Medo Effects

Table of Contents

1. Introduction

Medo effects come in 3 flavours:

- - Plugins, which are user modifiable text files using OpenGL Shading Language (GLSL 330)
- - Addons, which are 3rd party binary additions
- - Embedded into the Medo application itself. These effects may transition to Addons in a future Medo release.

Medo Effects, Rev: 01, Published: Dec 2020. Medo is a Haiku Media Editor, written by Zenja Solaja.

2. Plugins

Plugins are user modifiable text files that use OpenGL Shading Language (GLSL 330) fragment shaders to modify rendered pixels.

The bundled Plugins are available in the *system/apps/Medo/Plugins* directory, which are treated as read-only by the system. User customisable Plugins must be stored in *home/config/settings/Medo/Plugins*, in a new subdirectory per plugin.

Eg:
home/config/settings/Plugins/Underwater/Underwater.plugin
home/config/settings/Plugins/Underwater/Underwater.frag
home/config/settings/Plugins/Underwater/icon.png

The **.plugin** file is a JSON text file with information about the plugin. The following elements are mandatory:

plugin
fragment

2.1 plugin

The “plugin” element contains mandatory fields which describe the Plugin.

Table 1: plugin

Field	Data type	Description
plugin/version	int	API version
plugin/vendor	utf-8	Developer name
plugin/type	“colour” “image” “transition” “special”	Submenu where effect appears
plugin/name	utf-8	Plugin name. Internal to Medo application, the combination of “vendor/name” uniquely identifies the plugin in .medo project files.
plugin/labelA	Array of utf-8(*)	Effect name as shown to user. Please keep labelA short.
plugin/labelB	Array of utf-8(*)	Extended effect name, which also appears in tooltip.
plugin/icon	filepath	Icon to display in Effects BOutlineListView. BTranslatorUtils must be able to parse the image format.

(*) - To support multiple user languages, the text labels are an array of utf-8 strings. If the number of elements is less than the number of available languages, the first element (English_British) will be used. The order of language strings are:

Table 2: Array of Languages

Language
English (Britain)
English (USA)
German
French
Italian
Russian
Serbian
Spanish
Dutch
Ukraine

2.2 fragment

The “frament” element describes information about the fragment shader

Table 3: frament

Field	Data type	Description
fragment/source/file	filepath	GLSL Fragment shader source file
fragment/uniforms	Array of uniforms	Uniform variables used by fragment shader. See 2.2.1
fragment/gui	Array of gui elements	Gui elements presented to user. See 2.2.2

2.2.1 uniforms

Fragment shaders use Uniform variables to communicate with the Medo application and accept user customisations from a dynamic GUI. Each uniform variable has a type (from the table below) and a unique name.

Element	Data type	Description
type	utf-8	See table uniforms::type
name	utf-8	GLSL uniform variable

Table 4: uniforms::type

uniforms::type	Consumer	Description
sampler2D	Application	utexture0, uTexture1
float	GUI	GLSL float
vec2, vec3, vec4	GUI	GLSL vec2, vec3 and vec4
int	GUI	GLSL int (can be used as a boolean)
timestamp	Application	Float with elapsed time (from 0.0) in seconds
interval	Application	Float from [0, 1] for interval (inclusive)
resolution	Application	vec2 with project resolution (eg. 1920.0, 1080.0)

The uniforms with an Application consumer are automatically filled by Medo, eg. interval from [0, 1], resolution of project (1920x1080) etc.
The uniforms with a GUI consumer are user modifiable by the dynamic GUI widgets.

2.2.2 gui

All GUI widgets share the following fields:

Table 5: uniforms::gui

Field	Type	Description
gui/type	“slider” “checkbox”	BSlider BCheckbox

	“vec2”, “vec3”, “vec4” “colour” “text”	Spinners (2/3/4) BColorControl with colour picker (*) BStringView
gui/rect	Array of 4 floats	Position/size of GUI widget
gui/label	utf-8	GUI label
gui/uniform	utf-8	Must match a unique name from the uniform table

(*) Only the first colour picker is shown to the user.

The following GUI types have additional information:

Table 6: uniforms::gui::type

gui/type	Field	Description
“slider”	“label_min” “label_max” “default” “range”	Array of utf-8 (per language) Array of utf-8 (per language) Float Array of 2 floats eg. [0.0, 100.0]
“checkbox”	“default”	Int, 0=off, 1 = on
“vec2”	“default” “range” “mouse_down”	Array of 2 floats Array of 2 floats bool, optional, will populate GUI with MouseDown (*) coordinates [0, 1]
“vec3”	“default” “range”	Array of 3 floats Array of 3 floats
“vec4”	“default” “range”	Array of 4 floats Array of 4 floats
“colour”	“default”	Array of 4 floats, [0,1]
“text”	“font”	“be_plain_font” or “be_bold_font”

2.3 Fragment shader

The fragment shader needs to comply with GLSL version 330, which matches OpenGL Core profile 3.3. The Medo application expects the following fragment shader variables:

Medo application expects up to 2 samplers (optional).

uniform sampler2D uTexture0;
uniform sampler2D uTexture1;

The fragment shader receives the following input:

in vec2 vTexCoor0;

The output colour is linked to the following shader variable:

out vec4 fFragColour;

The bundled shaders in *system*/apps/*Medo/Plugins* can be used as references for user customisable shaders. The bundled effects also have shaders souce-embedded into the EffectNode classes.

3. Add-ons

Add-ons are 3rd party binary additions to the Medo application. The bundled add-ons are available in the *system/apps/Medo/Addons* directory, which are treated as read-only by the system. 3rd party add-ons must be installed to *home/config/settings/Medo/Addons*, in a new subdirectory per add-on.

Eg:
home/config/settings/Addons/Underwater/Underwater.so
home/config/settings/Addons/Underwater/icon.png

To compile a binary add-on, the latest Medo source code must be available on the developers system. The Jamfile (or CMakeFile) must specify the target as a Shared Library, and the #include filepath must resolve to the Medo source directory. It is recommended to start by cloning an existing add-on, eg. the "Fade" addon for video effects, or the "Equaliser" addon for audio effects. Once you've rename the derived effect class, you can start developing the new effect.

The new effect is instantiated by the following function which the add-on must supply:

```
extern "C" __declspec(dllexport) MyEffect *instantiate_effect(BRect frame)
{
	return new MyEffect(frame, nullptr);
}
```

All effects must derive from **class EffectNode,** which is speficied in Editor/EffectNode.h

Group	Description
EFFECT_SPATIAL	*void RenderEffect() override* is optional. As a rendering performance optimisation, implement the following methods (1): *bool IsSpatialTransform() override {return true;}* *void ChainedSpatialTransform() override*
EFFECT_COLOUR	*void RenderEffect() override* is optional. As a rendering performance optimisation, implement the following methods (2): *bool IsColourEffect() override {return true;}* *void ChainedColourEffect() override*
EFFECT_IMAGE	*void RenderEffect() override*
EFFECT_TRANSITION	*void RenderEffect() override*
EFFECT_SPECIAL	*void RenderEffect() override*
EFFECT_AUDIO	*void **AudioEffect**() override* (3)
EFFECT_TEXT	*void RenderEffect() override*

(1) The OpenGL Renderer creates a scene graph with all sources and effects per frame. Each scene graph input is rendered to a OpenGL render buffer, and read back as the input for the next effect. As a performance optimisation, EFFECT_SPATIAL effects can skip a RenderBuffer write/read cycle by modifying the yrender::Spatial transformation.
(2) Likewise, EFFECT_COLOUR can skip a write/read render cycle by modifying the fragment shader colour multiplier. This only works for simple effects (eg. Text, Colour effect).
(3) Audio effects are processed **after** the resampler and channel convertor.

3.1 Instantiation

Only a single instance of each EffectNode is instantiated, therefore multiple effects of the same type share the same GUI and rendering elements. Each Project::MediaEffect has two references:

```
class MediaEffect
{
public:
        EffectNode          *mEffectNode;
        void                *mEffectData;
};
```

mEffectNode is a reference to the shared EffectNode.
mEffectData is the per instance storage of unique instance data.

When an EffectNode is selected, it is added as a child to EffectsWindow. Likewise, when it is deselected, it is removed. The method *EffectNode::MediaEffectSelected()* should be used to update the GUI with current effect parameters (if they exist).

3.2 Multithreading

All OpenGL rendering is done from the RenderActor. The OpenGL context is tied to the RenderActor thread (which is locked to it's spawn thread, ie. it isn't a candidate for Work-Stealing by idle threads). The EffectNode GUI is attached to the EffectsWindow BLooper, ie. all BMessages and view rendering is done from a different thread to the RenderActor thread. *Welcome to the BeOS API.* Likewise, the AudioManager runs in a different thread to the EffectsWindow BLooper.

The following table lists which methods are invoked by a corresponding thread:

Method	Thread
EffectNode (constructor) ~EffectNode (destructor) GetEffectGroup GetEffectsListPriority GetVendorName GetEffectName LoadParameters SaveParameters GetIcon GetTextEffectName GetTextA GetTextB CreateMediaEffect	MedoWindow::BLooper
FrameResized MouseDown MediaEffectSelected OutputViewMouseDown OutputViewMouseMoved OutputViewZoomed	EffectWindow::BLooper
InitRenderObjects DestroyRenderObjects RenderEffect IsSpatialTransform ChainedSpatialTransform ChainedMaterialEffect IsSpeedEffect UseSecondaryFrameBuffer IsColourEffect ChainedColourEffect	RenderActor::Actor
AudioEffect	AudioManager::SoundPlayer::thead

3.3 Interaction with MedoWindow::OutputView

An EffectNode can receive normalised MouseDown/MouseMoved() messages from MedoWindow::OutputView. This can be used to reposition an element (eg. Effect/Text) or to control a GUI overlay (eg. Effect/Mask).

> ***Danger** MedoWindow::OutputView runs in a different Blooper to EffectsWindow/EffectNode. You must lock the OutputView looper when interacting with it.*