Getting Started with POVMS

The POVMS Concept

POVMS is a layer below POV-Ray and on top of the host operating system. It abstracts all input and output of POV-Ray. That is, rendering options are set using POVMS, and status messages are generated using POVMS. The abstraction allows complete control of POV-Ray using POVMS even if there is only a network connection between the render engine core and the platform specific GUI. Of course, right now there is no platform that implements such a feature, and the current license does not really allow adding it either!

This documents seeks to get you started by walking you through the basic steps needed to implement a GUI version that exclusively uses POVMS to control POV-Ray and receive output from POV-Ray. This is the recommended way to interact with POV-Ray for POV-ray 3,5 and beyond, because the existing hooks will gradually fade away and will surely no be available in POV-Ray 4.0.

Throughout this document, remember that the core concept behind POVMS is a simple asynchronous message driven communication. POVMS handles all the abstraction for you. It can even deal with different byte orders automatically!

Basic Types

There are a bunch of basic data types you may need to adjust. This list is a quick walk-through:

POVMSType
: POVMSType needs to be 32 bit! (unsigned), it will be composed of four characters, i.e. 'MyTp', 'any ', '4Mac', etc. It defaults to unsigned int.

POVMSInt
: Defaults to int. Whatever the actual type, it has to be a four byte signed integer.

POVMSLong
: Defaults to a special structure representing an eight byte signed integer. If you have some form of native 64-bit data type, you should redefine it as the default again assumes in and unsigned int are four bytes each.

SetPOVMSLong(v,h,l)
: Special function which is used to set a eight byte integer using two four byte integers for the upper and lower four bytes.

GetPOVMSLong(h,l,v)
: Special function which is used to get a two four byte integers for the upper and lower four bytes from the eight byte integer.

POVMSFloat
: Defaults to float, which should offer IEEE 32-bit float precision.

POVMSBool
: Defaults to int. Note that bool is most likely not a suitable type for it. Most likely you want to leave it as is.

POVMSPixel
: Defines a simple RGBA pixel. Defaults to unsigned char[4], which is the most appropriate. Note that as of August 2003 it is not really supported…

POVMSStream
: A single byte. Defaults to unsigned char.

POVMSIEEEFloat
: If your platform does not use the IEEE floating-point data format for its float data type, you will need to provide such a representation.

POVMSFloatToPOVMSIEEEFloat(p, f)
: Converts a native float type into a IEEE float type. Defaults to a simple assignment f = p.

POVMSIEEEFloatToPOVMSFloat(f, p)
: Converts a IEEE float type into a native float type. Defaults to a simple assignment p = f.

POVMS_ASSERT_OUTPUT

As explained, POVMS is a layer below POV-Ray and thus when it fails, i.e. due to lack of memory, it needs to be able to output a message. By default it takes a message string, filename string and line number integer and outputs the message to stderr using fprintf. Of course, this should never happen, but be prepared and override this function if you need a better user interface.

POVMS_LOG_OUTPUT

When debugging, define this function to i.e. write the string it gets to some file. By default this macro simply does nothing.

Message Queue Configuration

Before you can use POVMS, you will need to configure it. This works just like config.h/frame.h for the rest of the POV-Ray code. However, for POVMS you have the option to disconnect povms.cpp from the rest of the POV-Ray code if you define POVMS_DISCONNECTED in config.h. This is extremely useful if you want to place POVMS in a separate shared library that i.e. handles communication between an independent POV-Ray render and GUI process. You can then simply define POVMS_DISCONNECTED on the command-line of you compiler and build the library consisting of povms.cpp and the platform specific code and leave povms.cpp out of the POV-Ray code.

Regardless of the way you use POVMS, if you want more than simple single-threaded communication using POVMS, you will need to override a few methods. You do this the usual way using defines.

Most important for POVMS are message queues. So, if you want i.e. a thread-safe or interprocess message queue, you will have to redefine these:

POVMS_Sys_Queue_Type

The data type of a system specific message queue. Note that this data type needs to support byte-by-byte copies of itself.

POVMSAddress

A POD or POD-only struct that can be send around and uniquely describes a particular POVMS message queue in a particular process and thread.

POVMSAddress POVMS_Sys_QueueToAddress(POVMS_Sys_Queue_Type q)

Returns the POVMSAddress corresponding to a system specific message queue.

POVMS_Sys_Queue_Type POVMS_Sys_AddressToQueue(POVMSAddress a)

Returns the system specific message queue corresponding to a POVMSAddress.

POVMS_Sys_Queue_Type POVMS_Sys_QueueOpen()

Creates a new message queue.

int POVMS_Sys_QueueClose(POVMS_Sys_Queue_Type q)

Destroys a message queue.

void *POVMS_Sys_QueueReceive
(POVMS_Sys_Queue_Type q, int *bytes, bool blocking)

Gets the next message from the specified message queue. Note that the blocking
is optional, that is even if the parameter is true, this function does not have to
block! If you got the message from the queue in some special way (that is, not simply a pointer put on the queue using POVMS_Sys_QueueSend), note that POVMS will use POVMS_Sys_Free to free the memory.

int POVMS_Sys_QueueSend
(POVMS_Sys_Queue_Type q, void *message, int bytes)

Adds a message to the message queue. The message consists of bytes in memory. POVMS_Sys_QueueSend gets ownership of the pointer passed to it and has to make sure it is properly freed using POVMS_Sys_Free, i.e. by simply putting the pointer on the queue and letting POVMS_Sys_QueueReceive free it.

unsigned int POVMS_Sys_Timer()

Returns current time in seconds. It does not have to follow any particular timebase as long as time values return increase monotonously.

Support Function Configuration

These low level functions allow building POVMS without a C library. Usually you do not have to override them:

POVMS_Sys_Strlen(p)
    C strlen function. Defaults to strlen.
POVMS_Sys_Memmove(a, b, c)
    C memmove function. Defaults to memmove.

These function provide independent memory management to POVMS. Remember that the POVMS cannot use the standard POV_MALLOC, POV_CALLOC, POV_REALLOC, POV_FREE calls because it works on a level below those calls (i.e. POVMS needs to be able to report out of memory conditions). Thus, you may want to make sure POVMS can even allocate a basic amount of memory around a few kilobytes if memory is full from the perspective of POV_MALLOC.

POVMS_Sys_Malloc(s)
    C-like malloc function. Defaults to malloc.
POVMS_Sys_Calloc(m,s)
    C-like calloc function. Defaults to calloc.
POVMS_Sys_Realloc(p,s)
    C-like realloc function. Defaults to realloc.
POVMS_Sys_Free(p)
    C-like free function. Defaults to free.

Initializing POVMS

The following assumes you are in a multi-thread environment and the render engine core and the GUI are running in two different threads. It uses the core POVMS concept of a “context”, which is essentially a single POVMS message queue that has a unique address. So, here is the quick guide to create a POVMS-controlled render engine core thread.
In your render engine core thread function, you will need something like this:

```
extern POVMSContext POVMS_Render_Context;
volatile POVMSAddress renderthreadaddress = POVMSInvalidAddress;
volatile bool threadstopflag = false;
void renderthread()
{
        (void)povray_init();
        (void)POVMS_GetContextAddress
                        (POVMS_Render_Context, &renderthreadaddress);
        while(threadstopflag == false)
        {
                povray_cooperate();
        }
        povray_terminate();
}
```

First this code inits POV-Ray. This already creates a valid POVMS context, which is in the povray.cpp global variable POVMS_Render_Context. Thus, to later let the GUI thread communicate with the render thread, the first thing that has to be done is to extract render context. You may want to make sure this really works as expected using just volatile on your platform.
Now, in the GUI you will need to wait until the render thread has started. Thus, you will probably be checking renderthreadaddress if it is valid before sending anything to it. Another thing you will need in the GUI thread is to create another POVMS context such that it will receive output messages. So it will look something like this:

```
POVMSAddress frontendthreadaddress = POVMSInvalidAddress;
POVMSContext frontendcontext;

void init_guipovms()
{
	int err;

	err = POVMS_OpenContext(&frontendcontext);
	if(err == 0)
		err = POVMS_InstallReceiver(frontendcontext, receivehandler,
		 kPOVMsgClass_Miscellaneous, kPOVMSType_WildCard);
	if(err == 0)
		err = POVMS_InstallReceiver(frontendcontext, receivehandler,
		 kPOVMsgClass_RenderOutput, kPOVMSType_WildCard);
	if(err == 0)
		err = POVMS_GetContextAddress(frontendcontext,
		 (POVMSAddress *)(& frontendthreadaddress));

	return err;
}
```

Now, how do you receive messages? Well, this is easy. All you need in your GUI thread is a function that is called periodically. The you simply call:

```
(void)POVMS_ProcessMessages(frontendcontext, false);
```

This was it! Nothing more to do, you will now receive message from the render engine core. But where do the messages end up? Well, in the init_guipovms code above, a message receive callback is installed. It is actually installed twice, for general messages and for all render output messages.
The basic handler could look like the one below. Of course, the individual message data will need to be extracted. There are (in September 2003) going to be functions to conveniently decode messages, going to explained by then…

```
int receivehandler(POVMSObjectPtr msg, POVMSObjectPtr, int)
{
	POVMSAddress addr = POVMSInvalidAddress;
	POVMSAttribute attr;
	POVMSType hid;
	int l = 0;
	int s = 0;
	int ret = 0;
	int line;

	ret = POVMSObject_Get(msg, &attr, kPOVMSMessageIdentID);
	if(ret == 0)
	{
		l = sizeof(POVMSType);
		ret = POVMSAttr_Get(&attr, kPOVMSType_Type,
		 (void *)(&hid), &l);
		(void)POVMSAttr_Delete(&attr);

		(void)POVMSMsg_GetSourceAddress(msg, &addr);
	}
	if(ret == 0)
	{
```

```
                switch(hid)
                {
                        case kPOVMsgIdent_RenderStatus:
                                break;
                        case kPOVMsgIdent_RenderTime:
                                break;
                        case kPOVMsgIdent_Error:
                        case kPOVMsgIdent_FatalError:
                                break;
                        case kPOVMsgIdent_Warning:
                                break;
                        case kPOVMsgIdent_Debug:
                                break;
                        case kPOVMsgIdent_RenderOptions:
                                break;
                        case kPOVMsgIdent_RenderStatistics:
                                break;
                        case kPOVMsgIdent_RenderDone:
                                break;
                        case kPOVMsgIdent_InitInfo:
                                break;
                }
        }

        return ret;
}
```

So, is this all? No, there is one more thing you need to do! The render engine needs to know how to send messages to the GUI, thus, it needs to know its POVMSAddress. This is done by defining FRONTEND_ADDRESS in config.h to a function which extracts the necessary address. So in config.h you could have:

```
#define FRONTEND_ADDRESS get_guiaddress()
```

And then you would need a function like:

```
POVMSAddress get_guiaddress()
{
        return frontendthreadaddress;
}
```

This is sufficient to let the render engine send all messages to the address specified by FRONTEND_ADDRESS. That is it!

Controlling Rendering

Here is the simplest possible way to control rendering. These three functions allow you to set the options that will be used, to start rendering the whole image, and to abort rendering. Note that due to the asynchronous way of communication, stopping may actually take a bit of time. Thus, wait for the render done message in the message receiver callback to keep your GUI consistent. Further, note that the render thread will continue to run even when you are not rendering. You are not supposed to shut it down unless you end the application.

```
void renderstart()
{
        POVMSObject msg;
```

```
	int err = 0;

	err = POVMSObject_New(&msg, kPOVMSType_WildCard);
	if(err == 0)
		err = POVMSMsg_SetupMessage
		 (&msg, kPOVMsgClass_RenderControl, kPOVMsgIdent_RenderAll);
	if(err == 0)
		err = POVMSMsg_SetDestinationAddress(&msg, renderthreadaddress);
	if(err == 0)
		err = POVMS_Send(frontendcontext, &msg, NULL,
		 kPOVMSSendMode_NoReply);

	return err;
}

void renderstop()
{
	POVMSObject msg;
	int err = 0;

	err = POVMSObject_New(&msg, kPOVMSType_WildCard);
	if(err == 0)
		err = POVMSMsg_SetupMessage
		 (&msg, kPOVMsgClass_RenderControl, kPOVMsgIdent_RenderStop);
	if(err == 0)
		err = POVMSMsg_SetDestinationAddress(&msg, renderthreadaddress);
	if(err == 0)
		err = POVMS_Send(frontendcontext, &msg, NULL,
		 kPOVMSSendMode_NoReply);

	return err;
}

void setrenderoptions()
{
	POVMSObject msg;
	int err = 0;

	if(err == 0)
		err = POVMSObject_New(&msg, kPOVMSType_WildCard);
	if(err == 0)
		err = POVMSMsg_SetupMessage
		 (&msg, kPOVMsgClass_RenderControl, kPOVMsgIdent_RenderOptions);
	if(err == 0)
		err = POVMSMsg_SetDestinationAddress(&msg, renderthreadaddress);
	if(err == 0)
		err = POVMSUtil_SetString(&msg, kPOVAttrib_INIFile, inifilename);
	if(err == 0)
		err = POVMSUtil_SetString(&msg, kPOVAttrib_InputFile, inputfilename);
	if(err == 0)
		err = POVMS_Send(frontendcontext, &msg, NULL,
		 kPOVMSSendMode_NoReply);

	return err;
}
```