Michele Mischitelli

Unreal Engine 4: Delegates, Async and Subsystems

A follow up session on UE4’s async execution model
Main topics of this meetup

Delegates

Data types that reference and execute member functions on C++ objects

Asynchronous execution

Strategies and classes that allow devs to run asynchronous code using the UE4 framework

Subsystems

Automatically instantiated classes with managed lifetimes
Delegates

Type-safe dynamic binding of member functions
There are 4+2 types of delegates in UE4

- **Single**
  - A single function is bound to the delegate

- **Multicast**
  - Delegates that can be bound to multiple functions and execute them all at once

  **Dynamic Multicast**
  - Delegates that can be serialized and rely on reflection (instead of function pointers)

  **Events**
  - Similar to multicast, but only the class that declares it can broadcast

- **Dynamic**

- **Sparse**
  - 1-byte multicast implementation. Even slower than dynamic multicast

### Characteristics

- **Safe to copy**
  - Prefer passing by ref

- **Declared using MACROs**
  - In global scope
  - Inside a namespace
  - Within a class declaration

- **Support for signatures that**
  - Return a value
  - Are const
  - Have up to 8 arguments
  - Have up to 4 additional payloads
Single (or unicast) delegate type

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Binding</th>
<th>Usage</th>
</tr>
</thead>
</table>

```csharp
void Function()
DECLARE_DELEGATE( DelegateName )

void Function( <Param1> )
DECLARE_DELEGATE_OneParam( DelegateName, Param1Type )

void Function( <Param1>, ... )
DECLARE_DELEGATE_<Num>Params( DelegateName, Param1Type, ... )

<RetVal> Function()
DECLARE_DELEGATE_RetVal( RetValType, DelegateName )

<RetVal> Function( <Param1> )
DECLARE_DELEGATE_RetVal_OneParam( RetValType, DelegateName, Param1Type )

<RetVal> Function( <Param1>, ... )
DECLARE_DELEGATE_RetVal_<Num>Params( RetValType, DelegateName, Param1Type, ... )
```
Single (or unicast) delegate type

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Binding</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BindStatic(func, args...)</td>
<td>BindSP(objPtr, func, args...)</td>
<td>BindIfBound()</td>
</tr>
<tr>
<td>Binds a raw C++ pointer global function delegate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BindLambda(func, args...)</td>
<td>BindThreadSafeSP(...)</td>
<td>These keep a weak reference to your object. You can use ExecuteIfBound() to call them</td>
</tr>
<tr>
<td>Binds a C++ lambda delegate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technically this works for any functor types, but lambdas are the primary use case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BindRaw(obj*, func, args...)</td>
<td>BindUFunction(uObj*, funcName, args...)</td>
<td></td>
</tr>
<tr>
<td>Binds a raw C++ pointer delegate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw pointer doesn't use any sort of reference, so may be unsafe to call if the object was deleted. Be careful when calling Execute()!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BindUObject(uObj*, func, args...)</td>
<td>BindWeakLambda(obj*, func, args...)</td>
<td></td>
</tr>
<tr>
<td>UObject-based member function delegate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Just like the non-weak variant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single (or unicast) delegate type

### Declaration

```cpp
DECLARE_DELEGATE_OneParam(FDataIsReadyDelegate, float, value)
```

### Binding

```cpp
UCLASS()
class TEST_API UProducer : public UObject 
{
public:
    FDataIsReadyDelegate OnDataIsReady;
    void Register() {
        auto funName = GET_FUNCTION_NAME_CHECKED(UProducer, Receive);
        OnDataIsReady.BindUFunction(this, funName, true);
    }
    void Invoke() const {
        OnDataIsReady.ExecuteIfBound(10.0f);
    }
    UFUNCTION()
    void Receive(float arg1, bool payload1) { ... } 
};
```
Multicast delegate type

```csharp
void Function()
DECLARE_MULTICAST_DELEGATE( DelegateName )
void Function( <Param1> )
DECLARE_MULTICAST_DELEGATE_OneParam( DelegateName, Param1Type )
void Function( <Param1>, ... )
DECLARE_MULTICAST_DELEGATE_<Num>Params( DelegateName, Param1Type, ... )
```

Similar to unicast delegates, both in declaration and in usage

Can register multiple functions, thus binding methods are more array-like in semantics

Registered functions are stored in an invocation list

The order in which bound functions are called is not defined

`Broadcast()` is always safe to call
Dynamic delegate variants

```cpp
void Function()
DECLARE_DYNAMIC_DELEGATE( DelegateName )
void Function( <Param1> )
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam( DelegateName, Param1Type )
void Function( <Param1>, ... )
DECLARE_DYNAMIC_MULTICAST_DELEGATE_<Num>Params( DelegateName, Param1Type, ... )
```

Can be serialized

Functions can be found by name (reflection)

Slower than regular delegates as functions are found via reflection compared to C++ functors

Binding via helper macros AddDynamic(obj*, &Class::Func), BindDynamic(...), RemoveDynamic(...)

Executed via Execute(), ExecuteIfBound(), IsBound()
Event delegate type

```cpp
void Function()
DECLARE_EVENT( OwningType, EventName )
void Function( <Param1>, ... )
DECLARE_EVENT_<Num>Params( OwningType, EventName, Param1Type, ... )
void Function( <Param1>, ... )
DECLARE_DERIVED_EVENT( DerivedType, ParentType::PureEventName, OverriddenEventName )
```

It's a multicast delegate

Any class can bind to events but only the one that declares it may invoke `Broadcast()`, `IsBound()` and `Clear()` functions.

Event objects can be exposed in a public interface without worrying about who's going to call these functions.

Use case: callbacks in purely abstract classes.

`Broadcast()` is always safe to call.
Sparse dynamic multicast delegate type

void Function()
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE( DelegateClass, OwningType, DelegateName )
void Function( <Param1>, ... )
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_<Num>Params( ... )

It works just like a (slower) dynamic multicast delegate

Stores just a bool in the owner, signalling whether it’s bound or not

There’s a global static manager that stores:

- Delegate owner ptr
- Delegate name
- Multicast delegate
- <OwningType, DelegateName> pair
- Offset to delegate
Asynchronous execution

Synchronization primitives, containers and parallelization
Synchronization primitives

**Atomics**

- **FPlatformAtomics**
  - InterlockedAdd
  - InterlockedCompare{Exchange, Pointer}
  - Interlocked{Decrement, Increment}
  - InterlockedExchange[Ptr]
  - Interlocked{And, Or, Xor}

- **What are atomics?**
  - Operations that allow lockless concurrent programming
  - Atomic operations are indivisible
  - Are also free of data races

```cpp
class FThreadSafeCounter {
    volatile int32 m_Counter;
public:
    int32 Add(int32 value) {
        return FPlatformAtomics::InterlockedAdd(&m_Counter, value);
    }
};
```
Synchronization primitives

Critical Sections

- **FCriticalSection** synchronization object (mutex)
  - OS-independent: **PThreads** (Android, iOS, Mac, Unix), **CRITICAL_SECTION** (Windows, HoloLens)
- **FScopeLock(mutex*)** for scope level locking
  - The mutex is released in the scope lock’s destructor
  - Very useful to prevent deadlocks
  - Fast if the lock is not activated

```cpp
class FScopeLockTest {
    bool m_Toggle = false;
    FCriticalSection m_Mutex;

public:
    // Thread safe toggling
    void Toggle() {
        FScopeLock lock(m_Mutex);
        m_Toggle = !m_Toggle;
    }
};
```
Synchronization primitives

- **FSemaphore**
  - Like mutex with signalling mechanism
  - Only implemented for Windows and hardly used
  - Don’t use 😞
  - **FEvent** is there for you!
Synchronization primitives

- **FEvent**
  - Blocks a thread until triggered or timed out
  - Frequently used to wake up worker threads

- **FScopedEvent**
  - Wraps an FEvent that blocks on scope exit

```c
void SomeFunction
{
    FScopedEvent Event;
    DoWorkOnAnotherThread(Event.Get());
    // stalls here until the other thread calls Event.Trigger();
}
```
High level constructs

Containers

○ **General thread-safety info**
  - Most containers (TArray, TMap, etc.) are not thread safe
  - Use synchronization primitives if needed

○ **TLockFreePointerList**
  - Lock free, stack based and ABA resistant
  - Used by Task Graph system

○ **TQueue**
  - Uses a linked list under the hood
  - Lock and contention free for Single-Producer, Single-Consumer (SPSC)
  - Lock free for MPSC

Helpers

○ **ABA Problem (lock-free data structs)**
  - Process P1 reads value A from shared memory
  - P1 is put on hold while P2 is allowed to run
  - P2 modified the shared memory A to B and then back to A before P2 is put on hold
  - P1 continues execution without knowing that the memory has changed

○ **Lock vs contention**
  - Lock is one of the possible scenarios that cause contention
  - Contention can happen on lock-free resources as well: two threads atomically accessing some variable
  - The result is that one thread runs slower than the other one
High level constructs

Containers

○ **FThreadSafe**
  • Counter, Counter64, Int32, Int64, Bool

○ **TThreadSingleton**
  • Creates only one instance for each thread

○ **FMemStack**
  • Fast, temporary per-thread memory allocation

○ **TLockFreeClassAllocator, TLockFreeFixedSizeAllocator**
  • Thread safe, lock free pooling allocator of memory for instances of T

○ **FThreadIdleStats**
  • Measures how often a thread is idle
**Parallelization**

### Threads
- **FRunnable**
  - Platform-agnostic interface
  - Override just 4 methods: `Init`, `Run`, `Stop` and `Exit`
  - Launch with `RunnableThread::Create()`

- **AsyncPool (Global)**
  - Execute a given function on the specified thread pool

- **AsyncThread (Global)**
  - Execute a given function using a separate thread

### Processes
- **Game Thread**
  - All game code, Blueprints and UI
  - UObjects are not thread-safe

- **Render Thread**
  - Proxy objects for materials, primitives run in this one

- **Stats Thread**
  - Engine performance counters
Parallelization

- **Task based multithreading**
  - Small units of work are pushed to available worker threads
  - Tasks can have dependencies to each other
  - Task Graph will figure out order of execution
  - Used internally for a lot of things:
    - Animations, message dispatch, object reachability analysis in GC, render and physics subsystems...

- **AsyncTask (Global)**
  - Execute a given function on the task graph

- **ParallelFor**
  - General purpose parallel for that uses the task graph

```cpp
FConstructor taskCtor = TGraphTask<TAsyncGraphTask<ResultType>>::CreateTask();
taskCtor.ConstructAndDispatchWhenReady(args...); // This or even...
taskCtor.ConstructAndDispatchWhenReady(MoveTemp(func), MoveTemp(future));

// Or, for something a little bit different...
AsyncTask(ENamedThread::AnyNormalThreadNormalTask, [](()){ ... });
```
Parallelization

Threads  Task Graph  Processes  Messaging

- **FPlatformProcess**
  - CreateProc() executes an external program
  - LaunchURL() launches the default program for a URL
  - IsProcRunning() checks whether a process is running
  - And many more utils for process management

- **FMonitoredProcess**
  - Convenience class for keeping track of some process
  - Even delegates for cancellation, competition and output

```cpp
FMonitoredProcess Process(*Executable, *Arguments, true/*hidden*/, true/*piped out*/);
Process.OnOutput().BindLambda([](){ ... });
Process.Launch();

while(Process.Update()) {
    ...
}
```
Parallelization

Threads ➔ Task Graph ➔ Processes ➔ Messaging

- **Unreal Message Bus (UMB)**
  - Zero configuration intra/inter-process communication
  - Request-Reply and Publish-Subscribe patterns
  - Messages are simple UStructs
  - Notable classes: FMessageBus, FMessageRouter, FMessageEndpoint

- **IMessageTransport**
  - Seamlessly connect processes across machines
  - Can use this interface to implement custom network protocols or API
  - Implemented for TCP and UDP for the moment

- **FGenericPlatformNamedPipe**
  - Yeah, named pipes..

```cpp
auto Endpoint = FMessageEndpoint::Builder(TEXT("SomeName"))
  .ReceivingOnThread(ENamedThread::GameThread)
  .WithCatchall(this, &FMyEndpoint::InternalHandleMessage)
  .NotificationHandling(FOnBusNotification::CreateRaw(this, &FMyEndpoint::OnNotify));

Endpoint->Subscribe(MessageTypeFName, EMessageScope::Thread | EMessageScope::Network);
Endpoint->Send(...);
```
Subsystems

Architectural pattern to better organize code
Subsystems intro

**Automatically instanced**
- Instantiated, initialized and destroyed by the engine
- No need to wire up systems to spawn and track this object

**Managed lifetime**
- Five different ones to choose from
- Multiple instances of the same object if it makes sense for the chosen lifetime

**WHY ?!**
- Architectural pattern
- Improved modularity
- Especially useful in plugins
- Save both programming time AND lines of code
Subsystem lifetimes / types

The base class you derive from determines also the lifetime of your subsystem

- **Game-centric Subsystems**
  - UGameInstanceSubsystem: lives before the world. Persists when changing levels (maps) in the game
  - ULocalPlayerSubsystem: each player active on the current client is represented by an instance of ULocalPlayer
  - UWorldSubsystem: a world can be a single persistent level with a list of streaming levels or composition of worlds

- **Advanced Subsystems**
  - UEngineSubsystem
  - UEditorSubsystem
Subsystem example

```cpp
//CLASS(DisplayName = "PrinterSubsystem")
class MEETUPNOV2019_API UPrinterSubsystem : public UGameInstanceSubsystem
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintSetter = SetColor, BlueprintGetter = GetColor, meta = (DisplayName="Color", AllowPrivateAccess=true))
    FColor m_Color = FColor::Yellow;
    UPROPERTY(EditAnywhere, BlueprintSetter = SetLifetime, BlueprintGetter = GetLifetime, meta = (DisplayName = "Lifetime", AllowPrivateAccess = true))
    float m_Lifetime = 4.0f;

public:

    UFUNCTION(BlueprintCallable, Category = PrinterSubsystem)
    void PrintString(const FString& str) const;
    void PrintString(uint64 key, const FString& str) const;

    UFUNCTION(BlueprintCallable, Category = PrinterSubsystem)
    void SetColor(const FColor& color) { m_Color = color; }
    UFUNCTION(BlueprintCallable, Category = PrinterSubsystem)
    FColor GetColor() const { return m_Color; }

    UFUNCTION(BlueprintCallable, Category = PrinterSubsystem)
    void SetLifetime(float duration) { m_Lifetime = duration; }
    UFUNCTION(BlueprintCallable, Category = PrinterSubsystem)
    float GetLifetime() const { return m_Lifetime; }
};

void UProducerSubsystem::Initialize(FSubsystemCollectionBase& Collection)
{
    // Tells Unreal that this subsystem depends on UPrinterSubsystem
    Collection.InitializeDependency(UPrinterSubsystem::StaticClass());
}
```
Thank you

LinkedIn: linkedin.com/in/michelemischitelli
Twitter: twitter.com/michelemischit1
Email: michelemischitelli@outlook.com
Website: mmischitelli.github.io