Easel: Purely Functional Game Programming

Bryant Nelson, Joshua Archer, Nelson Rushton
(bryant.nelson | josh.archer | nelson.rushton) @ ttu.edu
Dept. of Computer Science, Texas Tech University
Box 43104 Lubbock, TX 79409-3104

Abstract – *In response to a growing interest in functional programming and its use in game development we've developed the Easel Framework which describes an engine for creating real time games by defining pure functions. This paper describes the framework and an implementation of this framework in SequenceL.*

Keywords: SequenceL, Easel, Game Programming, Functional Programming, Parallel Programming

1 Introduction

In his keynote address at Quakecon 2013, John Carmack, founder of Id Software and creator of the computer games Doom and Quake, shared his views on functional programming within the realm of video game development [1]. Carmack expressed that the use of pure functions simplifies the code base for very large projects by, among other things, ensuring that various parts of the software do not interfere with each other. The benefit of the modularity inherent in functional programming is something that has been known for some time [2]. This paper describes a game programming framework which allows games to be written in the pure functional language SequenceL [3], and an implementation of that framework in C#. The framework is called Easel and can be used to test Carmack's hypothesis in its purest form, by writing games without producing any new procedural code, or code with side effects.

There have been previous attempts at making a purely functional game programming framework. A fairly popular example is the ELM language developed by Evan Czaplicki [4]. In the case of ELM, an entirely new programming language was developed in an attempt to facilitate the creation of responsive GUIs using a functional language. There are also examples of using Haskell to program games [5].

These previous attempts at functional game engines try to handle everything, from the I/O and rendering to the game logic, in a functional language. Our opinion is that this leads to unintuitive engines, and complex game programs. Easel is an attempt to distill the logic of real time games down to its simplest form using a functional language, and then handle the rendering of these games in a procedural language.

2 Easel Description

The goal of Easel is to enable the creation of real-time games by defining pure functions. The Easel Framework is a system description for a game engine which has two key parts, a functional language used to write games for the engine, called the game implementation language, and a program which runs games written in that language, called the rendering backend.

2.1 Overview

The Easel framework requires a built-in data model, consisting of the following types, to be defined in the game implementation language:

- *Point* -- a structure of the form `(x: int, y:int)`
- *Color* -- a structure of the form
  `(red: int, blue: int, green: int)`
  with $0 \leq$ red, blue, green ≤ 255
- *ImageType* -- one of the following strings:
  "segment", "circle", "text", "disc", "triangle", or "graphic"
- *Image* – a structure of the form:
  ```
  (kind:ImageType, iColor:Color,
  vert1:Point, vert2:Point,
  vert3:Point, center:Point,
  radius:int, height:int, width:int,
  message:string, src:string)
  ```

  In practice only a subset of the fields will be used to define a specific kind of image, and there are six kinds of images:
  - *Segment* --
    `(kind:"segment",vert1:Point, vert2:Point, iColor:Color)`
  - *Circle* --
    `(kind:"circle",center:Point, radius:int, iColor: Color)`
  - *Disc* --
    `(kind: "disc", center:Point, radius:int, iColor:Color)`
  - *FilledTriangle* --
    `(kind:"triangle",vert1:Point, vert2:Point, vert3:Point, iColor: Color)`

- ○ *ImgFile* -- `(kind:"graphic",source:string, center:Point, height: int, width: int)`

- *Sprite* – a sequence of images
- *Click* -- a structure of the form `(clicked: bool, clPoint: Point)`
  If clicked is false then this interpreted that there was no mouse click in the given frame.
  If clicked is true, then point is the mouse click for the frame.
- *Input* -- a structure of the form `(iClick: Click, keys: String)`
  This is interpreted as the input vector for a given frame, consisting of a possible mouse click and a sequence ascii of codes of pressed keys.
- *Sound* -- a string which is "ding", "bang", "boing", "clap", or "click", or the name of a .wav or .mp3 file.

To create a game, the game implementation language is used to define the following type and functions:

- `State` -- a data type whose instances are possible states of the game
- `initialState()` -- the starting state of the game
- `images(`*`S: State`*`)` -- is a sequence whose members are the images to be displayed in the program window when the game is in state *S*.
- `sounds(`*`I: Input, S: State`*`)` -- a sequence of sounds played when input *I* is accepted in state *S*.
- `newState(`*`I: Input, S: State`*`)` -- the new state resulting from accepting input *I* in state *S*.

The rendering backend is responsible for executing the game, retrieving input from players, and displaying the images and sounds from the game. The overall algorithm for the rendering backend is presented in Figure 1.

```
S := initialState()
while True:
    display images(S)
    retrieve userInput
    play sounds(userInput, S)
    S := newState(userInput, S)
```

Figure 1: Algorithm `PlayGame`

The `PlayGame` algorithm is very similar to the standard game loop that is often encoded by hand when writing a game. The Easel engine, however, removes the need to write a main game loop. This abstraction allows the game programmer to focus on the logic of the game and not have to worry about the details of handling input from the user and rendering graphics to the screen.

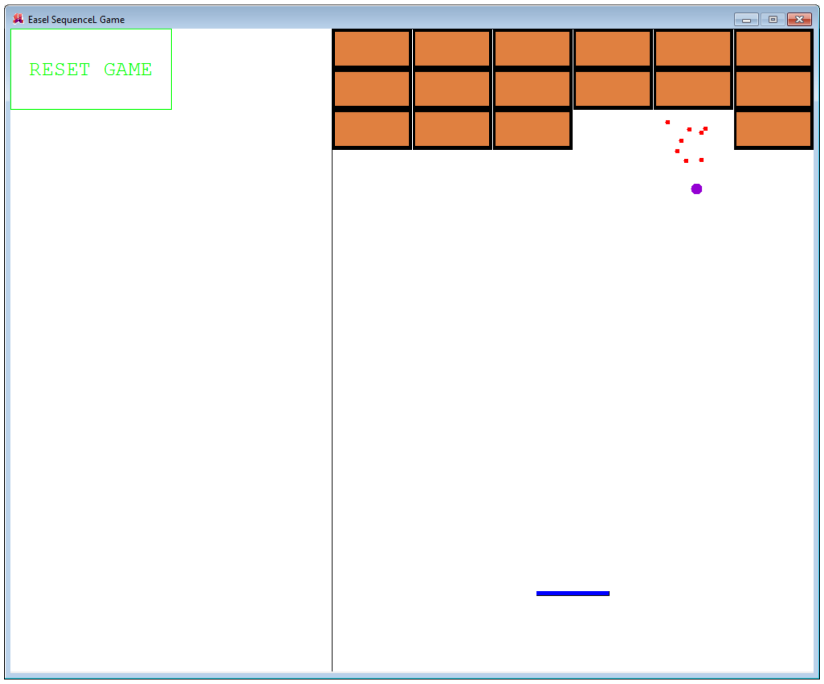


Figure 2: C# Rendering GUI Playing Breakout

2.2 Implementation

An Easel framework has been implemented, consisting of a rendering backend and graphical frontend written in C#, which runs games implemented using the Easel Framework in SequenceL. This implementation is referred to as Easel$_{SL}$.

SequenceL is a small, statically typed, general purpose, functional programming language [3]. The key reason for which SequenceL was chosen as the game implementation language is the fact that it is purely functional. SequenceL compiles to C++ code, allowing it to be easily interfaced with a graphical front end, which is a requirement for a game engine. Additionally, programs written in SequenceL are automatically compiled to highly parallel C++ [6]. All of these reasons contributed to the choice of SequenceL as the game implementation language.

Figure 3 shows a simple example of the SequenceL function definitions needed to encode a very simple game in the Easel$_{SL}$ game engine. The "game" simply displays the current time.

2.3 Rendering Backend

C# was chosen to implement the rendering backend, due to its extensive libraries and ease of graphical development.

An obvious drawback of this choice is that the framework is restricted to the Windows operating system.

The GDI+ libraries were used to render the graphics from the games. These libraries provide access to the standard Windows graphics API. They are not high-performance, but they have performed adequately thus far.

When a user runs the game engine, they are queried for the location of a game file written in SequenceL. The SequenceL compiler is then called to compile the game source file into C++ code. The Visual Studio C++ compiler is then called to compile that C++ code into a C++ DLL. The game engine is then able to access the Easel functions and execute the game.

The game engine runs the `PlayGame` algorithm until the player interrupts it by exiting the application.

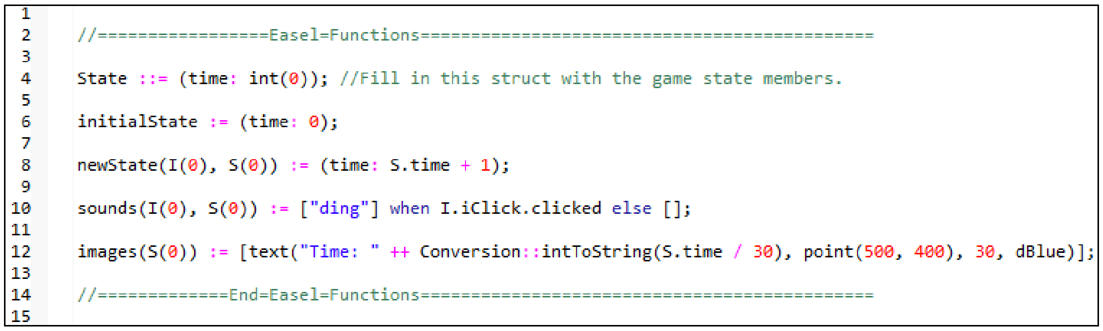
3 Conclusions

Several simple games have now been written using the Easel$_{SL}$ engine. These games range from Tic-Tac-Toe to Breakout. The engine was in fact used in an undergraduate Concepts of Programming Languages course at Texas Tech University to provide students with hands-on experience using a functional language for game programming.

It has become apparent that there are some inherent limitations in the design of Easel. One such limitation is that s the state gets large, as in most games of considerable size, the new state becomes too large to efficiently pass by value. In addition, all game actions that affect a single intuitive state variable must be located in one place together, which can be unintuitive in complex games.

4 Future Work

Work is currently under way to extend the current game engine to directly support 3D rendering. The project has also inspired research into what is currently being called Concrete State Machine Language (CSML). Future work includes the use of abstract state machines calling SequenceL functions to address the limitations discussed in the previous section.



```

//================Easel=Functions============================================

State ::= (time: int(0)); //Fill in this struct with the game state members.

initialState := (time: 0);

newState(I(0), S(0)) := (time: S.time + 1);

sounds(I(0), S(0)) := ["ding"] when I.iClick.clicked else [];

images(S(0)) := [text("Time: " ++ Conversion::intToString(S.time / 30), point(500, 400), 30, dBlue)];

//============End=Easel=Functions============================================

```

Figure 3: Simple Easel Game in SequenceL

5 References

[1] John Carmack's keynote at Quakecon 2013 part 4. 2013, http://youtu.be/1PhArSujR_A.

[2] J. Hughes, "Why Functional Programming Matters," in The Computer Journal - Special issue on Lazy functional programming archive Volume 32 Issue 2, April 1989, pp. 98-107.

[3] B. Nemanich, D. Cooke, and J. N. Rushton, "SequenceL: transparency and multi-core parallelisms," in Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming, 2010, pp. 45–52.

[4] E. Czaplick, "Elm: Concurrent FRP for Functional GUI", Master's Thesis, Harvard School of Engineering and Applied Sciences, www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf, March, 2012.

[5] M. H. Cheong, "Functional Programming and 3D Games," Master's Thesis, The University of New South Wales School of Computer Science and Engineering, www.cse.unsw.edu.au/~pls/thesis/munc-thesis.pdf, 2005.

[6] B. Nelson and J. N. Rushton, "Fully Automatic Parallel Programming," presented at the Worldcomp 2013, at The 2013 International Conference on Foundations of Computer Science, 2013.