Signals and Slots

Stephen Waits

November 10, 2006

This brief document provides an overview of the event and message handling system written for and used in the SCEA NBA PSP projects.

1 Goals

- Basic synchronous messaging
- Decouple game modules
- Appropriate parameter handling (i.e. no blind parms, casting parms, etc.)
- Ease of use
- Lightweight
- Strive for elegant, clean, readable, and maintainable code

2 History

When I started researching event messaging systems and libraries in early 2004, I initially looked at what Qt and other similar libraries used. I ended up discovering libsigc++, which was my introduction to the signal/slot concept. Unfortunately, libsigc++, while fascinating, was much too big for my needs, so, I set out to write my own, simpler, lighter library.

Once I got the library predominately functional, we integrated it into our codebase. I found that it took very little explanation to bring other programmers up to speed on the concepts and practical use of signals and slots. After using them for a bit, everyone on the team agreed that we had benefited greatly from adopting this simple concept throughout our code. The signals rocked for us.

Today, I can confidently say that this single library alone bears great responsibility in the overall cleanliness and readability of our codebase. Without it, we'd have a giant pile of spaghetti by now. Instead, we have clear, simple, decoupled systems which all communicate via a common interface.

3 Usage

Signals are effectively emitters. Slots are effectively callback functions, methods, or functors.

```
// create a signal
Signal< void, int /* x */, int /* y */ > OnClick;

// A function matching the signal's type
// (returns void, accepts 2 ints)
void beep(int x, int y) { ... }

// A class method matching the signal's type
// (returns void, accepts 2 ints)
struct Bark { void woof(int x, int y); }
Bark barker;

// connect a regular function
OnClick.Connect( beep ); // beep becomes a Slot<void,int,int>

// connect a member of a specific object
OnClick.Connect( &barker, &Bark::woof ); // woof is a Slot

// emit
OnClick(10,20); // calls beep(10,20) and barker.woof(10,20)

// disconnect
OnClick.Disconnect( beep );
OnClick.Disconnect( &barker, &Bark::woof );

// note that this won't compile (type safety)
void badbeep(int x, float y); // parameter mismatch
OnClick.Connect( badbeep ); // compile-time error!
```

4 In Practice

In our projects, we primarily use signals in two ways.

First, we maintain a master list of global signals. These handle things such as a player stepping out of bounds, the ball hitting the backboard, a team calling timeout, and, basically everything that happens in basketball. This list of signals has grown quite large, but remains organized.

Second, some classes include public signals. For example, class Clock might have a public signal named "Changed" which gets emitted any time its internal time is updated. The Clock object doesn't have to care if anyone connects - its responsibility ends at emission. Our UI system also makes heavy use of signals.

5 Summary

5.1 Positives

- Simple code is easy to use
- Emitter/Subscriber model is rather nice to use and allows for massive decoupling
- Synchronous emission
- Static type safety (this is a huge plus!!!)
- Very easy to add to your project (it's almost entirely a template library)

5.2 Negatives

- Current implementation does use some dynamic memory
- Requires RTTI[1]; though only on Connect/Disconnect/IsConnected calls (normally infrequent)
- Not a negative to most, but does use some template-fu to keep things easy on the outside
- Related to the template-fu, they can bloat; however, we've done our best to minimize this
- Return value heuristic never reached (i.e. what does the emitter return when calling multiple slots?)

[1] RTTI is commonly thought to be the devil amongst uninformed game programmers; however, the only expense is one of a very slight size increase. Performance, possible slight caching consequences aside, generally remains the same. The convenience, in my opinion, vastly outweighs the cost.