Odyssey Manual

The Meta-Guide

This guide will act as a manual and tutorial for using the Odyssey Meta-Engine to create a simulation environment. Basically, it will guide one through the process of creating a game engine using Odyssey. Each section will delve into a single area of Odyssey that can be customized to mold your final product.

Throughout this guide there will be a lot of example code and many diagrams. These will be adapted and copied from a current personal project in which I'm crafting an engine from Odyssey. This engine uses the Ogre3D rendering engine for graphics, and it will be the adaptation of Ogre to Odyssey's framework that will be the focus of many of these examples. No knowledge of Ogre's functionality is necessary to understand the examples. All classes from Ogre will be prefixed with the Ogre namespace identifier (Ogre::) while all classes from Odyssey will have no namespace identifier attached (all Odyssey classes are in the Odyssey namespace, but it will be omitted for clarity).

The Model-Facet Connection

The core of the engine's customization lies in the Model and Facet pairs that make up all Elements in Odyssey. A Model holds the data properties for a specific "element concept"[1] and exposes them for reading and writing. Its only job is that of data manager. The Facet is intended to embody the behaviors for the same type of "element concept." When linked together, the Facet may be able to use the Model's properties to control its own behavior. The represents the most clear link between Model and Controller that appears in Odyssey.

Writing a Model

The Model's main purpose is to hold properties. Your Model implementation will be a subclass of Model, and it will define new Properties. The Model subclass needs to register the defined properties in order to have them be exposed in the Model's generic interface. An important note about Models and their Properties is that both remember their dirty state. That is, if you write a property it will be flagged as changed, as well as its parent Model.

Below is an example for the Model that defines properties to adapt the Ogre Entity class. The entity embodies an instance of a mesh in the simulation world of Ogre. Most of the properties are analogies to settings that can be applied to the Entity. Later we'll see how the Model's Properties are handled in the Facet to complete the integration of Ogre's functionality.

```
class EntityModel : public Model
{
public:
      Property<Odyssey::String> Mesh, StaticGroupName;
      Property<bool> NormalizeNormals, Static;
public:
      EntityModel();
};
```

It is a fairly simple class definition. The Property template class from Odyssey is used to define data entries in the Model. As you can see multiple types can be used in properties. The only requirement for a type to be used is that it is assignable (that is, it has an overload for the = operator).

The implementation of this class is equally easy. It can be seen below. Notice that no default constructor is available for the Property class. This is a slight annoyance, but does

[1] See the Odyssey Design Document for an explanation of Model and Facet roles in Odyssey, and how they correspond to viewpoints of representing a simulated world.

not really prove to be a serious issue. This was done to make the Property class as flexible but simple as possible. Forcing a held type to define default constructors would be fair more restrictive than forcing the implementer to explicitly construct the Properties.

```
DECLARE_DYNAMIC2(EntityModel,
                 "Graphics/EntityModel")

EntityModel::EntityModel()
:Mesh(""),
 StaticGroupName("StaticGroup00"),
 NormalizeNormals(false),
 Static(false)
{
      REGISTER_PROPERTY(Mesh);
      REGISTER_PROPERTY(StaticGroupName);
      REGISTER_PROPERTY(NormalizeNormals);
      REGISTER_PROPERTY(Static);
}
```

The first thing you will see is the use of the DECLARE_DYNAMIC2 macro from Odyssey. This marks this class as able to be created dynamic using the functions found in the Odyssey::Dynamic namespace, or better, with the createModel function found in the ModelManager of Odyssey, giving it the string type identifier (in this case, “Graphics/EntityModel”). Dynamic creation is a useful feature and all Models should support it. This means they must have a default constructor, as shown in the example.

Below that are the explicit constructions of the Properties. This is just business as usual. The important part is what appears in the constructor’s body. These are Property registrations, and are vital to the Model. The registration makes it possible for a remote class to access specific properties by name through Odyssey’s PropertyProxy class, without having to know the specific type of the Model subclass. In this example a convenience macro is used. Below is the expansion of the macro for registering the first Property: Mesh.

```
registerProperty(“Mesh”, Mesh);
```

That’s it. The registerProperty function is a protected function from the Model class. The convenience macro simply uses the C++ variable name as the exposed Property name.

Once compiled this Model will be available for use in the application even if we don’t know the specific type of the subclass. All we need to know is the string type ID and the names of the Properties that were registered.

Let’s move on to the use of the above Model implementation. Defining Model data is a fairly simple process. You can create the Model from the ModelManager and immediately begin to access and modify the Model’s Properties. What more, you can modify Model Properties in mid simulation, since the Model support listeners to be

notified and stores its change state. In fact this type of operation is encouraged and is a very effective method for controlling Facet behavior.

```
model = ModelManager::getSingleton().createModel(
                          “Enigma/Graphics/EntityModel”,
                          “Demo01/Box1/Graphics/EntityModel”).lock();
model->getProperty(“Mesh”) = "box.mesh";
model->getProperty(“Static”) = true;
```

First, a new Model is created from the manager. The first parameter is the string type ID and the second is the name. What is returned is a boost weak_ptr which is used in this context (returned from managers) for several lifetime and sharing reasons. You simply lock the weak_ptr to receive the shared_ptr, which is the normal working version of these managed objects[2]. Once the Model has been created, it is available for editing. We can get a PropertyProxy instance from it using the getProperty method. Here we assign two Properties new values. You can see that the PropertyProxy is a generic data element of Odyssey, and can interact with multuple types of different data. Internally, after these assignments, the Mesh and Static properties as well as the overall Model will be marked as changed. You can call the function notifyListeners to send out a notification to anyone who has registered as a listener to this Model. Later we’ll see how Facets, through the linking process, use this facilitiy to interact with the Model and keep the simulation up-to-date.

Writing a Facet

Writing a Facet uses the same basic concepts as writing a Model. Your implementation derives from an Odyssey base class and uses registration to expose behaviors (instead of Properties). The Facet is also intended to link to a (possible several) analogous Models to use their Properties, but this functionality will be shown in the following section.

The Facet implementation can be broken down into two main pieces: the overridden standard behaviors, and the registered message handlers. All Facets can expose certain standard behavior, which you will see in the code fragment below. They also expose their non-standard, dynamic behavior, through handlers for messages sent through the Odyssey Messaging Framework.

[2] For a more detailed overview of managed objects and what they are, see the Odyssey Design Document.

```
class EntityFacet : public MovableObjectFacet
{
protected:
      EntityModel *_model;
      Ogre::Entity *_entity;
public:
      EntityFacet();

      void onCreated();
      void onDestroyed();
      void onActivated();
      void onDeactivated();
      void onUpdate();
protected:
      void onModelUnlink(ModelLinkHandle handle);
protected:
      void onSetMaterial(const Odyssey::Message &msg);
      void onYaw(const Odyssey::Message &msg);
      void onPitch(const Odyssey::Message &msg);
      void onRoll(const Odyssey::Message &msg);
      void onTranslate(const Odyssey::Message &msg);
      void onSetStatic(const Odyssey::Message &msg);
      void onQueryEntity(const Odyssey::Message &msg);
};
```

Above you can see a simplified class declaration for the Facet implementation that complements the previous example's Model declaration. This Facet overrides all the standard functions for a Facet, one of the Model linking overrides, and provides several message handlers. The five standard behaviors overridden are explained below:

- onCreated – called after the create method of the parent Element has been called.
- onDestroyed – called after the destroy method of the parent Element has been called.
- onActivated – called when the parent element is activated.
- onDeactivated – called when the parent element is deactivated.
- onUpdate – this is called after the element or facet has requested an update from the ElementManager, which can happen for various reasons[3].

The onModelUnlink method is called whenever a model is unlinked from the Facet. Similarly, onModelLink is called when a model first is linked.

Each handler shown in this declaration accepts a single argument: the Odyssey Message that is being sent. This is not the only form a handler can take, as you will see farther down in this section.

You will notice that this Facet inherits from MovableObjectFacet. This is simply a convenience base class which itself inherits from Odyssey's Facet class. This is done because many objects being managed from Ogre are derived a single base class: MovableObject. Since so many of these different types of objects share common behaviors, a common facet base class helped reduce code reproduction. The overall

[3] These different reasons will be explained in more detail later.

concepts still apply, but this shows that inheritance hierarchies fit in here just as well as other software systems.

```
DECLARE_DYNAMIC2(EntityFacet, "Graphics/EntityFacet")

EntityFacet::EntityFacet()
:_model(0),_entity(0)
{
	_object = _entity;
	registerHandler("setMaterial",
			boost::bind(&EntityFacet::onSetMaterial, this, _1));
	registerHandler("yaw",
			boost::bind(&EntityFacet::onYaw, this, _1));
	registerHandler("pitch",
			boost::bind(&EntityFacet::onPitch, this, _1));
	registerHandler("roll",
			boost::bind(&EntityFacet::onRoll, this, _1));
	registerHandler("translate",
			boost::bind(&EntityFacet::onTranslate, this, _1));
	registerHandler("queryEntity",
			boost::bind(&EntityFacet::onQueryEntity, this, _1));
}
```

Above is the first part of the Facet implementation. You see the now familiar DECLARE_DYNAMIC statement to allow dynamic Facet creation. The initialization that happens in the constructor is registration of message handlers. These handlers are registered using the boost library's bind function. Each message has an identifying string, which is what is used as the key for this registration. Whenever the "setMaterial" message comes through the Element with this Facet, the onSetMaterial handler will be called.

The rest of the implementation responds to the standard events and messages. The onCreate method will create the internal resources for the Entity. There are no restrictions as to what you include in your implementation. There are a few tools at your disposal, especially for dealing with Models that we'll examine below.

Put it All Together

Now that you have written a Model and a Facet, you need to bridge the gap between data and behavior. We want to control the behavior of our objects by changing the data in the Model at runtime. How do you do this?

The paradigm used is that Models get linked to Facets. A Model remains unaware that it is being "used" and it is left up the Facet to manage this connection (well, the Facet and the user). The Facet has a few tools at its disposal to deal with this connection.

The first thing to remember is that multiple Models may be linked to a single Facet. These Model links are held by the Facet, and may be queried at any time. There are also

events that the Facet may override so that it is notified at the time of linking that a new Model has entered the mix.

```
ModelLinkIterator i = getModelLinkIterator();
while(i.hasMoreElements())
{
	Model *model = i.getNext();

	if(!_model && typeid(*model) == typeid(EntityModel))
		_model = dynamic_cast<EntityModel*>(model);
}
```

This code fragment appears in the onCreate method of the EntityFacet. At this point in the execution, the Facet is just being created, and it is now that we want to first determine what Models have been linked. A ModelLinkIterator can be created to traverse the list of linked Models. RTTI is used in this project to determine if the Model is the correct type and then it is stored as the linked Model. In this case, the EntityFacet will only accept a single Model to be linked: the first EntityModel it finds.

Once the Model has been linked all of its functionality is at the disposal of the Facet. A common task will be to listen for notifications that the Model has been updated. This allows the Facet to only receive updates when the data has changed, saving processing time. The follow code fragment appears in the onActivated function.

```
if(_model)
	_model->addListener(_element);
```

Once activated, this Facet will begin listening for updates. We could have just as easily added the facet as the listener by passing “this” instead of the internally held reference to our parent element. Both Element and Facet derive from ModelListener. Something to remember: once registered we have to do no more work to receive updates. The Element and Facet classes already override the onNotify function and register themselves for update with the ElementManager. The next time the ElementManager is updated (usually every frame) a newly-notified listener will have its onUpdate function called. That is where we will place our handler for Model updates.

```
if(_model->CastShadows.isChanged())
	_entity->setCastShadows(_model->CastShadows);

if(_model->Visible.isChanged())
	_entity->setVisible(_model->Visible);
_model->clearChanged();
```

This appears in the Facet’s onUpdate function. We assume we’ve been notified because the Model has been changed. We make updates into the Ogre Entity’s state from the Model, and then clear the change flag as it has been processed.

This exchange brings up an interesting point. While it is not forbidden by the Odyssey Framework to have multiple Facets share a single Model, it is generally easier if only a

one Facet is linked with each Model. This allows for an ownership relationship in that the Facet can not only pull updates from the Model, but can also update the Model's Properties. For instance, when a physics update moves the physics body, the related Facet can update the Model's "Position" Property to keep up-to-date. When multiple Facets start updating a single Model, things can spiral out of control.

There is another mechanism for handling Model links in the Facet. That comes as the function pair onModelLink and onModelUnlink. These are called upon the linking or unlinking and allow the Facet to handle these events as they happen (instead of waiting for creation time as above). These function are especially useful for handling late linking or unlinking, as in, when Models are added or removed after an Element has already been created.

```
void EntityFacet::onModelUnlink(ModelLinkHandle handle)
{
      if(_model && _model == getLinkedModel(handle))
      {
            _model->removeListener(this);
            _model = 0;
      }
}
```

This is how the EntityFacet handles a model unlinking from it after creation has already happened. It uses an Odyssey class called ModelLinkHandle to refer to the linked Model. After checking that our currently linked model is indeed the one being unlinked, we stop listening and remove the model from our Facet.

To use this system you need a Model and Facet. We will create a Entity system now, showing how to set up all the pieces and how to link them together to get the system working.

```
Model model =
         ModelManager::getSingleton().createModel(
         "Graphics/EntityModel",
         "SkippyModel").lock();
model->getProperty("Mesh") = "skippy.mesh";
model->getProperty("CastShadows") = true;
Element element =
         ElementManager::getSingleton().createElement("Skippy").lock();
element->addDynamicFacet("Graphics/EntityFacet");
element->getFacet(0)->linkModel(model);

element->create();
element->setActive();
```

Above we create the complete system, linking it together so that it is fully functional. We use the facilities of dynamic creation through both the createModel call and the addDynamicFacet function to create our Model and Facet by type identifier. A single call to linkModel does all the linking, including the call to the Facet's onModelLink handler, and finally we create and activate the system.

Message Madness

This section will describe the intricacies of the Odyssey messaging system. This system allows for completely decoupled system to work together seamlessly. With a message-based system behaviors may be invoked across thread boundaries safely, or even across a network as if the call was a local one.

Queues

An essential component in the messaging system is the queuing mechanism. This is represented by the MessageQueue class, which acts as a dispatcher. The MessageQueue class accepts messages to be sent or posted. These operations will eventually lead to the notification of registered listeners that a message has arrived.

```
messageQueue.postMessage(Message("fireWeapon"));
messageQueue.pumpMessages();
```

The above code fragment shows a typical use case of the message queue. When a message is posted, as it is above, it is not immediately process but is stored. When the pumpMessages function is used, all stored messages are processed and sent to listeners for processing. To get immediate processing you can use the sendMessage function. Later, we'll see how that can be used to reserve returned information from message listeners.

Often, a queue can be embedded in the object that will be receiving notifications. This is the case with the Element class, for instance. The Element class contains a MessageQueue as one of its members. The Element derives from MessageListener and upon construction it registers itself to receive notifications from its member MessageQueue[4].

[4] Note here that this does not preclude other listeners. The MessageQueue supports multiple listeners at once.

Listening In

```
class Element : public MessageListener
{
private:
      MessageQueue _queue;
protected:
      virtual void onMessagePosted();
      virtual bool onMessage(const Message &msg);
      virtual void onRemoved(MessageQueue &queue);
public:
      Element();
};
```

The above is a code fragment from the declaration of the Element class, condensed to show only the members related to messaging. As you can see, there are three members to override when inheriting from MessageListener:

onMessagePosted – this is called when a message is posted to the queue
onMessage – this is the function that is supposed to process messages from the queue
onRemoved – this is called to notify the listener it has been removed from the queue and will no longer receive updates

```
Element::Element()
{
      _queue.addListener(this);
}

void Element::onMessagePosted()
{
      _notifyUpdate();
}

bool Element::onMessage(const Message &msg)
{
      return true;
}

void Element::onRemoved(MessageQueue &queue)
{
      queue.addListener(this);
}
```

Above are the trimmed implementations of the MessageListener overrides. Note that the constructor initializes the queue with itself as a listener. Also note that in onRemoved it is added as well. This ensures that the Element object is a listener of its own queue for its entire lifetime, even if an external object removes it.

The two other notifications provide the meat of the Element's response to messaging. In the onMessagePosted event callback, the Element notifies itself as needing an update.

The details of how this is done are not important. The important part of this is that the Element responds to the notification that a new message was just posted to the queue and schedules itself to handle this (by eventually calling pumpMessages). Within the onMessage function the Element must process the message. What it does in here is also not important. It could conceivably be anything.

Use the Message, Luke

Understanding how the Message class works can allow you to invoke all sorts of behaviors from objects. This class lets you invoke generic behaviors, with flexibility in arguments.

To first begin to understand the message, you have to realize that it was intended to model something similar to a function call, while allowing for genericity. Arguments are set and accessed through an array, much like function arguments are ordered. Each message has a name, an ID, and arguments.

```
Message msg(“Test”, var(10));
msg.argument(2, var(10.0f))
      .argument(var<String>(“test”))
      .argument(var(30));
```

Above is a fragment that demonstrates some basic usage of the Message class. Notice how the message is initialized with both its name and an initial argument. Then, the message can receive new arguments. You can see that the first new message is set at a specific index (2). All following arguments are added after this index. This means the argument at index 1 is, of course, empty.

Handle the Truth

Handling messages is a common task within Facets. There are several methods but the easiest to use is done through the MessageAdapter helper class. This class wraps up the interface of MessageListener and allows the programmer to register specific handlers for each message.

Above, you saw handling of message within the Element class. This was done by overriding the onMessage function and directly handling the incoming Message. This is the first, most direct method for handling messages. Below, you can see use of the MessageAdapter class.

```
class MessageHandler : public MessageAdapter
{
public:
      MessageHandler();
protected:
      void onTest(const String &msg);
};
```

This is a declaration for a test message handler. It declares a single handler: onTest. Let's take a look at how you register this handler.

```
MessageHandler::MessageHandler()
{
      registerHandler("test",
            callable<void(const String&)>(
            boost::bind(&MessageHandler::onTest, this, _1)));
}

void MessageHandler::onTest(const String &msg)
{
      cout << msg;
}
```

Registering the handler is done in the constructor. The actual handler function is very simple, and just prints out the message. There are a few key details here: the adapter is able to unwrap the message arguments. The arguments are taken out of the message and then fed into the handler in the correct order. How to use this handler should be apparent. Let's assume this adapter has been constructed and added as a listener to a MessageQueue called queue.

```
queue.sendMessage(Message("test", var<String>("Hello World")));
```

The above will be fed to our adapter, the argument unwrapped, and the correct handler (for "test") called. The message "Hello World" should be printed to cout.