Messages, Events, and Communication

What are messages?

- A way of communicating between systems or objects that does not rely on directly linked functions.
- Decouple the requester of the action from the object that performs the action.
- For example: Windows Message Loop

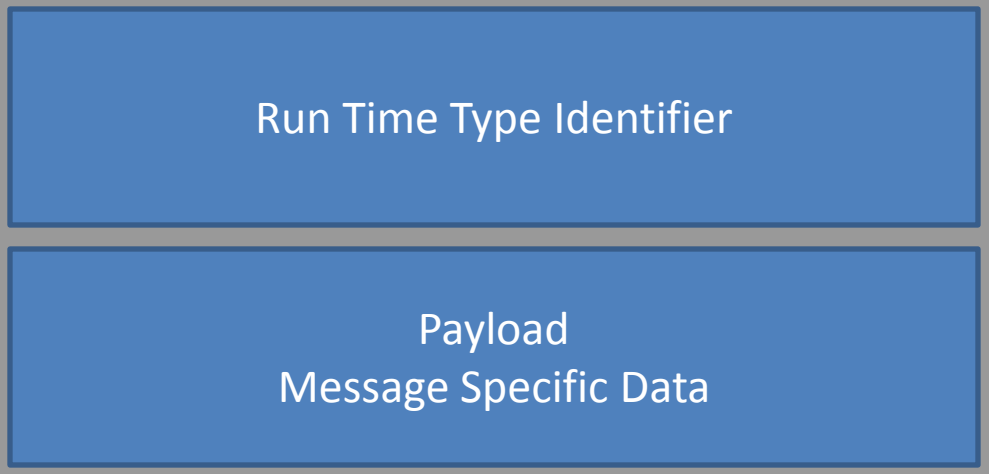
Scripting Languages

- Scripting languages relay entirely on non compiled function calls.
- Some languages use late binding.
- C++ has virtual functions which provide simple message abilities.

Why do we need them?

- Allows decoupled communication
  - Across game system boundaries
  - Across process boundaries
  - Across network boundaries
  - Across language boundaries
  - Across time!
- Reduces dependencies
  - Have to agree on format and message interface.
  - Enables very loose coupling.

Anatomy of a Message



Simple Object Oriented Messages

```
class Message
{
    MessageId Id;
    virtual ~Message(){};
};
//Making an individual message
const int MSG_ID_WEAPON_FIRE  = 35;
class WeaponFire : public Message
{
    WeaponFire(){ Id = MSG_ID_WEAPON_FIRE; };
    u32 Owner;
    u32 WeaponId;
    float x,y;
}
```

Simple Message Processing

```
void ProcessMessage(Message * msg)
{
     switch( msg->Id )
     {
          case MSG_ID_WEAPON_FIRE:
             ProcessWeaponFire( (WeaponFire*)msg );
          break;
     //Other messages
     }
};
```

Message Passing

- Messages can be passed through objects system as a black box.

```
void CompositeObject::SendMessage(Message * msg)
{
     foreach( child in children )
     {
        child->SendMessage(msg);
     }
};
```

Messages And Listeners

- Listener/Observer pattern
  - An object can sign up as an listener to receive events.
  - When an event is triggered a message is broadcasted to every listener
- Messages
  - Have an address or target object
  - Usually only sent to one interface or object
  - Can be broadcast or sent to every object
  - Some languages use duck typing

Messages and Listener

- Listener
  - Very effective in UI frameworks
    - C# delegates, ActionScript Events
  - Centralizes logic (this could be bad)
- Messages
  - Important in games and large frameworks
  - Decentralizes logic
- Not mutually exclusive

Message Queues

- Messages can be queued up to be processed at a later time. (Temporal Messages!)
- This is critical for networking and multithread applications.
- Also useful for AI and game logic.

Concrete vs Conceptual

- Concrete Messages
  - Linked to particular events.
  - Used for system communication.
  - OnCollide
  - AnimationFinish
- Conceptual Messages
  - Link to conceptual events object decides meaning.
  - Useful for making user game editors.
  - TakeDamage
  - TriggerFire
  - Activate

Signals and Slots

- Objects have a set of signals and slots.
- Signals are event that the object generates.
  - OnClick
  - OnCollide
  - OnClose
- Slots are functions that can be called on objects
  - Fire
  - Open
  - Unlock

Signals and Slots

- Setting up game logic is just connecting different signals to different slots.
- When this box is destroy unlock the door.
  - Connect(box.OnDestroy, door.Unlock);
- Powerful design for game logic

Message Extensions

- Record messages for playback.
  - Used to debug.
  - To Test Performance.
  - Make awesome videos.
- Use as command pattern
  - Store the message to be sent when the command is activated.

Improving Messages

- Templating can be used to reduce the amount of code it takes to make a message system.
- Macros can be used to clean up the switch statements.
- You can also use dll features or pdbs to link functions

Function Pointers with Map

```
class Object
{
    typedef void (Object::*MessageFunc)(Message * msg);
    std::map< MessageId , MessageFunc > MessageMap;
    void MapMesssages()
    {
        MessageMap[MSG_ID_WEAPON_FIRE] =
        MessageObject::ReciveWeaponFire;
    }
    void ReceiveWeaponFire(Message * msg)
    {
        WeaponFire * wfm = (WeaponFire * )msg;
    }
};
```

Messages as Functions

- Instead of creating a message object send data on a generic stack.
- The ID of the message can be a string.
- On the other end the data is removed from the stack.
- This interfaces very well with scripting languages.

Shared Objects

Direction Ownership

- Direct Ownership (Composition)
  - Every object is owned by exactly one parent object and access is controlled by the parent.
  - Effective, efficient, and powerful the default of C++.
  - Objects could also be in a tree structure.
- Also know as RAII or Resource acquisition is initialization

Shared Ownership

- Shared Ownership
  - Multiple objects share the object and its lifetime is not bound to any object.
  - When all objects no longer refer to the shared object it can be destroyed.
  - Objects do not have parent/child relationships only peer relationships.

Reference Counting

- Reference Counting
  - Every object has a count of the number of references (pointers) it has active. When all pointers are released and the count is zero the object is destroyed.
  - Efficient and prevents bad pointers.
  - Great for assets and transient objects.

Reference Counting!

```
void Sprite::Init()
{
     texture = GRAPHICS->Textures[textureFile];
     texture->AddRef();
};

void Sprite::~Sprite()
{
     texture->Release();
};
```

Game Objects

- Peer game objects need to be able to point to each other. (Missile -> Ship)
- Who owns the game objects?
  - Core System
  - Level

Shared Object Problems

```
void Missile::Update()
{
    if( WithinRange( this->targetObject ) )
        delete this->targetObject;//BOOM!
    else
        TrackTo(this->targetObject);
};
//Oh noes! This code is very crashy!
//who owns the target?
```

Reference Counting

```
void Missile::Update()
{
    if( WithinRange( this->targetObject ) )
        this->targetObject->Release();//Wait no boom
    else
        TrackTo(this->targetObject);
};
```

Reference Counting

```
void Missile::Update()
{
     if( this->targetObject->IsAlive() &&
        WithinRange( this->targetObject ) )
     {
          this->targetObject->MarkAsDead();
          this->targetObject->Release();
     }
     else
          TrackTo(this->targetObject);
};
//Now every object reference must have conditional death
//code and objects need a 'dead flag'
```

Reference Counting

```
void Ship::AntiMissileSystem()
{
    if( this->Missile == NULL )
    {
        this->Missile = GetObjectTrackingMe();
        this->Missile->AddRef(); //Prevent Crashes
    }
    else
        this->Missile->Jam();
};
//Something has gone horribly wrong
```

Reference Counting

- With just simple reference counting circular reference can be formed.
- If the dead flag is not checked you can get objects working with phantom objects.
- Objects will never be deleted (but they will be dead).

Garbage Collection

- The ‘Garbage collector’ scans all allocated objects, detects circular references and finally deletes objects.
- Every object not attached to the ‘root’ is destroyed.
- Use by Lua, C#, Java, etc.
- Huge topic beyond this scope.

Smart Pointers

- Can wrap reference counted objects. (objects that have their own reference count)
- Can have their own reference count. (the pointer tracks its own reference count)
- Scoped pointers. (automatically release pointers during destruction)
- Available in boost!

Handles / Object Ids

- Object Ids / Weak References
  - A reference to an object that does not effect its lifetime.
  - Owning code must handle the reference becoming null.
  - Used for peer to peer object referencing.

Handle Generation

```
GameObject* ObjectFactory::CreateObject(unsigned int ID)
{
    GameObject * newObj = BuildObject(ID);
    if (newObj)
    {
        this->LastObjectID += 1;
        newObj->Initialize(this->LastObjectID);
        this->ObjectMap[this->LastObjectID] = newObj;
    }
    return newObj;
};
```

Handles

```
void Missile::Update()
{
     GameObject* pTarget = GetObjectFromId(tID);
     if (pTarget)
        TrackToTarget(pTarget);
     else
        tID = FindNewObject();
};
```

Handle Benefits

- Ids are necessary for networking.
- Works for game objects because the reference going null case must be handled anyways.
- Useful for debugging the game world.
- Useful for serialization.

Destroying Objects

- Make your destructors private.
- Make your game objects friends of the factory.
- Use a “destroy” function to tell the factory to put the object on its “to-be-destroyed” list.
- This is “delayed destruction”, which has a number of benefits.

Delayed Destruction

- Objects destruction are delayed until the end of the frame or some other sync point.
- This prevents objects from being destroyed while a system is updating.

Destruction

```
void BOOM(int ID)
{
    GameObject* pObj = ObjectFactory->GetObject(ID);
    if (pObj) pObj->Destroy();
};
```

Destruction

```
void GameObject::Destroy()
{
    ObjectFactory->DestroyList.add( this );
}

void ObjectFactory::ClearDestroyList()
{
    foreach( GameObject * obj in DestroyList )
    {
        --NumberOfGameObjects;
        delete obj; //destructor handles cleanup
    }
}
```

Questions?