if you really want to saturate the effect.

Sunflow 0.07.2 doesn't have RGBA (0.07.3 will) so here is a work around to get the "A" part. Have your object(s) alone in a scene and have an ambocc shader (in override mode) in the scene have a distance of 0 and have the dark and light colors reversed (black is light, white is dark). Here is what the ambocc shader looks like in the scene file - note that you don't need any object to have this shader and the number of samples is irrelevant:

Code:
```
shader {
  name amboccshader
  type amb-occ2
  bright { "sRGB nonlinear" 0.0 0.0 0.0 }
  dark { "sRGB nonlinear" 1.0 1.0 1.0 }
  samples 1
  dist 0.0
}
```

if you add under this the following line:

Code:
```
override amboccshader true
```

It will cause the whole sceen to use the ambocc shader with a distance of 0, allowing you to create an alpha map - which is really an inverted flat ao pass.

Commenting this last line out will cause the scene to render as usual:

Code:
```
%override amboccshader true
```

SHADER OVERRIDE:

The name of these shaders are variable, so you can name them whatever you like. Names of the shaders are also used in “shader overriding.” You can use any shader listed in your scene file (it doesn’t have to be applied to an object) and with a single line make everything in your scene use that shader. To override, add this line to the scene file:

Code:

```
override shaderName true
```

If you want to scene to then be rendered as usual, simply comment out the line:

Code:

```
%override shaderName true
```

OBJECT GENERATING

There are several ways of generating objects in Sunflow: Primitives (e.g. spheres, hair, cornell box, janino-tesselatable primitives (teapot/gumbo), etc.) - I'm working on describing each of these, direct loading from obj/ra3/stl files, Meshes, and Bezier Patches (SVN 0.07.3 only). I will discuss the latter two here since these are the main bulk of what exporters would use to bring their data into Sunflow.

This information probably isnít as useful as lights, cameras, or shaders since when this data is in the scene file, there is very little you can do to other than change the shader and modifier since there can be thousands of data points. As I noted in the shader thread, this syntax will be getting a facelift in the SVN 0.07.3 and I will update it when itís released. 0.07.3 will still be able to read this format however, so itís still safe to use.

MESHES

Iíll list the general format below, then give examples of each variation. It ís important to understand that in these cases, a point is equal to a vertex.

Code:
```
object {
  shader default
  type generic-mesh
  name meshName
  points X
    x y z
    Ö
  triangles X
    A B C
    Ö
  normals none/vertex/facevarying
  uvs none/vertex/facevarying
}
```

For triangles, it is important to note that the triangle indices are listed using the point numbers starting with 0 as the first point. So if you have three points/vertices defined, the triangle values would look like 0 1 2. Given this way of indexing the triangles based on the number of points, there cannot be a triangle index number above ((the number of points) ñ 1). So in the below examples, you can't have a triangle with an index of 3.

For normals and uvs, you have the option of using none, vertex, or facevarying coordinates. For none, no normals/uvs will be used. The normals and uvs donít have to use the same type to work, so you can have ìnormals vertexî and ìuvs facevaryingî or ìnormals facevaryingî and ìuvs noneî. A key point is that if you are using the vertex type you need the same number of values that there are points. For facevarying, you need the same number of values equal to the number of triangles (see below for examples).

For the vertex type, each point in the mesh will need its own normal or uv coordinate:

Code:
```
...
points 3
   x1 y1 z1
   x2 y2 z2
   x3 y3 z3
triangles 1
   0 1 2
normals vertex
   d1 e1 f1
   d2 e2 f2
   d3 e3 f3
uvs vertex
   u1 v1
   u2 v2
   u3 v3
}
```

For facevarying you would use this format:

Code:
```
...
points 3
   x1 y1 z1
   x2 y2 z2
   x3 y3 z3
triangles 1
   0 1 2
normals facevarying
   d1 e1 f1 d2 e2 f2 d3 e3 f3
uvs facevarying
   u1 v1 u2 v2 u3 v3
}
```

Note that I donít define the number of vertex normals or uvs like I do for the points and triangles since point and triangle numbers will determine how many normal and uv coordinates I will need.

FACE SHADERS

If you want to assign multiple shaders or modifiers to different faces on the same mesh, you can do that like so:

Code:

```
object {
shaders 2
   shaderName0
   shaderName1
modifiers 2
   bumpName0
   “None”
type generic-mesh
name meshName
points 6
   x1 y1 z1
   x2 y2 z2
   x3 y3 z3
   x4 y4 z4
   x5 y5 z5
   x6 y6 z6
triangles 2
   0 1 2
   1 2 3
normals none
uvs facevarying
   u1 v1 u2 v2 u3 v3
   u4 v4 u5 v5 u6 v6
face_shaders
   0
   1
}
```

In the face shader section you are assigning shader 0 (the first shader in the shader list - shaderName0) to the first triangle in the triangle list, shader 1 to the second triangle list, etc. It’s the same with modifiers, but remember that for modifiers with textures (bump/normal map) you need uvs assigned. Also, if you don’t have a modifier for a face, just use “None” in the list.

BEZIER PATCHES

Bezier Patch

Code:

```
object {
shader shaderName
type bezier-mesh
n X Y
wrap false false
points x y z x y z...
}
```

Where n is the number of cv’s in u and v, wrap is equivalent to renderman’s uwrap/vwrap option (optional and defaults to false), and points are the cv data and should be in the same order as in the renderman spec and there should be exactly 3*(u*v) values.

As I noted in the shader thread, this syntax will be getting a facelift in the SVN 0.07.3 and I will update it when itís released. 0.07.3 will still be able to read this format however, so itís still safe to use.

sure if the new file format addresses this. Since we now know how to translate scenes into the new format, I intend to do some tests tonight to see if that has changed. In the mean time, Chris might pop by to tell us more. I’ll post when I know!

Not a big deal:
It seams to me, that in case of this notation:

Code:

```
object {
   shader default
   type generic-mesh
   name meshName
   vertexNumber triangleNumber
   v x y z d e f u v
   t A B C
}
```

I added a big part to this overview: Different shaders and modifiers on different faces of the same mesh. This is a big ommission, so I am posting so everyone is aware I added it. I also removed the depreciated mesh format since the original post should be more concise, but I’ll add it here for the record:

2. DEPRECIATED - Points/Normals/UVs Together

Code:
```
object {
   shader default
   type mesh
   name meshName
   vertexNumber triangleNumber
   v x y z d e f u v
   t A B C
}
```

Where x, y, and z is the vertex coordinate; d, e, and f is the normal coordinate; and u and v is the uv coordinate for the vertex. Itís important to note that each vertex and triangle coordinate must be preceded by a ìvî (for vertices) and a ìtî (for a triangle face). If there are no normals or uvs, zeros can be used.

An example:

Code:
```
...
name myTriangle
   3 1
   v x1 y1 z1 d1 e1 f1 u1 v1
   v x2 y2 z2 d2 e2 f2 u2 v2
   v x3 y3 z3 d3 e3 f3 u3 v3
   t 0 1 2
}
```

Instancing

Instancing is when you use the geometry of another object in a scene without having to duplicate the object's mesh data. Instancing an object is useful in reducing the memory overhead, scene file size, and giving control over the objects since you change the main object and those mesh changes will be propagated to all the instances. Instances in Sunflow are great because you can change the location, rotation, scale, shader, and modifier of each instance. In Sunflow 0.07.2, you would use the following to instance the geometry:

Code:
```
instance {
  name nameOfInstance
  geometry theOriginalObjectName
  transform {
    rotatex -90
    scaleu 1.0
    translate -1.0 3.0 -1.0
  }
  shader shaderForInstance
  modifier modForInstance
}
```

Of course, the modifier line is optional. For transform, you can use or not use a lot of the transform options: rotatex, rotatey, rotatex, scalex, scaley, scalez, scaleu. You could also use a transform matrix like:

Code:
```
instance {
  name nameOfInstance
  geometry theOriginalObject
  transform col my4x4MatrixReadByColumn
  shader shaderForInstance
  modifier modForInstance
}
```

I you are using Blender, the exporter can take dupligrouped objects in the scene and export them out as instances. I go over this in the Blender exporter usage thread.
There are 3 object modifiers in Sunflow. A bump, normal, and perlin. At this time, only one modifier type can be applied to an object at a time.

Bump Map

Code:
```
modifier {
  name bumpName
  type bump
  texture "C:\texturepath\mybump.jpg"
  scale -0.02
}
```

In 0.07.2 the bump scale is set for very small values and actaully is in the negative scale. In the SVN 0.07.3 version of Sunflow, the scale has been fixed making it positive, but also magnifying the effect of the values by a thousand. So a 0.07.2 value of -0.02 would be equivalent to 20 in 0.07.3. This magnification was due to finding that for a bump scale in most images, a value of around 0.001 was usually needed. So in 0.07.3, this value is now the more reasonable 1 value.

Normal Map

Code:
```
modifier {
  name normalName
  type normalmap
  texture “C:\texturepath\mynormal.jpg”
}
```

Perlin Noise (SVN 0.07.3 only)

Code:
```
modifier {
  name perlinName
  type perlin
  function 0
  size 1
  scale 0.5
}
```

Where The Modifiers Go

The modifiers are applied to objects by adding a line to the object (or instance) below the shader declaration. Here is an example of a sphere with a modifier:

Code:
```
object {
   shader default
   modifier modName
   type sphere
   c -30 30 20
   r 20
}
```

Also note that textures can also be in relative paths:

Code:
```
texture texturepath/mybump.jpg
```

COMMAND-LINE EDITING

Sunflow's GUI is great for getting your scene rendered to a png file, but to really unlock Sunflow, there are several command line options available to control the render result. There are several ways to access Sunflow through the command line like editing the sunflow.bat/sunflow.sh file or navigating to the sunflow.jar location to type in the commands. Here is the general look of the comamnd line with a few flags added:

Code:

```
%Sunflow Path%\java -Xmx1G -jar sunflow.jar -quick_uvs -nogui -o "C:\outputlocation\output.png" "C:\scenelocation\test.sc"
```

For completeness sake I will list all the available command line flags but you can use the -help flag to get a list as well:

Code:

```
-resolution W H
-aa MIN MAX
-filter X
-nogui
-o "filename"
-bucket X Y
-threads X
-smallmesh
-nogi
-nocaustics
-ipr
-anim
-bake <object_name>.instance, -bakedir ortho/-bakedir view
-sampler type
-lopri
-hipri
-dumpkd
-buildonly
-showaa
-pathgi X
-bench
-rtbench
-frame X
-X verbosity
-h or -help
-quick_uvs
-quick_normals
-quick_id
-quick_prims
-quick_gray
-quick_wire -aa MIN MAX -filter X
-quick_ambocc X
```

I’ve already gone over a few of these flags:

-anim for animating with Sunflow.
-bucket for changing the bucket order during render.
-threads X for forcing a certain number of threads.
-nogui used to render images without opening the GUI. Required for all images except png files.
-o “filename” used to render images out to a file type.
-bake <object_name>.instance, -bakedir ortho (for diffuse maps), and -bakedir view for lightmap baking.

Here are some quick descriptions of the remaining flags:

-resolution W H : Sets the image resolution, overriding the one set in the scene file.
-aa MIN MAX : You can set the anti-aliasing of the scene in the command line, overriding the one set in the scene file.
-filter X : You can set the filter used for the scene, overriding the one set in the scene file. The available filters are box, gaussian, mitchell, triangle, catmull-rom, blackman-harris, sinc, lanczos, and bspline (in the 0.07.3 SVN).
-ipr : Used to render a scene to the GUI as IPR rather than a final render.
-smallmesh : Load triangle meshes using triangles optimized for memory use.
-nogi : Turns off all global illumination in the scene.
-nocaustics : Turns off any caustics in the scene.
-sampler type : Render using the specified algorithm.
-lopri : Set thread priority to low (default).
-hipri : Set thread priority to high.
-dumpkd : Dump KDTree to an obj file for visualization.
-buildonly : Do not call render method after loading the scene.
-showaa : Display sampling levels per pixel for bucket renderer.
-pathgi X : Use path tracing with n samples to render global illumination.
-bench : Run several built-in scenes for benchmark purposes.
-rtbench : Run realtime ray-tracing benchmark.
-frame X : Set frame number to the specified value.
-X verbosity : Set the verbosity level: 0=none,1=errors,2=warnings,3=info,4=detailed.
-h or -help : Prints all commands.

-quick_uvs : Renders the UVs of objects.
-quick_normals : Renders the normals of objects.
-quick_id : Renders using a unique color for each instance.
-quick_prims : Renders using a unique color for each primitive.
-quick_gray : Renders the all the objects in the scene gray diffuse, overriding all shaders - great for lighting tests.
-quick_wire -aa MIN MAX -filter X : Renders objects as wireframe. You set the aa and filter to be used here because without it, the wire doesn’t look great.
-quick_ambocc X : Renders a scene in ambient occlusion mode with a certain distance (X).

Only one quick_ flag at a time can be used in the command line.

For SVN 0.07.3 only

Code:
-translate
Translate the old file format of the scene file to the new file format in 0.07.3 in either ascii or binary format:
Code:
...sunflow.bat -translate new_scene.sca my_old_scene.sc
Where sca is the file extension for ascii scenes, and scb is for binary scenes (not human readable unless you have a hexadecimal editor handy.

Animation

In order to render an animation in Sunflow you'll have to use a command line, not the GUI. The short of it is using an exporter of your choice to output individual scene files for each frame of your animation and a .java file which parses the information. For an example parsing file, the Blender exporter has this in the .java file (note that Blender exports a settings file that references the other framed files):

Code:
```
public void build() {
   parse("untitled" + ".settings.sc");
   parse("untitled" + "." + getCurrentFrame() + ".sc");
}
```

So what's the command line to send an animation to Sunflow?

Code:
```
-anim 1 10 -o output.#.png scenename.java
```

What this is saying is send frame 1 through 10 and output each frame as a png (or hdr, tga, exr), using the files identified in scenename.java. For us windows users who are using the sunflow.bat, this is how I run it from a windows command line:

```
c:\sunflow\sunflow.bat -anim 1 10 -o output.#.png scene.java
```

OR

add "-anim 1 10 -o outputfiles\output.#.png scenefiles\scene.java" after the sunflow.jar in the .bat file

Notice that in the second example, I have an outputfiles folder and scenefiles folder in the sunflow folder, so you have the ability to call different paths.

This, of course, will render out an image sequence. You'll need to then take this sequence and render it out into a movie (if that's what you want to do) using software that can process said images into movie files. You can use Blender's video sequence editor to do this by adding an image and selecting all the images in the file browser. You can then render out the image sequence.

Also, if you're using the Blender exporter, you can run an animation command from the script by exporting an animation, and with the animation button pressed, select the image type, then hit the Render button. I did a little video on the subject for Blender users. Other users still might find it helpful when I use the commad line to render.