OPath Users Manual

Jamie Clarkson

August 19, 2005

Contents

Chapter 1

Introduction

1.1 What is OPath?

OPath is a physically based rendering system. It takes a scene description written in a simple declarative language (OSM) and runs various ray or path tracing algorithms to generate high dynamic range images.

The purpose of OPath is to build a large rendering system in the Objective Caml language, and also to have fun with rendering algorithms! You may find that it isn't as high performance as other rendering systems, but (low-level) performance is not one of the important aspects of OPath. Rather the focus is on generality, flexibility and supporting advanced rendering features.

1.2 Using OPath

Invoke the opath executable and pass it one or more .osm files:

```
opath scene1.osm [scene2.osm ...]
```

The files are evaluated in order and the environment is kept between them (so symbols defined in scene1.osm are accessible in scene2.osm etc.).

The OSM format (OSM stands for 'OPath Scheme') is a Scheme variant. It isn't a complete implementation by any means although the possibility is there to add functionality as I go along. See chapter 3 for details.

In order to view the images produced by OPath (.hdr radiance compatible by default), you'll need a separate program. Under windows I like to use HDRView: http://www.debevec.org/FiatLux/hdrview/. Under unix systems I've used xview from the radiance package http://radsite.lbl.gov/radiance/.

You might also try http://excamera.com/articles/9/vrpic.html, although I haven't tried it.

1.3 Features

OPath currently supports a variety of features, including but not limited to:

Spectral rendering Rather than the standard RGB component model, OPath uses a full spectral approximation (although this is optional at compile time).

Multiple rendering algorithms Distribution ray tracing, path tracing and Instant Global Illumination are supported. Photon mapping is planned soon.

Physically based materials A variety of BSDFs are supported.

Chapter 2

Installation

2.1 Building from source

The build system is configured for Windows by default. I've tried to make it buildable on Unixy systems (I have actually built and tested on FreeBSD).

2.1.1 Windows

You'll need GNU make, MSVC and OCaml 3.08.3 (earlier versions may work, but untested). You should be able to type 'make' in the src directory and have everything built.

2.1.2 Unixalike

You'll need GNU make and OCaml 3.08.3 (earlier versions may work, but untested). First edit 'src/make.rules' and set the appropriate platform name - just comment out `PLATFORM_NAME=win32` and uncomment `PLATFORM_NAME=unix`. Then just make in the src directory.

There is a slight hitch, in attempting to make things easier I've left the depend files in the src tarball. Unfortunately they were made with the windows version of ocamldep and so the path separators are wrong. You need to do the following in the src directory:

```
make depclean
make depend
make
```

UPDATE: I've been told that certain versions of OCaml 3.08.3 don't support -ffast-math as an option to ocamlopt. This was reported for Gentoo building the OCaml distribution from source. If the build fails you'll have to remove that flag from all the Makefile.mk in each directory (I'm looking into completely revamping the build system).

2.2 Installing binaries

Windows binaries are provided in a separate package. Simply unzip the zip and you're done.

Chapter 3

Understanding OSM

3.1 Overview

OSM is a simplistic Scheme variant (imaginatively 'OPath Scheme'). Currently very little actual Scheme functionality is implemented so it's basically a declarative language for defining the scenes and rendering options.

The best documentation currently is the commented samples in the example directory.

3.2 Tutorial 1

3.2.1 Getting started

First of all, create a text file (we'll use myscene.osm) and open it in your favourite editor. Now, lets create our first geom. Geoms define all of the physical objects in a scene, including lights. For now lets create a sphere and take a look at the OSM code:

```
(make-sphere '(0.0 0.0 0.0) 10.0)
```

This creates a sphere centred at 0.0,0.0,0.0 with a radius of 10 units. As it stands this isn't particularly useful since it will just be created then instantly discarded, so lets give it a name:

```
(define sphere1 (make-sphere '(0.0 0.0 0.0) 10.0))
```

This defines a symbol 'sphere1' which evaluates to our sphere geom.

Next we need a transformation to give this sphere. Since we want to keep this simple we'll just define an identity transform (and bind it to the

symbol 'identity'). Don't worry about this command yet, rest assured we'll come back to it later when we want to instance and move/rotate geoms.

```
(define identity (make-transform '(0.0 0.0 0.0) (make-quaternion 1.0 0.0 0.0
0.0) ) )
```

Ok, now we need a material. Lets create a simple red diffuse material (and again bind it to a name):

```
(define material1 (make-material (make-surf "lambertian" (make-rgb 1.0 0.0 0.0))
) )
```

Whoa! A lot goes on there. Lets break it down a bit.

```
(make-rgb 0.9 0.0 0.0)
```

This function creates a spectrum from R,G,B components. In this case it is a (very) red colour.

```
(make-surf "lambertian" ...)
```

This function creates a surface function, the first parameter "lambertian" says we want a pure diffuse surface. The second parameter is a spectrum or texture which gives the surface a colour.

The final part is make-material. We now need to combine these three things into a primitive which we can actually add to the scene:

```
(define sphere1_prim (make-primitive sphere1 identity material1 ))
```

As you can see make-primitive simply takes a geom, transform and material and returns a primitive (here bound to sphere1_prim).

The next step is to create a light source so we can actually see our sphere:

```
(define light_mtl (make-material (make-surf "lambertian" (make-rgb 0.2 0.9 0.2))
(make-emittance "diffuse" (make-rgb 1.0 1.0 1.0)) ))
(define light1 (make-light (make-sphere '(0.0 75.0 0.0) 2.0) light_mtl identity))
```

This creates a white spherical light source (explained in more detail later).

Now lets build a scene with our sphere and light:

```
(define scene (make-scene '(sphere1_prim) '(light1) ) )
```

The quotes here are very important, they prevent the OSM interpreter from trying to evaluate the following list (in other words, make-scene takes two lists of things). Without the quotes the interpreter would attempt to find a procedure bound to 'sphere1_prim' and since that is bound to a primitive it would raise a nasty exception.

Next we need a rendering algorithm:

```
(define surface-integrator (make-si "direct" scene 4))
```

I won't explain this too much yet, just accept it does direct lighting.
Now we need to create a camera to actually look at the scene:

```
(define view (make-view '(100.0 60.0 0.0) '(0.0 10.0 0.0) '(0.0 1.0 0.0) ) )
(define camera (make-pinhole-fov view 0.035 40.0 40.0 128 128))
```

The make-view function takes three vectors (note the quotes again, vectors are typically represented by lists), an 'eye', a 'focus' and an 'up'. The eye is the position we're looking from, the focus is the point we're looking at and the up vector gives an indication of which way should be up. The make-pinhole-fov call makes a pinhole camera with focal length 0.035, an X FOV of 40 degrees, a Y FOV of 40 degrees and it presumes an image 128x128 pixels in size.

Finally, lets render our scene:

```
(define filename "myscene.hdr")
(save-hdr filename (render surface-integrator camera 128 128 16 scene ) )
```

The render call takes a rendering algorithm, a camera, an image size, a number of samples to take per pixel and a scene.

You can try this out with:

```
opath myscene.osm
```

3.3 Reference

3.3.1 Special forms

define Bind a variable name to a value

quote Do not evaluate the following parameters (usual ' syntax applies too)

3.3.2 Geometry construction/manipulation functions

make-lowmesh Construct a low-polygon (should be under 1,000,000 verts) mesh

load-wavefront-obj Load a mesh from an Alias Wavefront .obj file

make-aabbtree Construct a geom that represents a lowmesh + Axis Aligned Bounding Box Tree acceleration structure

make-sphere Construct a sphere

make-transform Build a transform from a displacement and a quaternion

make-quat-axis-ang Construct a quaternion representing a rotation from an axis and an angle

make-quaternion Directly construct a quaternion

make-primitive Bind a geom, transform and material to form a renderable object (also construct groups of other primitives)

make-light Construct a lightsource from a geom + transform + material (material must be emissive)

3.3.3 Material construction/manipulation functions

make-material Construct a material from a BSDF

make-rgb Construct a spectrum representing the given RGB values

make-spectrum Directly construct a spectrum

make-surf Construct a BSDF of the given type + parameters

make-emittance Construct an emission function of the given type + parameters

load-texture2d Load a 2D texture from a file (currently only TGA files supported)

3.3.4 Scene and rendering commands

make-scene Build a scene from a list of primitives and lights

make-si Build a surface integrator (rendering algorithm)

make-view Build a view (camera transform)

time Evaluate each argument (return value of last one) and time how long it takes

save-hdr Save a Radiance compatible HDR image (first argument is filename)

render Render the scene!