A Functional Approach to Games

# Abstract

In this work we describe how to use functional programming in the large in the context of making games. Our goal is to build the logic part of a game (that is, we do not care about graphics and drawing) by gaining the kind of clean and concise code we would expect from a functional program. We also set the (rather ambitious) goal of not making compromises in the following areas:

1. Similarity of our solutions to “mainstream” OO languages, which are the current standard in the area
2. Appearance of impure computations
3. Purity
4. Ease of writing
5. Integration of external, and possibly impure, libraries

## What we leave behind

We do not expect to be able to maintain full type-safety. While this may appear to be a major drawback, the reason for this is that to respect the requirement of interacting with as many external libraries as possible, we feel the need to pick a programming language which is part of some larger, commercial framework such as the .Net framework. For this reason we pick F# which, though being a powerful and modern functional programming language, misses a type system as powerful as that found in other languages such as Haskell. Thus we will proceed with emulating a more powerful type-system through reflection, thereby forsaking (even though just locally) type-safety but otherwise achieving higher expressivity.

We try to ameliorate this fact by using reflection in a way such that wrong usage of our reflective operators will throw an exception right on creation that is our reflective operators will be as strict as possible in the evaluation of reflective operations.

# Introduction

We start by describing why games are an interesting benchmark. We do so in the spirit of understanding what kind of challenges PLs can help for developers to face. A conceptual framework is built to describe how we can create games with the properties we desire. An example is discussed and the framework is applied: first we describe the particular game we will use as a benchmark; secondly we apply our framework to the sample game and discuss the results. At this point we discuss our conclusions and the future expansions to our framework.

# Games as an interesting benchmark

1. Games have always represented a big challenge to developers.
2. In games we can find aspects of most disciplines of CS.
3. Games are heavy on math and algorithms 🡪 appropriate languages are needed (fun).
4. Games are rather large projects 🡪 appropriate languages are needed (OO).
5. Games require high-performance 🡪 parallelism is needed.
6. Games interact with the user 🡪 concurrency is needed.

# Game Structure

The game is separated in two bigger components. The first component is the game logic, which is what we focus on because we believe it is the place where the most benefits can be reaped. The second component is the drawing and reading of user input; being drawing and reading user input purely stateful computations we believe the advantages of turning these into pure computations are much lower. Moreover, modeling IO in a pure functional language is definitely a solved problem.

A far more interesting additional aspect on which to focus is the interaction between the game logic, the user input and the drawing components. Executing these operations concurrently yields a great performance benefit, and is made far simpler to program by strong guarantees such as the purity of the game logic component (and the subsequent immutability of the game logic value).

LOGIC

COMPONENT

CONCURRENCY MANAGER

INPUT

COMPONENT

RENDERER

The game logic is composed of two functions:

The type variable is the type of the game state; the first value of the game state is obtained by calling initialize when the game starts. Each frame is called to update the current value of the game state, and to draw the new game state.

The second type parameter represents a time span. The value of type passed to the function is the amount of time passed since the last call to update; this parameter is required for all those computations that integrate over time, such as physics and AI.

The last type parameter is the type of the user input; values of type contain user interactions, such as keystrokes and mouse movements.

The function returns the first game state, and is invoked before the first frame, when the game starts. The function is invoked once per frame and returns the updated state which will be rendered on screen and then updated in the subsequent update operation. At a very high level of abstraction, we could model the game as a whole as an (impure) computation:

Notice that the function is not pure, as suggested by its signature: it takes as input the current state and draws it to the screen. The execution of a game can be seen as a function like the following:

The function is impure, but we believe this may be considered as a necessary evil: a game interacts heavily with the user, both in terms of input and output. To obtain an interactive frame-rate, every few milliseconds it is needed to re-draw the entire screen. Rather than model the *entire* game as a pure computation, we prefer to isolate the more impure portions and leave them as such, while focusing on those aspects of the game that would most benefit from full or partial purity. In particular, we will model the function as executing a stream of update operations concurrently with a stream of draw operations. To greatly simplify this (otherwise exceedingly difficult) task, we will model the function as a fully pure computation. Concurrency in a pure context allows for simple and elegant solutions, without recurring to locks and with strong correctness guarantees.

# The update function

The function easily tends to become quite complex. Its work is to update the game physics, AI and logic triggers and activators, in addition to meaningfully integrating the user input into the game state. We believe that such a task could not be achieved without a large dose of software engineering: dividing the operation (and the game state) into smaller units and composing these is fundamental to achieve clear, concise and easily extensible solutions. We could express updating as a sequence of sub-update operations, as in:

This simple solution is not particularly desirable. Its first problem is that no feedback is given from one function to the next beyond the updated state, resulting in no subdivision of intermediate computations: each function must be an entire updating function, and so while we are dividing the computations into smaller ones, the merging of said computations is practically absent. The second problem of this approach is that each function updates the entire state (we have no way of knowing otherwise), and as such there is no true subdivision of tasks: it might be advantageous to have smaller functions that update just a portion of the state, and to express this with typing relationships. Lastly, this approach is error prone: we have no guarantee that the function will be called to update the state ; imagine if we wrote:

A good abstraction system should probably make this mistake impossible to make. Also, note that in an impure, imperative, object-oriented context this mistake could not be easily made:

We believe that for functional programming to further gain real-world interest its manifest advantages must become available to developers without losing the advantages of the mainstream programming languages they are currently using.

1. We base most of our work on monads
   1. Higher degree of control
2. We use a slight variation of the state monad to share information and give an appearance of stateful computations
3. We completely skip the OO portions of the host language, and instead we recreate a simple OO framework which is compatible with our state monad
4. We hide from the functional structure mutable values such as those belonging to external libraries (like physics)
   1. Hash/comparisons
   2. Copies of essential fields
5. We augment types belonging to external libraries to make them “play well” with our functional code
   1. Hash/comparisons