Stacks Postmortem

Sjors van Gelderen

07-07-2016

Development tools

The various tools used during the development process.
The prototype was developed on *Arch Linux* and later *Ubuntu GNOME Edition 16.04*.

Engine

- Unity3D 5.4.0b16, experimental Linux version

Text/code editors

- GNU Emacs 24
- Visual Studio Code 1.0.0
- MonoDevelop 5.10
- Atom 1.6.2
- Processing 3.0.2
- Gedit 3.18.3

With the support of the Ionide framework.

Other

- LibreOffice 5.1.4.2
- Cleaver 0.8.2
- Evolus Pencil 2.0.5
- Remarkable 1.75
- Abricotine 0.3.3

Architecture

Stacks

This is where the core game logic and the data model are located.
It features a custom version of Monaco, the monadic coroutine, monadic state and entity closure framework.
This project produces a .dll file which exposes various components that may be used by Unity.
The way to define these exposed components is to write classes that inherit from MonoBehaviour; a core element of Unity game objects.

In order to run F# in Unity, the core .dll must be provided.
The .dlls should be placed in the Assets/Framework folder in the Unity project.

Stacks Unity

The main Unity project. Contains all assets and plugins.
This is where the interface and visual environment are set up.
A collection of C# scripts handles most of the UI logic,
though a large chunk of it is defined in various inspector windows.
Eventually, a build which includes the Stacks and F# .dlls may be produced from this project for several platforms.

Challenges

Because of my interest in researching F# for interactive software,
the separate Monaco framework was built, customized and included in the Unity project.
Though this framework is elegant and very powerful, and though F# is an excellent programming language,
this configuration produces several technical challenges.

There is a distinct lack of documentation and support for combining F# with Unity.
Several community tutorials helped me with the configuration process.
Unfortunately, some experimentation was required beside the provided information.
Eventually I had a working setup and learned how to manipulate Unity game objects from within the F# project.

Unity is developed with C# and object oriented programming in mind.
The base object in Unity is the game object.
One may attach to these a collection of components including, but not limited to:

- Transforms
- Renderers
- Scripts

Manipulation of these components often means mutating state.
This is fundamentally against the principles of our monadic framework.
The whole point of the framework is to simulate mutation while using completely immutable data.

Since almost every action in Unity relates to some form of mutation,
any interaction with game objects should occur through messages.
This is a very tedious process.

To circumvent the necessity for this inefficient communication,
C# scripts are in charge of any direct mutation.

One might wonder whether there is any real gain from the immutable logic at all.
The logic is obviously more reliable, but in this project it has too few responsibilities to

justify the complexity it introduces.
Common manipulations in Unity aren’t processed in a safe manner anyway.

Furthermore, in this game the user constantly interferes with the data processed by the logic.
The C# scripts can access data from the F# framework, but not the other way around.
This is because the F# framework is compiled into a .dll that becomes part of the Unity C# project.

Since direct manipulation of the logic is not supposed to happen,
the C# script calls functions from the F# project that queue data in lists.
When the logic is ready, it can import data from these lists and subsequently clear them.

This is an awkward mode of communication that still isn’t really safe.
The queues are manipulated by separate processes, and it is difficult to determine exactly when they are accessing the data.
As a result of this, data may be lost before it is processed.

I believe there are efficient and graceful solutions to this problem,
but with my limited experience and time, I was unable to discover them.

Another problem is that the Unity inspector expects common C# data types,
which are different from those commonly used in F# in that they perform mutation.
The inspector is unable to display the common F# data types without extending the editor itself.

Conclusion

These are just some of the challenges I faced with this configuration.
I recommend working completely with C#, which already introduces much in the way of functional programming.
Using design patterns, many common issues may be circumvented.
It may not be quite as reliable or graceful as F#, but it is completely workable for this relatively small-scale project.