OSL shader write tips

Wang wentao(wwtmac@gmail.com)

Global variables

OSL built in variable call supply useful information while writing shaders, but the built-in variables in our render system differ a little from the original definitions.

Variable	Description
point **P**	Position of the point you are shading. In a displacement shader, changing this variable displaces the surface.
vector **I**	The *incident* ray direction, pointing from the viewing position to the shading position P.
normal **N**	The surface "Shading" normal of the surface at P. Changing N yields bump mapping.
normal **Ng**	The true surface normal at P. This can differ from N; N can be overridden in various ways including bump mapping and user-provided vertex normals, but Ng is always the true surface normal of the facet you are shading.True geometric normal of the surface at P.
float **u, v**	The 2D parametric coordinates of P (on the particular geometric primitive you are shading).
vector **dPdu, dPdv** point **Ps**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ tangent to the surface at P. Position at which the light is being queried (currently only used for light attentuation shaders)
float **time**	Current shutter time for the point being shaded.
float **dtime**	The amount of time covered by this shading sample.
vector **dPdtime**	How the surface position P is moving per unit time.
closure color **Ci**	Incident radiance — a closure representing the color of the light leaving the surface from P in the direction -I.

Table 1: Global variables available inside shaders

- **Shading Normal**
  Normal vector is always face-forwarded and normalized in ER, generally we use the "N" as the shading normal while use it in closures, and we hardly use Ng in the shaders.
- **Coordinates u, v**
  Built-in u v in OSL can be used directly but OSL v is opposite to maya.
- **dPdu and dPdv**
  dPdu and dPdv is used to calculated tangent vector, but attention in ER-OSL implementation it is not normalized.
- **Ci**
  Ci is the final radiance for output, warning, OSL has no Oi variables, while transparency is needed, and it is needed to write like:

```
Ci = result * (1 - o_outTransparency) + cTransparent;
```

Support functions

Function Name	Parameters	Description
void computeSurfaceTransparency	int i_matteOpacityMode float i_matterOpacity color i_transparency output color o_outTranspa	Output the transparency value of the surface according to the maya mode
void computeSurface	color i_surfaceColor color i_transparency int i_matteOpacityMode float i_matteOpacity output color o_outColor output color o_outTrans	The function that counts the surface color according to the matte opacity mode
closure color getReflection	float i_reflectivity, color i_reflectedColor normal Nshading float i_refractiveIndex	Get the reflection closure color of surface
closure color doRefraction	normal Nshading float i_refractions float i_refractiveIndex color i_transparency	Get the refraction closure color of surface
void colorBalance	output color io_outColor output float io_outAlpha float i_alphaIsLuminance float i_alphaGain float i_alphaOffset color i_colorGain color i_colorOffset float i_invert	Compute the color balance and output the color and alpha, same to 3delight.
float filteredpulsetrain	float edge float period float x float dx	filtered pulse train function used in checker for u v filtering

Table 2: Maya utility functions

An example shader

We will take a maya checker shader for example

```
#include "mayautil.h"
shader maya_checker
[[ string help = "Maya checker"]]
(
                // Inputs:
                float   i_alphaGain                 = 1.0,
                int     i_alphaIsLuminance      = 0,
                float   i_alphaOffset           = 0.0,
                color   i_color1                = color(1, 1, 1),
                color    i_color2               = color(0, 0, 0),
                color   i_colorGain                 = color(1, 1, 1),
                color   i_colorOffset           = color(0, 0, 0),
                float   i_contrast              = 1.0,
                color   i_defaultColor          = color(0.5, 0.5, 0.5),
                float   i_filter                = 1.0,
                float   i_filterOffset          = 0.0,
                int     i_invert                = 0,
                vector  i_uvCoord               = vector(0, 0, 0),

                // Outputs:
    output      float   o_outAlpha              = 0,
    output      color   o_outColor              = color(0.0, 0.0, 0.0)
)
```

This is the announcement part, we recommend the formula and alignment like this.

Warning:
When input variable is a Boolean like i_invert in maya, we need to write it as int in the shader, because ER-OSL cannot translate Boolean now.

```
float ss = i_uvCoord[0];
float tt = 1.0 - i_uvCoord[1]; // we invert v in place2dTexture, but this makes the checker different with Maya. so we have to invert v back.

if(ISUVDEFINED(ss, tt))
{
	/* compute 'ss' and 'tt' filter widths.
	    In ER-OSL implemtation, the dx and dy are the differential of screen space
	    and are always 1 */
	/*uniform*/ float dx = 1.0;
	/*uniform*/ float dy = 1.0;
	float dss = abs(Dx(ss) * dx) + abs(Dy(ss) * dy);
	float dtt = abs(Dx(tt) * dx) + abs(Dy(tt) * dy);
```

This part we consider the u v which are imported from place2dTexture node, as the OSL u v is different from maya, we invert v back to make it align with maya.

While computing u v filter width, we need to calculate the differential of ss and tt, in RSL it is write like Du (ss) * du, which means:$\frac{dss}{du} \times du$. In OSL we do not have Du and Dv function, instead we use Dx and Dy function which compute an approximation to the partial derivatives with respect to each of two principal direction, in ER these two directions are screen x and y direction. So we use Dx(ss) * dx instead, however, OSL does not have built-in dx and dy, but we know that in ER-OSL implementation, the dx and dy are the differentials of screen space and are always 1.

```
        ss = mod(ss, WRAPMAX);
        tt = mod(tt, WRAPMAX);

        /*  Add  in  "Effects"  filter  values.  We  multiplie  the
i_filterOffset
           variable by 2 to match Maya's look */
        dss = dss * i_filter + i_filterOffset*2;
        dtt = dtt * i_filter + i_filterOffset*2;

        /* compute separation: 0 for half the squares, 1 for the others.
*/
        float f = 0.5 - 2 *
            (filteredpulsetrain(0.5, 1, ss, dss) - 0.5) *
            (filteredpulsetrain(0.5, 1, tt, dtt) - 0.5);

        /* contrast interpolates the separation from 0.5 to its normal
value. */
        f = 0.5 + (f - 0.5) * i_contrast;

        /* Compute final values. */
        o_outAlpha = 1 - f;
        o_outColor = i_color1 + (i_color2 - i_color1) * f;

        colorBalance(o_outColor,
            o_outAlpha,
            i_alphaIsLuminance,
            i_alphaGain,
            i_alphaOffset,
            i_colorGain,
            i_colorOffset,
            i_invert);
    }
    else
    {
        o_outColor = i_defaultColor;
        o_outAlpha = luminance( i_defaultColor );
    }
```

this part computes the colors and output the texture color, it is easy and has no difference with the 3delight implementation, we need to notice the alpha calculation, we use the built-in luminance function to turn a color to alpha.

Dangerous stuff

- Do not use “return” without value (void return) in OSL shaders, it will cause unexpected behaviors of shaders and hard to debug.

- Don’t use the phrase like:
  `outColor[1] = outColor[0] = outColor[2]`
  It compiles but it does not work at all in OSL.

Work flow

We will talk about our work flow here, after you install our renderer and finished the shader compilation, you can put your oso file in the folder of shaders (generally it is C:\Elara\mtoer\<maya_version>\shaders). Then you can start test you scene in maya:

1. Load ER in maya

Window->Settings/Prefrences->Plug-in Manager:

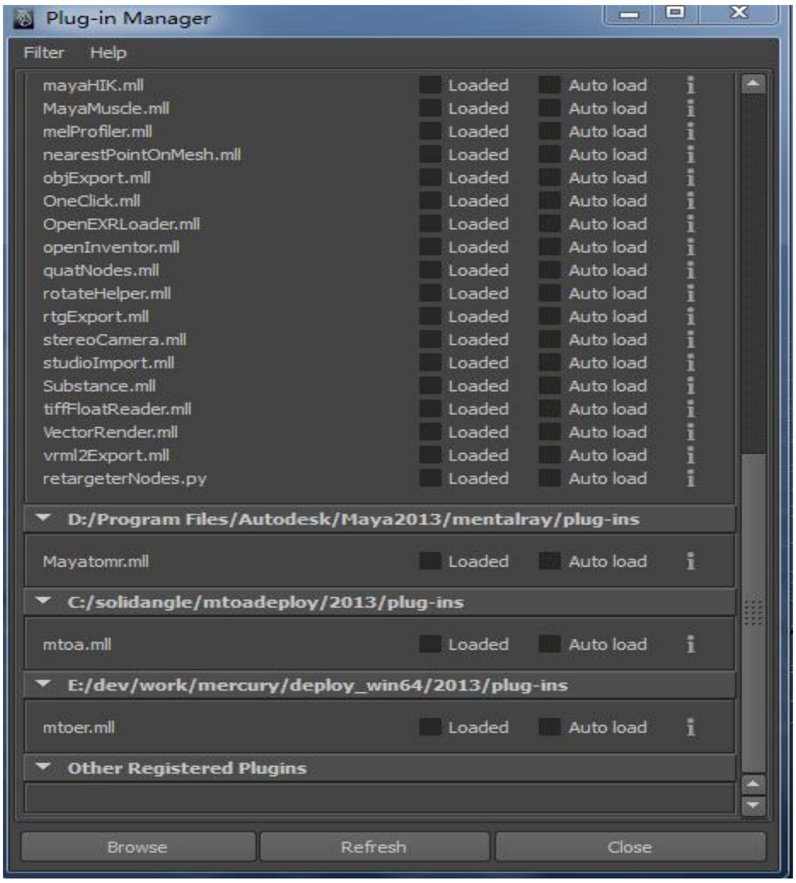


Figure 1. the plug-in manager in maya

Select mtoer.mll->loaded.

2. Change renderer

Open the render setting tab: 

And you will see:

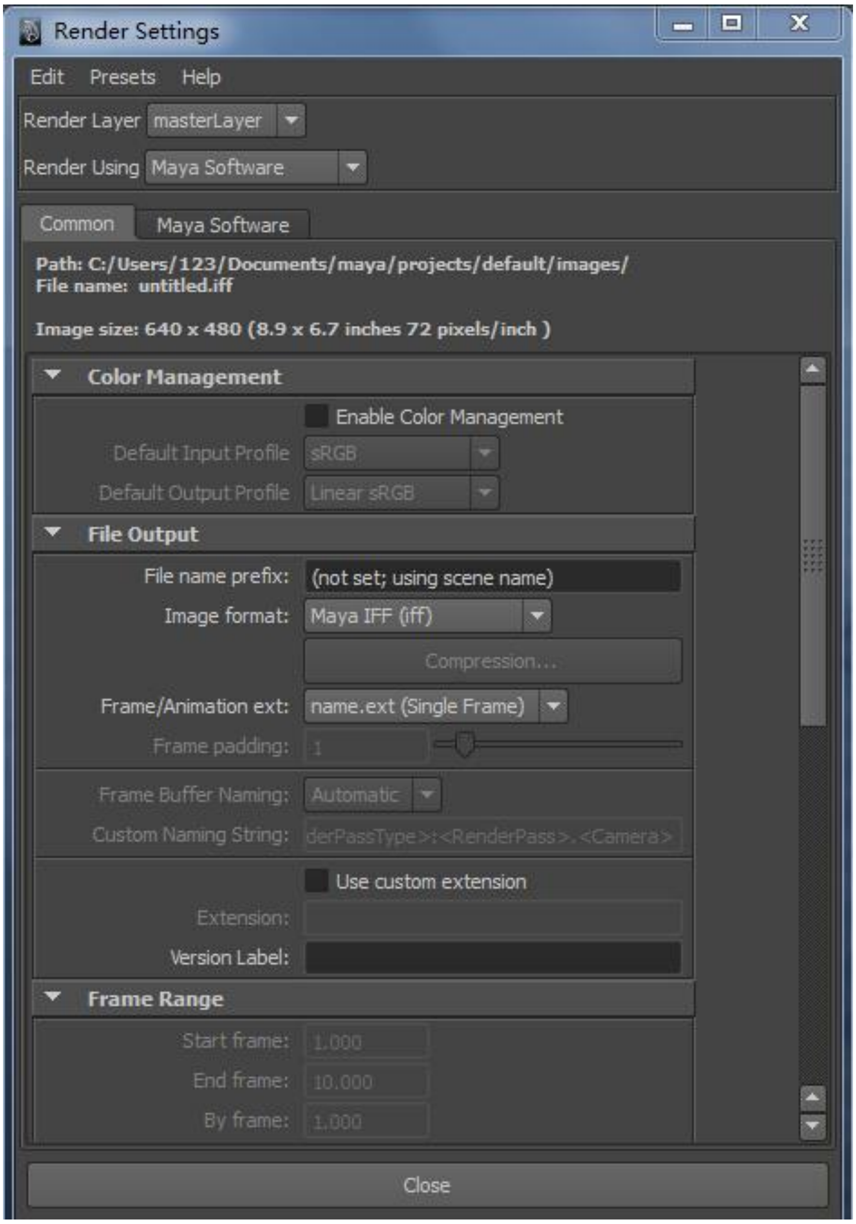


Figure 2. Render settings menu

Change the Render Using menu to “Elara renderer”, then you can start rendering the scene with Elara.

In our side, we will afford the shader translator and an empty oso (which are compiled by empty osl with only the parameters). What you need to do is to finish the implementation of shaders.

How to do the test and build the test case

- Use Maya 2013 x64 version to make the test scene and do the test, when you have finished the scene and want to save it, save it as **.ma**

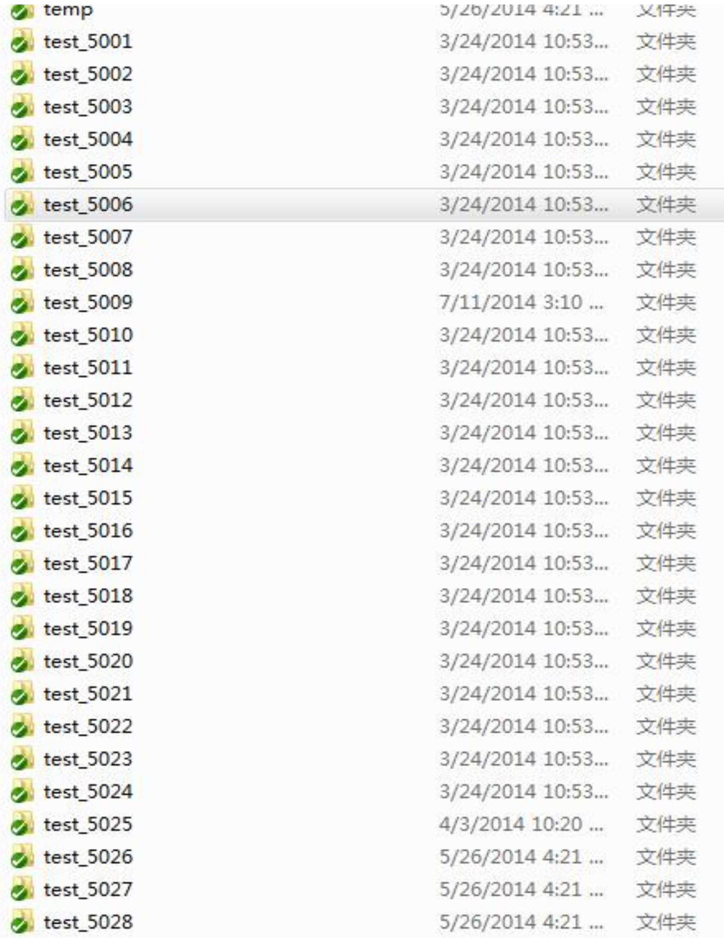
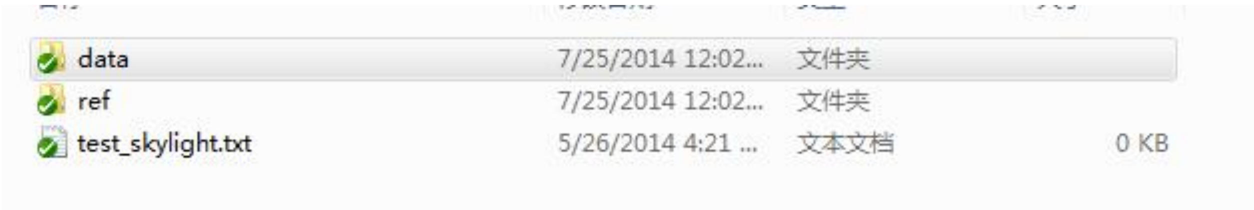


Figure 3. The automation test folder

- Then you need to create a folder called test_xxxx, this folder should contain:



You need to put the scene.ma in the data folder and ref is for the rendered image and exported ess file.

For every parameter which need to be tested (such as filter), build a scene called filter.ma, and reference the filter.ma in test.ma. (You can see how to use maya reference in the internet).

You will also need a test_xxx.txt in which xxx is the test object, this empty txt is just for search convenience.

- Sometimes you need to use texture file, please do not use absolute path in the test scene.

Shader write steps (important)

- Take maya_brownian.osl for an example:
- Shader name and the file name should start with "maya_" , like maya_brownian.osl

- Shader function should has the same name of the shader file name , the return type should be "shader" . Like shader maya_brownian(...){...}

- In mtoer.xml you should add this shader, like

```
<maya_brownian>
  <maya.name type="STRING">brownian</maya.name>
</maya_brownian>
```

maya_brownian is the shader name, while Brownian is the node type in maya.

- Input parameter names should start with i_ , output parameter names should start with o_ .