C++ Messaging Systems

Sean Middleditch
DigiPen Game Engine Architecture Club

About Mr. Sean Middleditch

DigiPen BSCSRTIS Senior

2012 Fall GAM200/300 TA

Games: Subsonic, SONAR, Core

Totally hot for game engine architecture

About This Presentation

What "messaging" is

What it is used for

Types of messaging systems

Implementation notes for C++

Certified 100% Picture-Free :(

What It Is

Messaging 101

Messaging Abstract

The purpose of messages and messaging is to allow different components/modules in an application to communicate with each other in a decoupled manner

An object can send a message without knowing who will receive it or what the response will be

Messaging Less Abstract

Messages are objects that are sent from a sender when something in the game occurs to notify other objects of the thing that occurred

Often you have a base Message class that encapsulates data, and then you have a method like HandleMessage on objects that can receive messages

But there's lots of different ways to do it

Types of Messaging Systems

Broadcasting - Send all messages to everyone

Direct Delivery - Send messages to one object

Observers - Register to an object to receive messages

Publisher-Subscriber - Subscribe to a *type* of message and broadcast to subscribers

Message Bubbling

"**Bubbling**" is the means by which messages are sent to children or parent objects

"Bubbling up" means that after an object handles a message, it forwards it to the parent

"Bubbling down" means that after an object handles a message, it forwards it to the children

Message Terminology Confusion

Messages, Events, or Signals?

They're basically synonyms

Common to use more than one in the same app for different parts of the messaging system

"Events send messages when signalled" - whatever terminology you prefer, just be consistent

Messaging Uses

Why bother?

Messages vs Methods

For most simple cases, messages look an awful lot like method calls

In fact, "pure" Object Oriented systems are defined in terms of messages, so the similarity is not coincidental

Then why do we make such a big distinction?

Message Metadata

Messages can carry additional data beyond what a method call normally carries

Messages can know who their sender is

Messages can know when they were sent

Messages can record receiver data for logging

Message Objects

Messages can be encapsulated as their own objects, rather than just being a series of parameters to a method

Message objects can be put in a container and stored (delayed) until later

Message objects can be tunneled over the network

Message Logging

Formalizing the messaging system allows additional code to be injected between the message sender and receiver, unlike with methods

Messages can be logged for debugging

Messages can be modified in transit

Messages can be blocked

Message Routing

Method calls only allow direct dispatch natively

Messages can be routed to multiple receivers automatically

Messages can "bubbled" to parents or children automatically

Messages can be sent to observers automatically

Ideas for Using Messages

Messages are ideal for many parts of a game engine, where decoupling the sender and receiver is valuable

Example 1: A collision message, because any particular component might be interested in knowing about collisions

Ideas for Using Messages

Example 2: Object destroyed messages, so that a locked door in a room can automatically unlock when all enemies are destroyed

Example 3: When the player presses the "use object" button, an Activate message can be sent to any objects in front of the player, without needing to know if the object wants to handle it

Types of Messaging

How you might go about it

Broadcasting

Broadcast systems send a message to every object in the game

Generally a (semi-)recursive operation

Call Engine::BroadcastMessage() which broadcasts to each system and to each game object which broadcasts to each component

Broadcasting

Broadcasting is the second easiest messaging system to implement

It requires a HandleMessage() method on every object that can receive a message

Objects which contain other objects (e.g. an ActionList containing Actions) calls its childrens' HandleMessage() methods

Broadcasting

Broadcasting is the slowest of all messaging systems in general

Requires calling a method for every object that exists for every message that is sent

Objects that don't care about a message will still get "woken up" (pulled into CPU cache) to handle each message

Broadcasting

Broadcasting is essentially when messages are always sent to the root engine object and always bubble down

Broadcasting can hence be implemented as a subset of direct delivery

Direct Delivery

A message is only sent to a single object

Implemented as a single virtual method call

Just call receiver->HandleMessage() and pass it the message object

Can be combined with limited broadcast features (broadcast to children only)

Direct Delivery

The easiest method to implement by far

Can use different methods for different messages rather than needing a message class

receiver->HandleCollision(), receiver->HandleInput(), etc.

Direct Delivery

Least flexible option

No way for an object to observe events that aren't targeted at that object

Really just a generalization of method calls

"Pure" object-oriented paradigm

Observers

Each object has a list of interested objects to which it will send messages

Observer objects register with the source object

Call this->SendToObservers() to send the message

Call source->RegisterObserver(this) to observe

Observers

Extremely fast... if implemented well

Requires more memory manager overhead than most other techniques

Generally implemented as a linked list of "**binding**" objects, may also just be std::vector's of objects

Observers

Requires two interfaces

Observer interface has HandleMessage()

Observable interface has RegisterObserver() and SendToObservers(), and also UnregisterObserver()

Observers

Observer objects should automatically unregister themselves when destroyed

Observable objects should differentiate between different types of events

- InputManager might send KeyUp, KeyDown, MouseMoved, etc. and each can be registered for individually

Observers

Observers are fairly fast

Most overhead is in iterating over the list of observers (they might not be in CPU cache)

Only the objects that are interested in an event are notified about it

Publisher-Subscriber

Objects can send events without knowing or caring who is listening for them, and objects can register to listen to a *type* of event

Different from observers because objects register for event types, not event sources

Completely decouples event sources and receivers

Publisher-Subscriber

Very scalable for large systems (thousand of servers in an Internet application cluster)

Not frequently used inside a game engine

Sometimes confused with observers, which is the only reason I'm bringing it up

Final Thoughts on Messaging Types

Three of the types of messaging systems mentioned are useful inside a game engine: Broadcasting, Direct Delivery, and Observers

Very large, complex engines often support all three, since they're useful in different contexts!

Most game engines support at least two

Final Thoughts on Messaging Types

Recommendation

- Small games should use broadcasting and direct delivery
- Medium and up games should use direct delivery and observers
- Support bubbling down

C++ Message Objects

Encapsulating message data

Message Object

The message object needs to contain all the necessary data to handle the object

- Message Type
- Message Sender
- Message Payload

Message Object

The message type can easily just be an enum, or even a string (for smaller/simpler games... probably including yours)

```
enum MessageType
{
  MSG_UNKNOWN = 0,
  MSG_KEY_UP,
  MSG_KEY_DOWN,
  MSG_MOUSE_MOVED,
  MSG_COLLISION,
  MSG_OBJECT_DESTROYED,
  // etc
};
```

Message Payload

The payload can be trickier

Generic options are possible, allowing you to stuff any values you want into any message

This is a "dynamic" approach, reminiscent of Lua/Python/JavaScript/etc.

Message Payload

The generic approach is not C++ friendly

It requires more work to code

It is slower

It is error prone

Message Payload

Message subclasses work much better

```
class Message {
public:
  MessageType m_Type;
};

class KeyMessage : public Message {
public:
  KeyCode m_KeyCode;
};

class MouseMessage : public Message {
public:
  vec2 m_Position;
};
```

Message Payload

Subclasses in this case are a good use of inheritance, not to be confused with many other abuses of inheritance

Prefer "heavy" objects, e.g. a single object for all keyboard messages, versus different objects for KeyUp and KeyDown and KeyRepeat

Even consider one object for all input messages

Message Dispatch

Objects that handle messages can have a simple HandleMessage method

```
virtual void HandleMessage(const Message& msg)
{
  switch (msg.m_Type) {
  case MSG_KEY_DOWN:
  {
    const KeyMessage& key_msg =
      static_cast<const KeyMessage&>(msg);
    // handle key message here
    break;
  }
}
```

Message Dispatch

The disadvantage is all that ugly casting and the switch statement

Neither are necessarily inefficient, but they are error prone

Macros can hide the details, but still aren't that pretty

Message Dispatch

An option is to have a different HandleMessage() overload for each type of message

We would select which method to call based not only on the receiver (single dispatch via virtual method table) but also on the argument type ("**double dispatch**")

Message Double Dispatch

C++ does not natively support double dispatch

Requires some inventiveness to implement

Usually done via a "thicker" interface that defines all possible HandleMessage overloads and a method on the Message object that selects the proper overload

Message Double Dispatch

All classes that handle messages would derive from this interface:

```
class IMessageHandler {
public:
  virtual void HandleMessage(const Message& msg) {}
  virtual void HandleMessage(const KeyMessage& msg) {}
  virtual void HandleMessage(const MouseMessage& msg) {}
  // etc.
};
```

Message Double Dispatch

The Message class is extended with a new virtual method that all children override (just showing new methods here)

```
class Message {
public:
  virtual void Dispatch(IMessageHandler& handler) const =
0;
};

class KeyMessage {
public:
  virtual void Dispatch(IMessageHandler& handler) const
  {
    handler.HandleMessage(*this);
  }
};
```

Message Double Dispatch

Sending a message then looks like this:

```
GameObject* receiver_object = blah;

// send the "up arrow pressed" message
KeyMessage msg;
msg.m_Type = MSG_KEY_DOWN;
msg.m_KeyCode = KEY_CODE_UP_ARROW;
msg.Dispatch(*receiver_object);
```

Message Double Dispatch

The virtual Dispatch method is called, which invokes the version on KeyMessage

This method calls the receiver's HandleMessage with parameter **this* which is of type *const KeyMessage&*

The best overload available on IMessageReceiver is *HandleMessage(const KeyMessage& msg)*

Message Double Dispatch

Note that you don't necessarily need to make the Dispatch method a virtual method if you don't want to

Just be sure to always call it on the specific Message subclass you're using, not on a base class pointer/reference (if you're unsure what that means, ask)

Bubbling (in C++)

Sorry kids, this has nothing to do with an Mormon PhotoShop tricks

Bubbling

Bubbling is a core part of broadcasting and also is used for making direct delivery more useful

Implementable in two ways: simple and generic

Simple Bubbling

We extend HandleMessage to forward the message to the parent or children

Works with any version of message dispatch

We must put the code to handle bubbling in each object's HandleMessage, which can be a bit redundant (generic implementation handles that)

Bubble Down

```
virtual void HandleMessage(const Message& msg)
{
  // code to handle message goes here
  ...

  // bubble down (C+11 "pseudocode")
  foreach (auto child : m_Children)
    child->HandleMessage(msg);
}
```

Bubble Up

```
virtual void HandleMessage(const Message& msg)
{
  // code to handle message goes here
  ...

  // bubble up
  if (m_Parent != null)
    m_Parent->HandleMessage(msg);
}
```

Generic Bubbling

The generic bubbling method works by adding a few new fields and methods

We'd like to be able to specify on each message what bubble direction it would like to use, rather than hard-coding that into the HandleMessage method

We can then just use an if-condition to select which bubbling code to use

Generic Bubbling

Ideally, we wouldn't put this if-condition inside of HandleMessage, but rather we'd put it inside of a the message dispatch code

Our IMessageHandler interfaces needs some extensions, however

Generic Bubbling

```
class IMessageHandler
{
public:
  virtual void HandleMessage(const Message& msg) = 0;
  virtual IMessageHandler* GetParent() const = 0;
  virtual void SendToChildren(const Message& msg) = 0;
};
```

Note that we added GetParent() and SendToChildren()

Generic Bubbling

The GetParent() method should return an object's parent

- For a Component, this would be its GameObject
- For a GameObject, this might be its Space (or the GameObjectFactory system, or so on)

Generic Bubbling

The SendToChildren method should call msg.Dispatch() for each child in the object

For a GameObject, its children are its Components

For the Engine, its children would Systems

For the game object system, its children are all the GameObjects

Generic Bubbling

Add a new public non-virtual method Send to Message (Dispatch becomes protected):

```
void Message::Send(IMessageHandler& handler) const
{
  // dispatch to original object
  Dispatch(handler);

  // dispatch to parent if we bubble up
  if (m_Bubble == BUBBLE_UP && handler.GetParent() !=
NULL)
    Dispatch(*handler.GetParent());

  // dispatch to children if we bubble down
  if (m_Bubble == BUBBLE_DOWN)
    handle.SendToChildren(*this);
}
```

Observers in C++

Keeping tabs on the neighbors

Observers in C++

There are a number of different ways to implement the observer pattern in C++

I present one simple way to do it, but not necessarily the best way to do it

Observers in C++

Ways to go about supporting observable objects

- Observable for all messages it can send, via a base class
- Explicit members to the object for each message, like C# events

Observers in C++

I will go over the first method, as it is slightly simpler and more memory efficient (if slightly less CPU time efficient when implemented the easy way)

We will need to create a base class to support objects that can observe or be observed

We will also need a way to bind methods

Code Samples Online

The code required does not fit into slides very well

Available on my GitHub: https://github.com/seanmiddleditch/dpengineclub

Link to code: http://tinyurl.com/96s8pl4

MessagingBase Class

Primary use case is to allow *unbinding*

When Observer A is destroyed, we want to unbind from any objects it is observing, to avoid crashes by accessing deallocated objects

Need the same thing for when an observed object is destroyed

MessagingBase Class

The base class *MessagingBase* includes a simple list of all bindings

Includes both objects it is observing and objects observing it

No, this is not at all the ideal or most efficient implementation, it's for easy illustrative purposes only

MessagingBase Class

When the object is destroyed, the destructor automatically removes any bindings

Bindings can also be removed explicitly

Obviously, also includes a means to add a binding

MessagingBase Class

```
class MessagingBase {
public:
  std::vector<Binding> m_Bindings;

  ~MessagingBase() {
    // for each binding in m_Bindings {
    //   remove binding from other object
    // }
  }
}
```

Sample code in the links given... also, in a very rare case, std::vector is actually not the ideal data structure here, but it's the simplest to use

Simple Observers

The simplest way to implement observers is with a single *HandleMessage* method

```
class Message {
public:
  MessageType m_Type;
  IMessageHandler* m_Recipient;
};

void MyClass::HandleMessage(const Message& msg) {
  if (msg.m_Recipient != this)
    ... // message I intercepted as observer
  else
    ... // message sent directly to me
}
```

Simple Observers

```
struct Binding
{
  MessageType type;
  Observer* observer;
  MessagingBase* observed;
};

void MessageBase::BindObserver(MessageType type,
  Observer* observer)
{
  Binding binding;
  binding.type = type;
  binding.observer = observer;
  binding.observed = this;

  m_Bindings.push_back(binding);
  observer->m_Bindings.push_back(binding);
}
```

Delegates in C++

The easy version of observers requires all observer objects to have a single virtual method

```
class Observer {
public:
  virtual void OnObserveMessage(const Message& msg) = 0;
};
```

This is not particularly flexible, however

Delegates in C++

C++ lacks easy support for doing things better

Method pointers cannot easily be used, and the naive hacks are all thwarted by compiler discrepancies

sizeof(void(MyObject::*Method)()) varies from 4 to 20, so method pointers are not convertable to void*

Delegates in C++

Ideal solution gets us as close to having "void* for member functions" as possible

- Least memory usage possible is 8 bytes (32-bit platforms): pointer to object and pointer to method
- Least CPU override possible is three x86 ASM instructions

"Impossible" to achieve in portable C++ (almost)

Delegates in C++

Several solutions exist, all sub-par

std::function/boost::function - portable bu slow

"Impossibly Fast C++ Delegates" - ugly syntax

"Fast C++ Delegate" - non-trivial implementation

Delegates in C++

Syntax to use Impossibly Fast C++ Delegates is pretty bad (some clever will help a lot)

```
class MyClass {
public:
  void MyFunction(int a, float b);
};

typedef delegate<void(int, float)> CallbackFn;

MyClass foo;
CallbackFn callback =
  CallbackFn::make_delegate<MyClass,
  MyClass::MyFunction>(&foo);
```

Observer Delegates

Even this solution still locks us in to a specific method parameter list

We want to be able to bind to any method that takes any version of *Message*

The Impossibly Fast C++ Delegates implementation gives us a normal C function pointer that we can cast as needed

Observer Delegates

The stub function from the Impossibly Fast C++ Delegates technique, simplified:

```
template <typename Class, typename Param,
  void(Class::*Method)(const Param&)>
void bind_method(void* instance, const Message& msg)
{
  (static_cast<Class*>(instance)->*Method)
    (static_cast<const Param&>(msg));
}
```

If that makes you nervous, that's a good sign :)

Observer Delegates

We can now store just two pointers for any observer

- Pointer to the observer itself (as base class Observer*)
- Pointer to template-generated stub function for desired method

Observer Delegates

```
class MyObserver : public Observer
{
public:
  void ObserveKeyMessage(const KeyMessage& msg);
};

struct Binding
{
  Observer* observer;
  Observer::Callback callback;
};

Binding binding;
binding.observer = my_observer;
binding.callback = &bind_method<MyObserver, KeyMessage,
  &MyObserver::ObserverKeyMessage>;
```

Observer Delegates

You can wrap up the code to create the *Binding* object and register it with both the *Observer* and the object being observed

Again, macros can make it a little easier to use

Do-It-Yourself Tips

Ways to improve and expand on this pattern

Cleaner Observers

The observer implementation given here could use some cleanups

The example code online ***is not meant to be copied***

Use it as a working example and make your own improved version (no std::vector, cleaner API, etc.)

Use Metadata for Messaging

You can use metadata to list which messages an observable object will emit

You can use metadata to get a list of all bindable methods an observer has

You can combine these two to make GUIs or text files to easily bind a component's methods to other objects' events for designers

Observable Dispatch

You can combine observers and direct dispatch systems

Instead of calling a method directly on another object, use a mediator class that also signals any bound observers

Keep it Simple

Use the simplest messaging system that works for your needs

There are far better things to spend your time on than fancy messaging systems you don't need

Making a good game is your priority, not fancy technology

Questions?

An illuminating flashlight shines brightest on whatever it's pointed at